Optimal Algorithms for GSM Viterbi Modules

Kehuai Wu
M.Sc. Student
at
Department of Informatics and Mathematical Modelling
Technical University of Denmark

Optimal Algorithms for GSM Viterbi Modules

Optimal Algorithms for GSM Viterbi Modules

Kehuai Wu
M.Sc. Student
at
Department of Informatics and Mathematical Modelling
Technical University of Denmark

July 30, 2003

Abstract

A power/area optimum design of the 3rd-Generation Global System for Mobile communications(GSM 3G) unit's channel code decoder has been described. This decoder can perform the convolutional code decoding and the CRC check for a burst of transmitted data encoded by other GSM 3G units. The constraint length of the decoder varies between 5 and 7, the code rate varies from 2 to 6, and the received data are calibrated into 4 or 5-bit soft-decision bits. The power and area optimization is only considered at the algorithm and architecture level. The power/area efficiency is measured from the gate-level synthesis result. When the decoder is clocked at 100MHz and is given a 1.3V voltage supply, the measurement results show that the decoder's power consumption is typically less than 1.4mW. The decoder core consists of approximately 35.7 k gates (142.8 k transistors), which is equivalent to the area of $0.24mm^2$ *for the given 0.09 μm 5-metal-layer CMOS technology.*

Keywords: *Convolutional decoder, Viterbi algorithm, Low power, Low area, VLSI, GSM.*

Contents

1 Introduction . . . 12
1.1 Background . . . 12
1.1.1 Mobile phones and 3G . . . 12
1.1.2 Communication and channel coding . . . 12
1.2 Overview . . . 13
1.3 Document organization . . . 14

2 Technology overview . . . 16
2.1 Digital communication methodology . . . 16
2.1.1 Data transmission . . . 16
2.1.2 Data receiving . . . 17
2.2 Channel coding . . . 17
2.2.1 Block code . . . 17
2.2.2 Convolutional coding . . . 20
2.2.3 Interleaving . . . 23
2.3 Viterbi algorithm . . . 24
2.3.1 Encoding state diagram . . . 24
2.3.2 The Trellis encoding diagram . . . 24
2.3.3 Convolutional decoder . . . 25
2.4 Advanced Viterbi algorithm . . . 31
2.4.1 Metrics with soft-decision bit . . . 31
2.4.2 Threshold based Viterbi algorithm . . . 32
2.5 GSM standard . . . 33
2.5.1 Generator polynomials . . . 33
2.5.2 Punctured data and trace back . . . 35
2.6 Low power CMOS design . . . 35
2.6.1 Power dissipation in CMOS circuit . . . 35
2.6.2 Low power design methodology . . . 36

3 Decoder design . . . 38
3.1 Requirements . . . 38
3.1.1 Coding scheme and structure . . . 38
3.1.2 Viterbi decoder . . . 39
3.1.3 Cyclic decoder . . . 41
3.1.4 Data storage and control signal . . . 42
3.1.5 Timing requirements . . . 43
3.1.6 Interface design . . . 43
3.2 Decoder structure design . . . 44
3.2.1 Structure design . . . 44

3.2.2 State transition . . . 46
3.2.3 Function description . . . 47
3.3 Viterbi unit design . . . 48
3.3.1 Decoding scenario . . . 49
3.3.2 Decoder structure . . . 50
3.3.3 State diagram and control . . . 52
3.3.4 Path extension unit . . . 54
3.3.5 Algorithms . . . 59
3.4 CRC unit design . . . 69
3.4.1 State transition of the cyclic decoder . . . 69
3.4.2 Structure of the cyclic decoder . . . 70
3.4.3 Structure of the configurable decoder . . . 71

4 Design evaluation . . . 74
4.1 Test bench . . . 74
4.2 Function verification . . . 74
4.3 Timing verification . . . 76
4.3.1 Gate level delay . . . 76
4.3.2 Layout level delay . . . 76
4.3.3 Unit delay . . . 76
4.4 Area cost . . . 76
4.5 Power consumption VS Threshold . . . 77
4.5.1 SYNOPSYS power compiler . . . 77
4.5.2 Decoding dynamics . . . 80

5 Future work . . . 82
5.1 Path metric storage power optimization . . . 82
5.1.1 Structure modification . . . 82
5.1.2 Gray code indexing . . . 83
5.2 Redundant path merge unit . . . 85
5.3 Reduced temporary storage . . . 85

6 Conclusion . . . 88

Appendix 91

A Source code . . . 93
A.1 Decoder source code . . . 93
A.1.1 Decoder unit . . . 93
A.1.2 Decoder control unit . . . 97
A.1.3 Input buffer . . . 102

A.1.4 Viterbi decoder 103
A.1.5 Viterbi control unit 111
A.1.6 Trace back memory 125
A.1.7 Soft bit decoder 127
A.1.8 Find new index 130
A.1.9 Path extension unit 131
A.1.10 Even memory 140
A.1.11 Odd memory 141
A.1.12 Read PM 142
A.1.13 Path prune 144
A.1.14 ACS cc 145
A.1.15 Pipeline 1 149
A.1.16 ACS BM 152
A.1.17 ACS PM 153
A.1.18 Pipeline 2 155
A.1.19 TB update 157
A.1.20 MAX 173
A.1.21 Write PM 175
A.1.22 Pipeline 3 176
A.1.23 Cyclic decoder 177
A.1.24 Cyclic decoder control 180
A.1.25 Shift chain 190
A.2 Test bench 207
A.2.1 Decoder stimuli 207
A.2.2 Test bench entity 233
A.2.3 Clock and reset generator 234
A.2.4 Test configuration file 235
A.3 Gray code test 237
A.3.1 Entity declaration 237
A.3.2 Gray code read 239
A.3.3 Gray code write 240
A.3.4 Write enable 240
A.3.5 Write DEMUX 241
A.3.6 Storage 1 242
A.3.7 Storage 2 252
A.3.8 Read MUX 263
A.3.9 Output 264
A.3.10 Test bench and stimuli 264

B Data sheet 269
B.1 GL00624032040 data sheet 269
B.2 MG00032010020 data sheet 295

C Scripts . . . 306
C.1 Forward SAIF file creation . . . 306
C.2 Activity monitoring . . . 306
C.3 Backward SAIF file creation . . . 307
C.4 Power analysis and report generation . . . 308

List of Figures

2.1 Data communication methodology 16
2.2 Channel coding block diagram 18
2.3 Systematic n-stage cyclic encoder 18
2.4 Systematic n-stage cyclic decoder 19
2.5 Convolutional encoder input-output correlation 21
2.6 Convolutional encoder (r=1/2 K=3) 21
2.7 RSC encoder (r=1/2 K=3) 22
2.8 Interleaving circuit 23
2.9 Encoder state machine 25
2.10 Trellis diagram 26
2.11 Trellis encoding diagram 26
2.12 Trellis decoding diagram 27
2.13 Path metric 27
2.14 Path merge 28
2.15 Butterfly unit of non-RSC code 29
2.16 Trace back 30
2.17 Input-output transfer curves for hard-decision and 4-bit soft-bit decision ADC . . 32
2.18 Low power design methodology[14] [7] 37
3.1 The structure of decoder 38
3.2 Soft-decision coded data for each decoding stage contained in one word 40
3.3 32 decoded bits merged in one word 41
3.4 32 encoded bits merged in one word 42
3.5 30 bits of control data merged into one word 43
3.6 The interface between decoder and DSP unit 43
3.7 The signal waveform of control signal 44
3.8 The top level structure of the decoder 45
3.9 The state diagram of the decoder 46
3.10 The structure of the Viterbi decoder 51
3.11 The state diagram of the Viterbi decoder 52
3.12 The pipelined path extension unit 57
3.13 The pipelined operation 59
3.14 The combination circuit for soft-decision bit data 62
3.15 Calculation of credit 63
3.16 Branch metric calculation 63
3.17 Butterfly unit configurations 64
3.18 Path merge unit configurations 66
3.19 Content evolution of path metric memory 67
3.20 State diagram of the cyclic decoder 69
3.21 Structure of the cyclic decoder 70

3.22 Implementation of the shift chain unit 72
4.1 Test bench structure 74
4.2 SAIF RTL design flow 79
4.3 Threshold and power consumption 81
5.1 Modified pipeline stage 1 82
5.2 Gray code experiment 84
5.3 Redundant ACS unit 86

List of Tables

2.1 Cyclic coding polynomial in GSM 33
2.2 Convolutional coding polynomial in GSM 34
2.3 Convolutional coding schemes in GSM 34
3.1 The operation mode 39
3.2 Code rate and code duplication 61
3.3 Branch metrics derived from B0 64
4.1 The test cases and results 75
4.2 The unit delay 77
4.3 The circuit area cost 78
4.4 The power statistics 80
5.1 The cost of Gray code 85

Acknowledgements

This thesis is the outcome of my two-year M.Sc. study at the Technical University of Denmark. A lot of people have given me invaluable help during this period of time. First of all, I would like to thank Ph.D. Dan Rebild and NOKIA DANMARK A/S. Without their assistance, I couldn't have been so devoted to the study during the last two years. I would also like to thank Associate Professor Flemming Stassen, who always inspires me to achieve better solutions during my study. Working with him is challenging, but I found it greatly worthwhile. Another person I would like to show my appreciation to is my supervisor Roy Hansen. He has given me considerable amount of brilliant suggestions about my design, many of which are beyond my own knowledge. From him, I have learnt many techniques which are not described in any books I have read. Finally, I would like to thank Flemming Dahl Jensen and Stig Rasmussen of NOKIA DANMARK A/S. I am indebted to their time and help.

1 Introduction

1.1 Background

1.1.1 Mobile phones and 3G

During the last few years, the function of the mobile communication service becomes more and more versatile. Next generation's cellular phones tend to be more involved in sophisticated computer graphics, real-time speech recognition, and even real-time video. In these applications, not only will voice data be transmitted through wireless network but also video data and others. As a result, the cellular phone's operating clock frequency and needs of memory are no longer related to the relatively low speech data rates, but to the demand for supporting all those media related functions. In all aspects, the evolution of the cellular phone gives a low power and portable solution to support these multimedia capabilities challenges.

A major factor in the weight and size of the mobile phone design is the size of the battery, which is directly impacted by the power dissipated by the circuits. Even when the wired power supply is available, the power dissipation of the cellular phone has to be limited. The heat sinks and cooling fans are widely used on the microprocessors for the heat removal. These techniques cannot be applied on the DSP processors embedded in the cellular phones because of their size. To reduce the power and size cost of the cellular phone, careful electric circuit design must be retained.

The upcoming 3rd generation(3G) mobile phones offer enhanced service like General Packet Radio Service(GPRS) and Enhanced Data rates for Global Evolution(EDGE). As the need of data transmission increases greatly, a new set of communication standards are proposed. To follow the new standard closely, many parts of the current mobile phones need to be redesigned. This document mainly discusses a low cost design of an important unit for 3G GSM mobile phone – the channel coding decoder.

1.1.2 Communication and channel coding

The wireless communication facility like cellular phones execute data transmission and receiving very frequently. When data are transmitted through a noisy communication channel with no protection, erroneous results at the receiving end could be expected. Such a impairment could be originated from the noise, fading and jamming. To superimpose an effective protection on the data, several method could be applied on data transmitter and receiver, e.g. by using a higher power transmitter, a larger antenna or a popular technique called channel coding. Channel coding becomes favorable among the others because it costs least power, physical size and is suitable for VLSI implementation.

The most frequently seen channel coding methodology in the digital domain are block coding, convolutional coding and data interleaving. Block coding is also known as CRC check and is widely used in many applications. Convolutional coding is a popular topic for chip design and communication system design, however it is less known. It will be the focus of this document and will be discussed in detail in later chapters. Data interleaving is simply a reordering technique of transmitted data. It can ease the effect of the burst error and take more advantage of the convolutional coding. For GSM mobile phones, all these three techniques' application are involved and clearly described in the standard [2]. Since those coding techniques are applied on every single bit of transmitted and received data, the power consumption to carry out those coding and decoding function needs special consideration.

1.2 Overview

The focus of this document is the design of the convolutional coding decoder and the block coding decoder. The major part of the design is the convolution code decoder. The function of the decoder unit is based on the recently proposed GSM 3G standard [2] and some discussion with the supervising group. Besides the functional requirement [25], the power and area optimization of the decoder is also the major part of the project. The decoder should be considered as the functional unit(FU) of the DSP processor in cellular phones.

There are several convolutional decoding techniques described in previous works [19]. Here, the Viterbi algorithm is chosen as the convolutional coding decoder in this project. Viterbi algorithm is famous for its independence of channel characteristics and its nature of potential parallelism. However, the cost for Viterbi algorithm is much higher than other algorithms in some cases. To reduce the cost of the power, a threshold based algorithm is introduced into the design.

The register transfer level (RTL) description is done in Very high speed integrated circuit Hardware Description Language(VHDL-93). The design is optimized at the architecture and algorithm level. To evaluate the design, at least the gate level synthesis is required. The area of the design is estimated with the *Design Compiler* of the available synthesis tool *SYNOPSYS 2000.05.* The power consumption estimation is done with the *SYNOPSYS 2000.05.* and *ModelSim 5.5E* through their interface *DPFLI*.

The design library *GS50*'s vendor is Texas Instrument(TI). The *GS50* is a 0.09 μm 5-metal-layer CMOS technology. In the design some SRAM and register files cells are used. All the memory modules are offered by TI and compatible with the library *GS50*. This technology library characterizes the design area in the number of minimum sized NAND gates. Normally the layout route tool *AVANT!* can place 150,000 gates in 1 mm^2.

1.3 Document organization

Chapter 1 **Introduction** gives a short overview of the project. It briefly describes the application domain and gives the overview of the project.

Chapter 2 **Technology overview** dissects the design into many topics. It covers the following areas:

- Communication basics and channel coding.
- Viterbi algorithm.
- Advanced topics of Viterbi algorithm in this project.
- GSM 3G standard and impact on the Viterbi algorithm.
- Low power CMOS design methodology.

Chapter 3 **Decoder Design** describes the current design of the decoding unit. Each part of the current design is explained in this chapter. The power and area of each part are analyzed and the optimization of the circuit is pointed out. Even if the design process is iterative, the decoder is still introduced as top-down design.

Chapter 4 **Design evaluation** describes the power and area measurement of the current design. This chapter will also explain how the power efficiency is related to the decoding dynamics.

Chapter 5 **Future work** describes the possible further power optimization.

Chapter 6 **Conclusion** compares the design with T-algorithm based sequential transmission design and concludes the project.

2 Technology overview

2.1 Digital communication methodology

The functional block diagram in figure 2.1 shows a typical digital communication system [19]. The upper blocks are in the data source and the lower blocks are to the data sink side. It can be seen that the lower blocks in the figure 2.1 are mostly the inverse operation of the upper blocks. The function of each block is briefly described below.

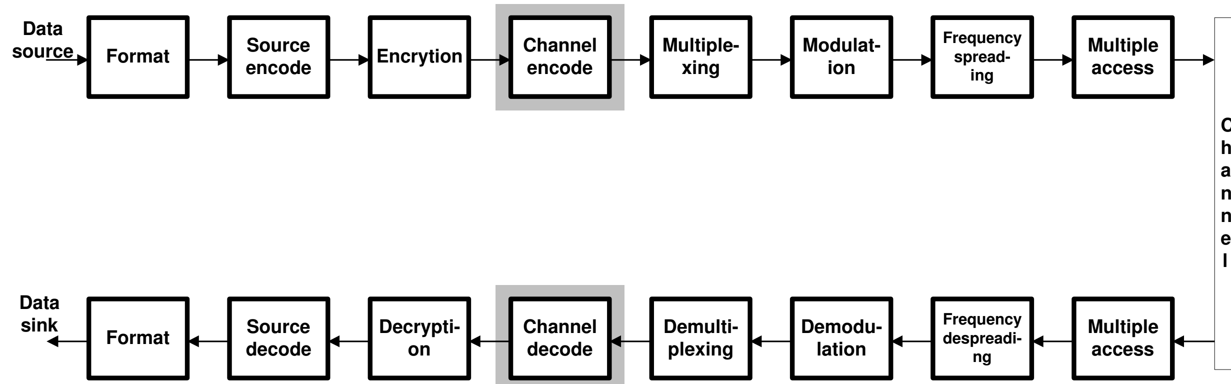


Fig. 2.1. Data communication methodology

2.1.1 Data transmission

Format transfers the source data into digital symbols.

Source encode removes the redundant data and increases the information density by compression.

Encryption prevents the data being understood and modified by the unauthorized group.

Channel encode offers trade off between reliability, in terms of error probability(P_E) and signal-to-noise ratio(SNR), and hardware complexity, in terms of bandwidth and decoding complexity.

Multiplexing modifies the digital data received from several source so that they could be transmitted in one channel.

Modulation converts the digital data to a form which is suitable for transmission channel.

Frequency spreading produces a signal that is less vulnerable to interference and enhances the privacy of the communicator.

Multiple access sends data through a shared channel.

2.1.2 Data receiving

Multiple access detects data on a shared channel and collects relevant data.

Frequency despreading recovers the spread data.

Demodulation converts the channel signal to digital signal.

Demultiplexing identifies which data sink the received data should be dispatched to.

Channel decode checks the correctness of the received data or corrects them to some extend.

Decryption recovers the encrypted data.

Source decode recovers the compressed data.

Format transfers the digital data into useful format for applications.

In this document, the channel decoder is the major topic. To understand the function of a channel decoder, the function of channel encode must also be understood. The next section describes a typical channel coding scenario as an example.

2.2 Channel coding

The channel encode in digital domain normally includes block code, convolutional code and interleaving. The figure 2.2 depicts the normal processing flow of the data. Each of these steps is adaptable and complicated. The rest of this section will explain those steps through relevant examples.

2.2.1 Block code

The overall idea of the block code is to generate some certain redundant code from a data stream and transmit both redundant code and the data stream for error correction. If the redundant code are carefully designed and the channel noise power is acceptable, the receiver of the data can probably detect a corrupted data stream or even correct the error. Block code has many subclasses like rectangular code, Reed Solomon code and cyclic code. Here only a cyclic code example is given, since the cyclic code is the only subclass of block code involved in this design.

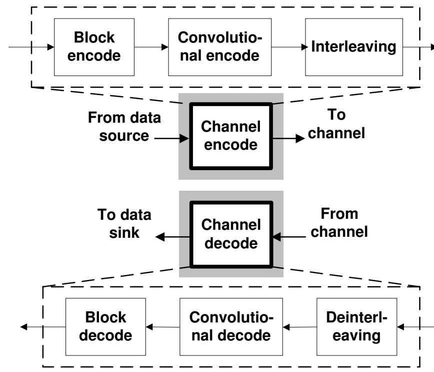


Fig. 2.2. Channel coding block diagram

Encoding circuit

The cyclic code encoder uses a modulo division circuit to generate the redundant code. The input data stream is divided by the encoder and the remainder of the division is used for error correction/detection. Figure 2.3 depicts a typical n-stage division circuit. In the figure all the adders are physically XOR gates. The cells named r_0 to r_{n-1} are implemented with shift registers.

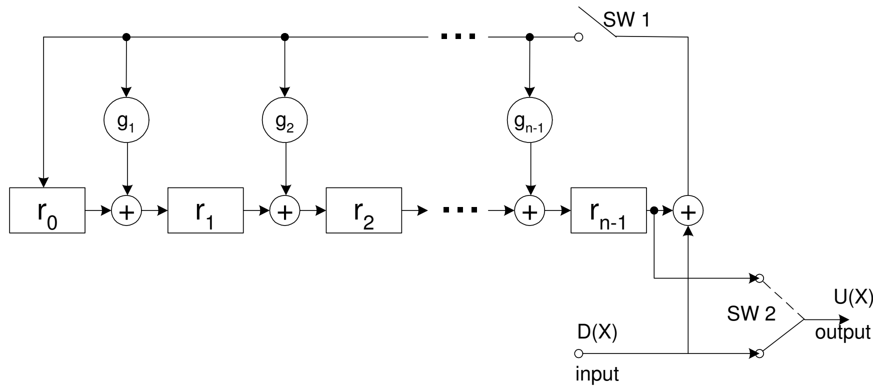


Fig. 2.3. Systematic n-stage cyclic encoder

The feedback loop in the circuit could be described with a polynomial g(X) of the form

$$g(X) = g_0 + g_1 X + g_2 X^2 + \cdots + g_n X^n \tag{2.1}$$

This polynomial is called a generator polynomial. For an n-stage division circuit, the parameter g_0 and g_n must be 1, while g_1 to g_{n-1} could be either 0 or 1. Parameter g_1 to g_{n-1} describes the physical feed back connection in the figure 2.3 for each stage, while X^0 to X^n represent the

delay. For a certain stage, if the associated 'g' parameter is 1, the feedback should be physically connected to the associated XOR gate in front of the register. In case the 'g' parameter is 0, the feedback connection is absent and the XOR gate is unnecessary. The number of shift registers is not related to the generator polynomials. For an n-stage divider, there are always n-1 registers.

This class of block encoder has no constrain on the length of the input sequence. Data stream of any size could be coded with any cyclic encoder, but the error correction capability is limited. To use this circuit to generate the redundant codes, first the contents in all the registers must be reset to 0. The switch 1(SW1) should be closed and the switch 2(SW2) should be connected to lower input node. When those steps are accomplished, the input data are shifted into the divider 1 bit each clock cycle. Since the output is directly connected to the input, the output will now be exactly the input sequence. After all the input bits are shifted into the divider, the remainder of the division should resides in the register chain. The remainder will be used as the redundant data for error detection/correction. To get the remainder out of the decoder, the switch 1 should be open and the switch 2 should be connected to the upper node, where the LSB of the remainder is currently latched. During the next n-1 clock cycles, the sequential output will be the latched content in the shift register chain. After all the remainder bits are shifted out, the encoding procedure finishes. The output from the cyclic encoder, including both the input date stream and the remainder, will then be sent to convolutional coding unit.

To summarize, if an input data stream D of length k is encoded, the output data stream U will be

U(m)=D(m) for m=0, 1, 2, ..., k-1;

U(m)=r(m-k) for m=k, k+1, k+2, ..., k+n-1;

where n is the number of encoder stage, r is the remainder of the division.

Decoding circuit

A typical decoder is shown in the figure 2.4. The decoder at the receiving side is also a modulo division circuit.

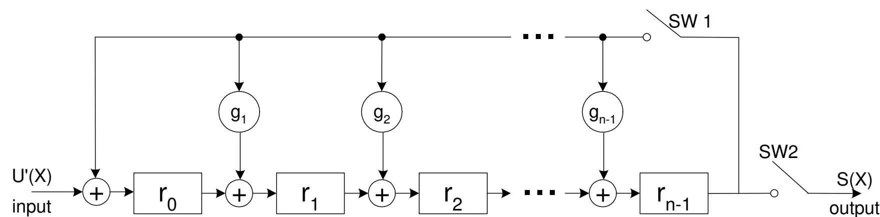


Fig. 2.4. Systematic n-stage cyclic decoder

To decode a data stream encoded with a certain generator polynomial, the decoding division circuit must contain exactly the same feedback polynomial as the encoder. Before decoding the

input data, the switch 1 should be closed, the switch 2 should be open and all the registers should be reset. After that is done, the received data stream, which contains both encoded data and the encoding division's remainder, will be shifted into the decoder 1 bit each clock cycle. After all the input data are shifted into the decoder, the switch 1 will be open and the switch 2 will be closed. The output of the decoder will only be the content in the registers, since the encoded data is just a part of the received data stream and can be easily determined. The resulting remainder of the division is called syndrome. If the received data is correct, the syndrome should be a known constant, which is normally 0. In some cases the syndrome could even be used for error correction by doing syndrome analysis. Since the syndrome analysis is not part of the project, it will not be discussed here.

The theory of block coding has been described in many books [19] [24]. Since the underlying theory requires too many explanation and yet has little to do with the VLSI implementation, it is not explained here.

2.2.2 Convolutional coding

Overview

The convolutional coding gives the protection to every bit of transmitted data. The encoding has few requirements on encoding hardware, but the decoding circuit is very expensive. Furthermore, the bandwidth requirements of the channel and the decoding circuit are sometimes increased several times. The encoding circuit and some overview of the decoding is described here, while the detailed Viterbi decoding algorithm is explained two separate sections.

The name of convolutional coding somehow gives the insight of the coding technique itself. As shown in figure 2.5, each bit of data will be convolved into several consecutive coded outputs, as each encoded output will contain information of several consecutive input bits. In case a certain bit is corrupted during the transmission, the other data could be used to regenerate the corrupted information.

To offer more error correction capability, the number of the output bits of convolutional encoder is normally several times more than that of the input bit. The example shown in figure 2.5 demonstrates an encoder that has one-bit input symbol and two-bit output symbol. In each clock cycle 1 input symbol is shifted into the encoder. At the same time, the corresponding output symbol is collected as a pair of bits. An important definition *code rate (r)* is defined as **the rate between the number of input data in bit and the number of input data in bit**. In this example, the code rate is 1/2. The lower the code rate is, the better the error correction potential an coding scheme will have, and the higher bandwidth is required for the data transmission.

As shown in figure 2.5, each output symbol is convolved from 3 input symbols. To make this relationship possible, there must be two stages of memories in the encoder. Another important definition of the convolutional encoder is the *constraint length (K)*. Constraint length is defined as

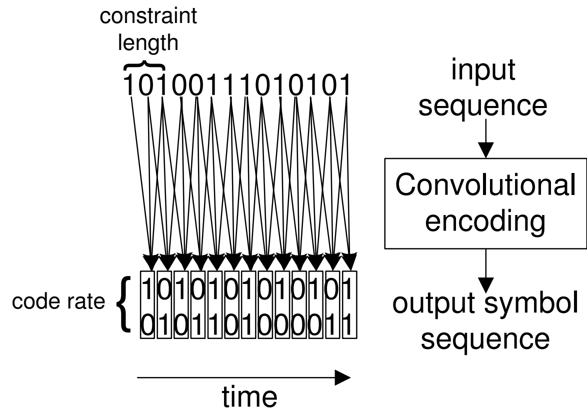


Fig. 2.5. Convolutional encoder input-output correlation

the **number of memory stages+1**. For example, the encoder in figure 2.5 has a constraint length of 3. The constraint length has the most significant influence on the decoder structure. For the Viterbi decoding algorithm, the decoding complexity grows exponentially as constraint increases, but only linearly if code rate increases.

Typical encoder

Figure 2.6 shows the structure of a typical convolutional encoder. This configuration is frequently used in GSM 3G standard.

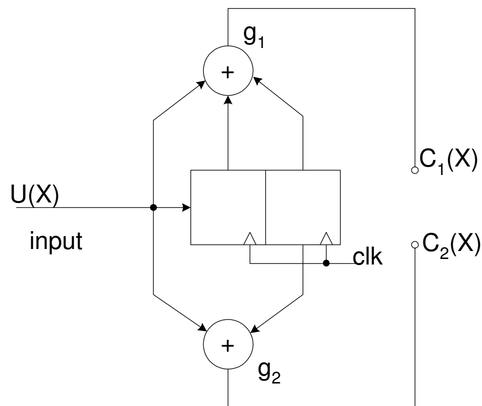


Fig. 2.6. Convolutional encoder (r=1/2 K=3)

This encoder has a code rate of 1/2 and a constraint length of 3. Its physical implementation simply contains a two-stage shift register chain, a two-input XOR gate and a three-input XOR

gate. For this example, the upper and the lower part of the connection could be expressed with two generator polynomials:

$$g_1(X) = 1 + X + X^2 \tag{2.2}$$

$$g_2(X) = 1 + X^2 \tag{2.3}$$

The output C_1 and C_2 could be described as

$$C_1(X) = U * g_1(X) = U(X) \bigoplus U(X-1) \bigoplus U(X-2) \tag{2.4}$$

$$C_2(X) = U * g_2(X) = U(X) \bigoplus U(X-2) \tag{2.5}$$

where U is the input data stream, X is the delay element and the arithmetic units $\bigoplus$ are physically XOR gates. Before the encoding process starts, the encoder's memory element is normally reset. As a result, U(-1) and U(-2) are simply zeros.

RSC encoder

The other encoding system involved in GSM system is the Recursive Systematic Convolutional (RSC) encoder. Figure 2.7 shows an example of an RSC configuration.

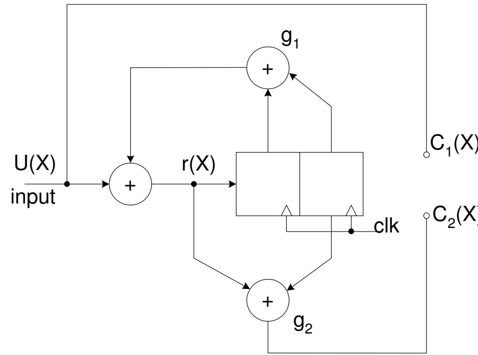


Fig. 2.7. RSC encoder (r=1/2 K=3)

This encoder has the same code rate and constraint length as the previous one, but different generator polynomials. Notice that the g_1 is the feedback polynomial, and the other generator polynomials are expressed in its form. e.g. The top and lower connection are expressed as:

$$g_1(X)/g_1(X) = 1 \tag{2.6}$$

$$g_2(X)/g_1(X) = 1 + X^2/1 + X + X^2 \tag{2.7}$$

The output C_1 and C_2 could be described as

$$r(X) = U(X) \bigoplus r(X-1) \bigoplus r(X-2) \tag{2.8}$$

$$C_1(X) = U(X) \tag{2.9}$$

$$C_2(X) = r(X) \bigoplus r(X-2) \tag{2.10}$$

Decoding technics

There are three well known decoding techniques for convolutional coding: the Viterbi algorithm, the sequential decoding and the feedback decoding. Sequential decoder's complexity is independent of constraint length K(K could be 41), thus can achieve very good error protection. But the main drawback of the sequential decoding is the requirement of a large buffer memory. Also the time needed for the decoding process is random. Feedback algorithm could only be used for hard-decision bit, which is not suitable for the targeting application.

As a maximum likelihood decoding, Viterbi algorithm is the most used algorithm for low constraint length codes. Since the decoding complexity grows exponentially as K increases, the Viterbi algorithm is scarcely used if K is larger than 13. The GSM standard mostly uses low constraint length code like 5 and 7, thus makes the Viterbi algorithm a good choice. Since the main interest of this project is to find a power and area optimal Viterbi algorithm for a GSM cellular phone, the next two section will give a detailed explanation of it.

2.2.3 Interleaving

Interleaving is a technique which prevents burst of errors during the transmission. Figure 2.8 shows a typical interleaving circuit.

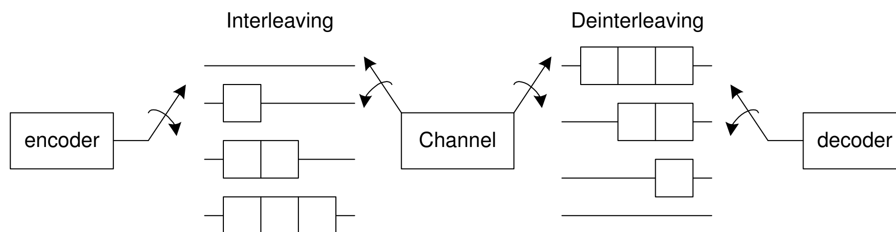


Fig. 2.8. Interleaving circuit

The convolutional coding offers correction probability for a few bits' error, but if the channel is interfered for a short moment so that several consecutive transmitted bits are contaminated, the

convolutional coding will be of little help. A simple way to make better use of the convolutional coding is to reorder the encoder output so that the data are transmitted out of order. At the receiver side, the received bit sequence is rearranged into the correct order before convolutional decode starts. If there are burst errors in the channel, the erroneous bits will be distributed into many places among other correct bits so that the convolutional decoder will have better chances to correct the error. Such a reordering technique is called interleaving. The interleaving is often used in the GSM standard, but the input to the decoder is assumed to be deinterleaved in this project.

2.3 Viterbi algorithm

The Viterbi algorithm is known as the maximum likelihood decoding algorithm. This class of decoder is proven that it can find the most likely coded data. It can implicitly correct errors and offer some information about the decoding quality. This section introduces the basics of the Viterbi decoding with short examples. Due to the complexity of the decoding, other books [19] [24] use a large amount of figures to demonstrate the decoding process. Here the most important concepts of the Viterbi are described, but the explanations are omitted for the sake of brevity.

2.3.1 Encoding state diagram

The encoder in figure 2.6 can be represented as a Mealy machine. The content in the shifter registers could be used to denote the states. The state diagram of the encoder in figure 2.6 is shown in figure 2.9. On every edge there is a set of value assigned to U and C. The value of U corresponds to the current encoder input, while the value of C corresponds to the output of the decoder.

The state transition happens every clock cycle. It is easy to confuse the state S0 to S3 with the output C1 and C2. The current state is only decided by the content in the register chains but the output C is the output of the XOR gates.

2.3.2 The Trellis encoding diagram

An alternative presentation of the state diagram is the trellis diagram. The figure 2.10 is the trellis presentation of the state diagram in figure 2.9.

The trellis diagram emphasizes on the time progress of the state transition. The time is expressed as the horizontal axis on the diagram. The left side states on the diagram are all the possible states at t=n, while the right side states are the states at t=n+1. The 8 edges are all the possible state transitions from time n to time n+1. The values assigned on each edge inherit the same notation convention from the figure 2.9. When the time progresses, the trellis diagram extends to the right.

The trellis diagram can be used to express the encoding process better than the state diagram. Supposing the encoder is reset to state 0 at time t=0, if there is a stream of data 101100 coming from the cyclic encoder, the encoding activity could be indicated by the bold path in figure 2.11. The output and the state transition of the encoder could be read out from the noted path.

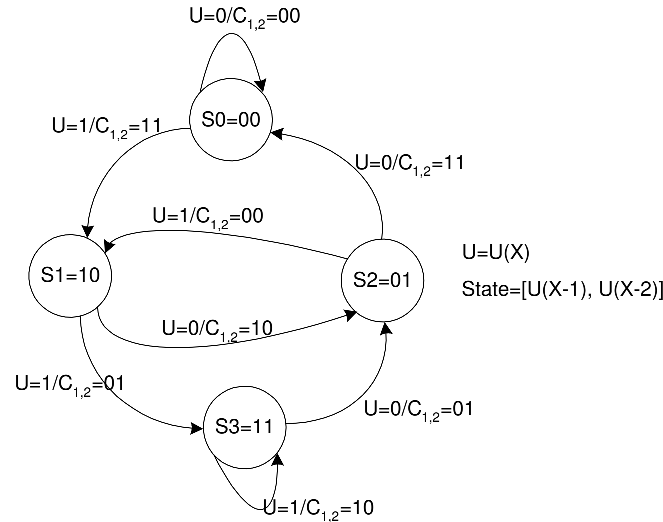


Fig. 2.9. Encoder state machine

2.3.3 Convolutional decoder

The concept of trellis diagram is of little use for the convolutional encoder, but it is the basic of Viterbi decoding. The decoding process could be described as looking for the best path on the trellis diagram that matches the decoder input.

Consider a decoder that receives the transmitted signal 11 01 01 00 10 11 going from t=0 to t=6. Assume the encoding and decoding trellises were both reset to state S0 before t=0. The decoding path could be shown as in figure 2.12. Notice that the data on each edge is shown as $C_1'C_2'$/U', where C' is the decoder input and the U' is the decoder output. The decoded data could be read from the path as 110100. Note that the channel is assumed as ideal. If the noise is present, the decoder input could be any data. If, for instance, the decoder receives a 01 at t=0 and the known initial state is s0, how to find a decoding path then? A way to measure the likelihood of the decoder input and the edge must be found.

Branch metric

The Hamming distance is the simplest decoding-correctness measurement to demonstrate. It is also closely related to the actual measurement in this project. The metric used in this project will be introduced in a later section with the soft-decision bit.

Compare the bits in the same positions in two different binary numbers. The number of positions that are different is the Hamming distance. For example, the distance between 00 and 11 is 2, while the distance between 00110 and 10100 is also 2.

In trellis diagram, each state has two edges that connect to the states in the previous decoding stages and two edges extend to the states of the next stage. Those edges are called *branches*. As

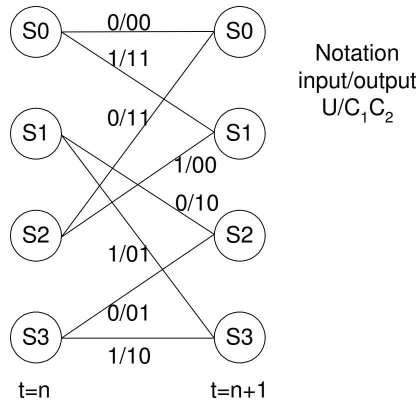


Fig. 2.10. Trellis diagram

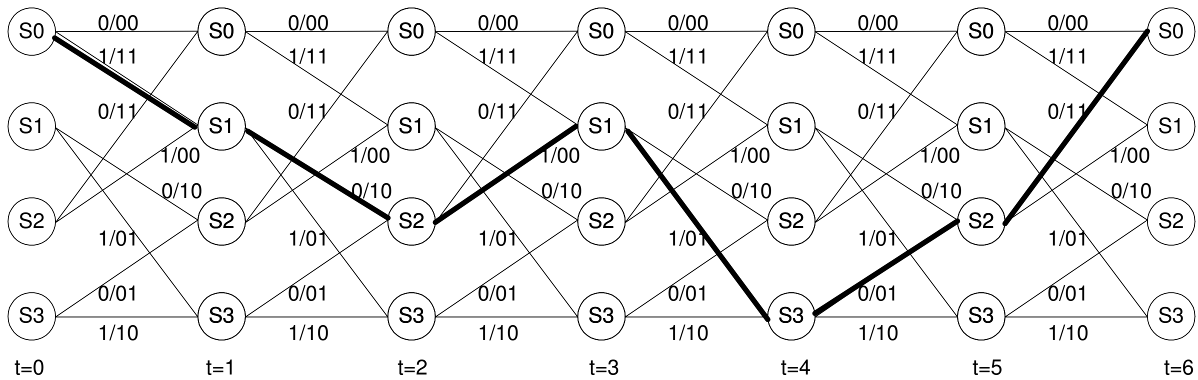


Fig. 2.11. Trellis encoding diagram

time progresses, the branches will extend into *paths*, as shown in figure 2.11 and 2.12.

As mentioned before, each branch is assigned a set of value $C_1'C_2'/U'$, where C' is the speculated decoder input for this specific edge. The definition of the *branch metric* is **the Hamming distance between a received data and the corresponding input of the branch**. A Hamming distance of 0 depicts a perfect match between the received input and the corresponding input for a certain branch.

Suppose that the received data is 00 at t=0 and the decoder is reset to S0. Those two branches extended from the current state S0 correspond to input 00 and 11. The branch metrics of those two edges are 0 and 2. Apparently the branch with smaller metric is more likely the correct branch for decoding, but a stage-to-stage based decision on branch selection is improper. To make fully use of the convolution characteristic, the metric should be measured on path.

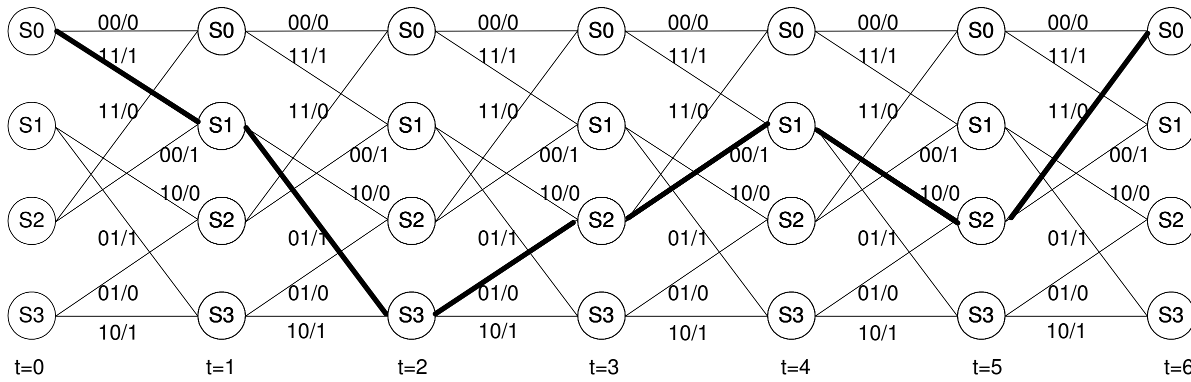


Fig. 2.12. Trellis decoding diagram

Path metric

Suppose an input stream 11 11 01 is received and the initial state is S0, the decoder extends paths as shown in figure 2.13. The first decoding stage has only one winner S0-S1, since the other branch have metric which is larger than 0. The second stage has more than one winners, since both branches have a metric of 1. At the third stage, the two previous winners extends themselves into four branches, each of which has its own branch metric. Starting from S0 at t=0, the branches fork into four paths at t=3. To evaluate which path is most likely to be the winner, the branches metrics along each path are accumulated to form a *path metric*. In this case, the path S0-S1-S3-S2 has the lowest path metric, thus is the potential winner.

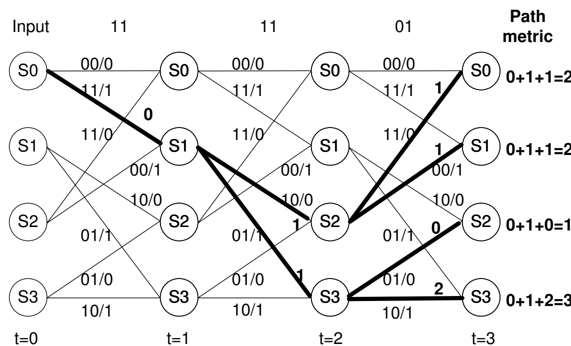


Fig. 2.13. Path metric

Path merge and decoding

Every time a new input is received, the new branches' metric will be added to the existing paths and form new paths. At a first thought, every time a new input is received, the number of paths

will be doubled. Actually, it is not necessary to keep track of all the paths, since the paths will cross and overlap each other. Figure 2.14 shows the situation where paths meet at t=4.

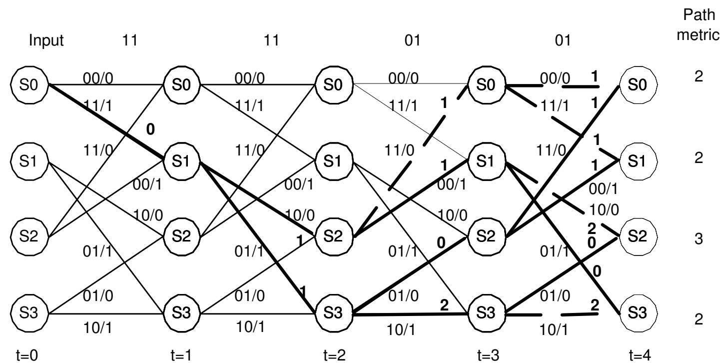


Fig. 2.14. Path merge

Between the stages when t=3 and t=4, the four pathes at t=3 fork into 8 new paths and meet at t=4 in pairs. At t=4, each state is a joint of two paths with their own metrics. If the two joint paths have different metrics, the path with larger metric will not be kept track of after this point. If those two joint paths have the same path metrics, the rejected path could be randomly chosen. On figure 2.14, the rejected paths are marked with dashed lines. After such a rejection, there will only be four paths left for further evaluation. Such a mechanism is called path merge.

Path merge guarantees that the number of valid paths at any moment will not be more than the number of states in the encoder Mealy machine. The process that the paths are doubled and reduced by half is sometimes called path extension. As the constraint length increases, the number of valid path increases exponentially, which also makes the number of decoding computation grow exponentially.

Note that the edge S2-S0 at the third decoding stage is also marked as rejected, since all the forked branches from it are marked as invalid in the fourth stage. At the end of the third stage this branch looks like a possible candidate, but the later decoding stage shows that all the paths extended from this branch are merged by the other paths. As path merge continues, more and more earlier branches will be rejected by using this principle. In case there is only one branch left in a certain decoding stage n, the most likely data sequence before stage n could be decoded. For instance, if the only surviving branch in stage 4 in figure 2.14 is S1-S3, then the first 3 decoded bit could be 1011.

It is normal that a unique surviving branch for each stage cannot be found even if all the encoded input are processed. If so, the path with the minimum metric at the last decoding stage is considered as the most likely decoding path. As a matter of fact, the path merge based decoding is not very practical, since the time it takes for a stage to be resolved is not constant. A popular way to decode the input is the stage based trace back, which will be introduced shortly.

ACS unit

The trellis decoder's branches in figure 2.10 form two butterfly shapes. The butterfly-shape is often called double add-compare-select (ACS) unit. Figure 2.15 shows the butterfly unit in general form. In the figure, the left side state J and $J+2^{K-2}$ extend to the state $2J$ and $2J+1$, where J is a less-than-2^{K-2} non-negative integer and the K is the encoder constraint length.

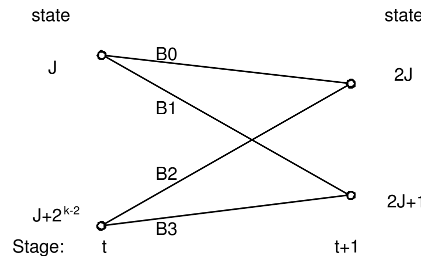


Fig. 2.15. Butterfly unit of non-RSC code

There are two sets of data named corresponding decoding output and path merge decision bit. The data called corresponding decoder output has the same value as the values of U in figure 2.10 and U' in figure 2.12. The normal convolutional encoder has a fixed relationship between the edges and the value of U, but the RSC encoder has not. The second set of data, the path merge decision bit, points out from which previous state the branch is merged. Given the coding generator polynomials, a current state and the path merge decision bit on this state, the previous state on this path and the corresponding decoder output could be justified. This set of data is used for trace back decoding, which will be explained shortly.

Each cone at the right side of the butterfly is called an ACS unit. The ACS unit performs the core algorithm of the path merge. It adds the two new branches' metric to the metric of the path ended at J and $J+2^{K-2}$ in the previous state, compares the new path metrics and selects the path with the smaller metric as the new path ended at the new state. While doing so, the path merge decision bit is saved in a memory for decoding uses.

Trace back

The path merge will sometimes make a certain branch to be the sole survivor of a decoding stage, thus indicates the proper path for decoding. However, there is no guarantee when the sole survivor can be found. A popular method of decoding only does a fixed number of path extension and decodes some data by using the temporary best path. If the number of path extension is big enough, the decoding quality will have little degradation. For a typical sequential decoder, it might do 35 stages of path extension before decoding. Such a decoder will first do 34 stages of path extension with no decoding. After that, every time the decoder receives a new input, it will extend the paths and trace back 35 stages along the temporary best path to produce one bit of output. The number of stages the trace back travels through is called the *trace-back depth*. If the

data is transmitted as blocks, the decoder could first extend paths with the whole data block and do a full path trace back. in this case, the trace-back depth is equal to the number of the input symbol.

As described before, the decision bits of the path merge should be stored in the memory. If a trace back needs to be done, the decisions made when this path is extended and the state at which the path is ended must be known. Figure 2.16 shows an example of trace back decoding. In the figure, the minimum-metric path is ended at the state S0. The decision bit vector is 101100 while the MSB depicts the last decision made. The decoded output could be derived as 110100 while the MSB corresponds to the first received input.

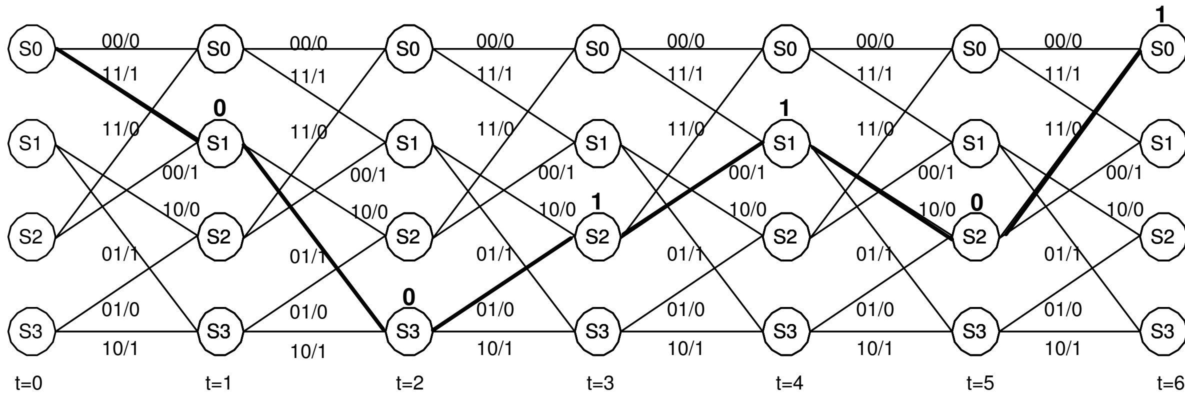


Fig. 2.16. Trace back

There are two ways to store the decision bits. If all the decision bits along a path is stored in one memory address, the decoding only needs to locally regenerate the states along the trace back path and decoded data. The problem of this way of storing data is the width of data bus of the memory. If the path is too long, e.g. 500 stages, the decision bits along a path can not be stored in one word.

If the decision bits are stored as one word for each stage, the decoder will have to read out all the decision bits for all the previous stages and properly index them. Such a design will slow down the trace back, but have little implementation difficulty.

Viterbi algorithm summary

The Viterbi algorithm is a trellis based decoding algorithm. The path extension and the trace back are the major operations of this algorithm. There are other techniques that can be used instead of those two, but in this project they are the chosen implementation due to their low cost.

As the trellis structure shows, the path extension could be done concurrently with many ACS units. Such a special characteristic implies that the timing/cost trade-off is possible. This trade off is one of the most important topics in this project.

Due to the complicity of the GSM standard and the application domain, the actual implementation of the Viterbi unit is in many aspects more complicated than the Viterbi algorithm described

here. The detailed differences and strategies to handle these difference will be discussed in the next few sections.

2.4 Advanced Viterbi algorithm

2.4.1 Metrics with soft-decision bit

Soft-decision bits

Before the signals are transmitted through the channel, they are often converted to the analog signals by modulator. At the signal receiver side, the transmitted analog signals need to be converted back into digital signal before being decoded. The simplest way do to such an analog-to-digital conversion(ADC) is to compare the received analog signal with a fixed analog threshold voltage V_T. The received analog signal is treated as digital 1 if it is higher than the fixed voltage V_T, and vice versa. Such a conversion makes sure that 1 transmitted digital bit will be converted back to 1 bit at the receiver side. It could be said that the receiver is making a hard decision on the signal from channel and the resulting 1 bit data is called hard-decision bit.

The previous examples in figure 2.10 use hard-decision bits as the input since the decoder input has the same format as the output of the convolutional encoder, e.g. they are both two-bit symbols. The introduced branch metric that uses Hamming distance is one of the most frequently used metric for hard-decision bits. This kind of metric is easy to use but has relatively low decoding quality. To achieve a better decoding quality, this project uses soft-decision bits instead of hard-decision bits.

The soft-decision bits are generated from a multilevel ADC. Depending on the precision of ADC, the received analog signal could be represented with a 4 or 5-bit integer as shown in figure 2.17. The hard-decision could be considered as the output of a two-level ADC. The soft-decision could be understood as a measure of confidence along with the code symbol. Those integers close to 0 could be considered as low confidence values, and the values close to 7 or -7 could be considered as high confidence values. With the confidence information, a better branch matric could be obtained.

new branch metric

Suppose that the positive integer $i \in [+1,+7]$ formed by the 4-bit soft-decision bit denotes binary 1, the negative integer $i \in [-7,-1]$ denotes binary 0 and integer 0 denotes neither 0 nor 1. The two-bit input symbol in the previous examples should now be denoted by two signed integers. If a branch's corresponding input C' matches the sign of the input integer, the absolute value of the integer is counted as credits to the branch metric, otherwise the negated absolute value of the integer is counted as credits. Sometimes the negative credit is called penalty.

An example is given here to demonstrate the use of the credit. Suppose a branch's $C_1'C_2'$ is 01 and the soft-decision input is -4,-3, the branch will totally get 1 point of credit for the following reason. Because the C' of value 01 expects the first soft decision number to be negative and the second to

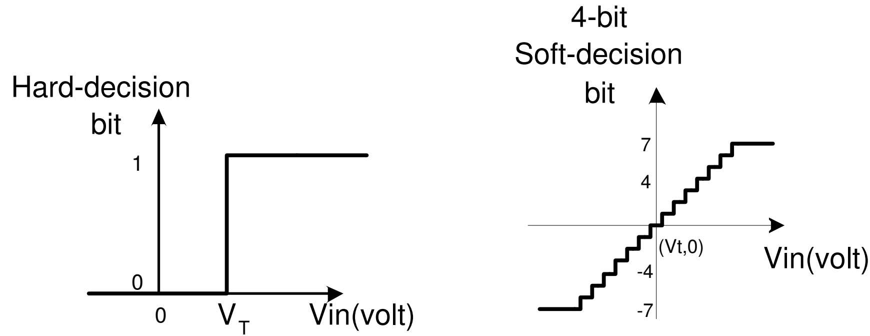


Fig. 2.17. Input-output transfer curves for hard-decision and 4-bit soft-bit decision ADC

be positive, the -4's absolute value will give 4 points of credit because of the sign matching and the -3's negated absolute value will give -3 points as credits (or take 3 points as penalty) because of the sign mismatching. From all the credits, totally 1 point credit is accumulated. The total credit for a branch is actually the new branch metric.

The path metric is accumulated as before: every time a path is extended, the branch metric is added to the previous path metric. But now the path with largest metric is the best decoding path. The cost of such a metric computation is more complicated ADC, more storage for the path metric and more arithmetic units.

2.4.2 Threshold based Viterbi algorithm

The Threshold based Viterbi algorithm(T-algorithm) is a low power Viterbi algorithm. The idea of the T-algorithm is to reduce the number of computation without losing too much decoding quality.

The most of the computation of Viterbi algorithm is spent on the path extension. To reduce the power consumption of the Viterbi algorithm, the path extension needs special care. The author of [18] discovered the best decoding path normally exists in the best 25-50% of all the paths. In another word, the best decoding path exists among those 25-50% of the paths with the highest path metrics. The author first developed an algorithm called M-algorithm[4]. The M-algorithm tried to extend only a fixed number(M) of path to reduce the power consumption of the decoder. Later the author developed another algorithm called T-algorithm, which uses a relative path metric(T) as a threshold to prune low metric paths. To do that, after the path extension for a certain stage is done, the best path of the current stage is found and its metric PM_{max} is recorded in the memory. Next time the path extension starts, the Viterbi unit first checks if there are any paths that have a metric lower than PM_{max}-T. If such paths are found, they will not be extended by ACS units. The paths that passed the threshold are called survivors. By carefully designing the threshold T, the T-algorithm can have better performance than M-algorithm. The statistic data from [18] shows the

larger the constraint length is, the less the percentage of the survivor paths is needed. Article [9] suggests a variable value of T for data received from different channels. If the channel's error rate is high, a larger value of T can give a decoding quality that is close to the ideal Viterbi algorithm. If the channel's error rate is low, a smaller value of threshold can be used to save more battery power.

Till now, all the other known implementation of the T-algorithms are used for sequential transmission. For a GSM mobile phones, the data is transmitted in blocks, which makes the decoding technique different. As the other paper [9] describes, proper use of T-algorithm can sometimes saves 70-95% of total power in the sequential decoding. The problem here is whether or not it can also be so power efficient in block based data decoding? This is the major problem this design needs to answer.

2.5 GSM standard

2.5.1 Generator polynomials

Block code

The GSM standard [2] uses both block code and convolutional code to protect data. Data transmitted through channels use many different kinds of block code and convolutional code generator polynomials. There are totally 10 block code used in the GSM standard. Nine block codes are cyclic code that is similar to the coding describer in section 2.2.1. The special block code is a Reed Solomon code. Since the Reed Solomon code is not well accepted by most of the mobile phone vendor, it will not be considered here. The generator polynomials of those nine cyclic code are listed in table 1.

ID	Generation polynomial	Remainder
1	$D^8+D^4+D^3+D^2+1$	0
2	D^3+D+1	$1+D+D^2$
3	$D^{14}+D^{13}+D^5+D^3+D^2+1$	$1+D+D^2+\ldots+D^{13}$
4	$D^6+D^5+D^3+D^2+D^1+1$	$1+D+D^2+\ldots+D^5$
5	$(D^{23}+1)(D^{17}+D^3+1)$	$1+D+D^2+\ldots+D^{39}$
6	$D^{10}+D^8+D^6+D^5+D^4+D^2+1$	$1+D+D^2+\ldots+D^9$
7	$D^{16}+D^{12}+D^5+1$	$1+D+D^2+\ldots+D^{15}$
8	$D^8+D^6+D^3+1$	$1+D+D^2+\ldots+D^7$
9	$D^{12}+D^{11}+D^{10}+D^8+D^5+D^4+1$	$1+D+D^2+\ldots+D^{11}$

Tab. 2.1. Cyclic coding polynomial in GSM

In table 2.1, the remainders are the expected syndrome left in the decoder shift registers if there is no error present in the decoded data. Notice that the most of the remainders are expected to

be all-1s, which means the redundant code generated from the encoder are bit-inverted before transmission. The D in the table is the convention of the delay in GSM standard, which has the same meaning as the X in equation 2.1.

Convolutional code

The GSM standard used 13 combinations of generator polynomials listed in table 2 to do convolutional encoding. The combinations are listed in table 3. The encoder includes both normal convolutional encoder and RSC encoder. Again, the D in table 2 is the same as the X in equation 2.2 and 2.3.

Name	Generation polynomial
G0	$1+D^3+D^4$
G1	$1+D+D^3+D^4$
G2	$1+D^2+D^4$
G3	$1+D+D^2+D^3+D^4$
G4	$1+D^2+D^3+D^5+D^6$
G5	$1+D+D^4+D^6$
G6	$1+D+D^2+D^3+D^4+D^6$
G7	$1+D+D^2+D^3+D^6$

Tab. 2.2. Convolutional coding polynomial in GSM

ID	Coding Polynomials	Constraint length	RSC
1	G0, G1	5	No
2	G1, G2, G3	5	No
3	G1, G2, G3, G1, G2, G3	5	No
4	G4, G5, G6	7	No
5	G4, G6	7	No
6	G4, G7, G5	7	No
7	G1/G3, G2/G3, 1, 1	5	Yes
8	1, G1/G0	5	Yes
9	G1/G3, G2/G3, 1	5	Yes
10	G1/G3, G1/G3, G2/G3, 1, 1	5	Yes
11	G4/G6, G5/G6, 1, 1	7	Yes
12	G4/G6, G4/G6, G5/G6, 1, 1	7	Yes
13	1, G5/G4, G6/G4	7	Yes

Tab. 2.3. Convolutional coding schemes in GSM

The GSM standard encodes data with three methods:

1. The data is only encoded with block code;
2. The data is only encoded with convolutional code;
3. The data is encoded with convolutional code after it is encoded with a block code;

By combining varies coding generators and coding methods, GSM has more than 40 different ways to encode the data for different channels. To deal with such a variable configuration, the decoder need to have a similar structure. GSM standard also describes the way that the transmitted data is interleaved. In this project, the input to the decoder is assumed to be deinterleaved, thus requires no further study.

2.5.2 Punctured data and trace back

GSM standard also describes how the data are grouped as block before transmission. Here the MCS-9 coding scheme is used as an example.

The data burst input of MCS-9 convolutional encoder has 612 bits. The convolutional encoder uses the 6th set of generator polynomials in table 2.3, thus has a code rate of 3. Since the input of the encoder has 612 bits, the output of the encoder should be 3 bits * 612 output symbol=1836 bits. MCS-9 code scheme then only transmit 1/3 of the encoded data by puncturing 2 of every 3 output bits. Such a puncture scheme reduces the number of the transmitted data by a factor of 3 so that only 612 out of 1836 encoded bits are transmitted. By doing the puncturing, the actual code rate of transmission r_t is increased to 1. At the receiver side, before the data are decoded, the punctured data are replace by soft-decision bit 0. It can be seen that the decoding quality of the punctured data block will be reduced. To improve the decoding quality, the trace-back depth introduced in section 2.3.3 must be increased. In [6], the authors found that for a rate equal to 1/2 encoder, if r_t is also 1/2, the trace back depth only need to be 40. But if r_t is increased to 15/16, even trace back 240 stages will lose performance. Since the GSM standard uses heavily punctured data, the trace-back depth must be equal to the length of the data block. In other words, trace back must be done only once per data block. This requires a large memory to store trace-back decision bit. The MCS-9 coding scheme example actually creates the biggest data block, thus is the most costly code. Other coding schemes can also use the same decoding method for MCS-9 with no extra cost.

2.6 Low power CMOS design

2.6.1 Power dissipation in CMOS circuit

The power dissipation in a CMOS circuit could be described with the following equations [14] [7]:

$$P_{total} = P_{static} + P_{dynamic} + P_{leakage} \tag{2.11}$$

$$P_{dynamic} = P_{switching} + P_{short-circuit} \tag{2.12}$$

$$P_{switching} = \alpha C V_{dd}^2 f \tag{2.13}$$

$$P_{short-circuit} = I_{sc} V_{dd} \approx \frac{\alpha C V_{dd}^2 f}{10} \tag{2.14}$$

$$P_{static} = I_{static} V_{dd} \tag{2.15}$$

$$P_{leakage} = I_{leakage} V_{dd} \tag{2.16}$$

There are 4 main components of power dissipation: switching power, short circuit power, leakage power and static power. Switching power is normally the main source of power dissipation. It is the result of the transient switching activity. The switching power is decided by the total load capacitance C of the circuit, the magnitude of the voltage supply V_{dd}, the operating frequency f of the circuit and the average percentage of nodes α being charged during switching activity. The short circuit power is the power dissipated in the switching activity when both NMOS and PMOS network are conducting. The amount of power dissipated in the short circuit period is normally 10% of the switching power, so the low switching power design can also reduces the short circuit power. The leakage power is caused by leakage current and substrate injection current of the transistors. Previous research in [14] found that the leakage power is normally several orders of magnitude below dynamic power, thus is not a very interesting field for power optimization. The last component of power dissipation is static power. It is the power consumed by pseudo-NMOS digital circuit or bias circuit of the analog circuit.

2.6.2 Low power design methodology

Low power design overview

To achieve a low power design, totally 5 design abstraction layers need to be considered. The figure 2 shows those 5 layers and related low power design methods for each layer. The low power design should start from the top layers system design and traverse to lower levels at later design phases.

Scope of this project

In this project, not all of the 5 layers of design are considered. The circuit/logic layer of the design is related to gate level and layout level design. In this project the gate level design is mostly dealt with by computer aid tool SYNOPSYS. The layout level design is not studied because of the lack of the tool. The given technology library GS50 could not be modified and studied in detail, so the technology layer is also not considered. The most important layers in this design are the algorithm layer and architecture layer, since they can be optimized at RTL level.

The most interesting factors for low power design are V_{dd}, α, C, f and I_{static}. In this project I_{static} is not considered since the design library GS50 does not have pseudo-NMOS cells. The operation frequency f=100MHz is the fixed requirements of the project, thus should not be changed. V_{dd} of

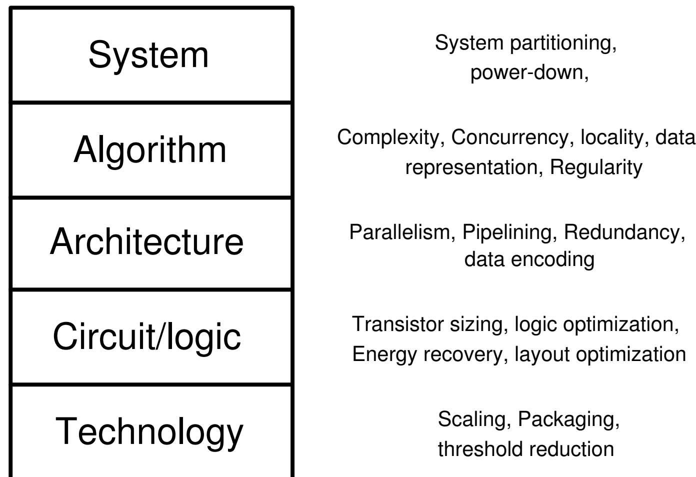


Fig. 2.18. Low power design methodology[14] [7]

the design is fixed at 1.3 volt since it is the operating voltage for GS50 cells. The only two terms left to consider are the activity factor α and the load capacitance C. The load capacitance of the design is partly optimized by SYNOPSYS and partly decided by GS50 library. Sometimes the timing of the design will also have influence on the load capacitance. It is not explicitly measured and optimized in this project, but a design with reasonable timing should reduce the capacitance to some extend.

The major target for power optimization is to reduce the activity factor α. In fact, the reason why the T-algorithm can have such a high power efficiency is the low α. Many of the design methods on the architecture layer and the algorithm layer are considered and experimented in the project, and some of them do have effect on power reduction.

3 Decoder design

3.1 Requirements

This section describes the functional requirements of the design. Most of the contents in this section could be found in [25].

3.1.1 Coding scheme and structure

Each of the coding schemes described in [2] follows one of the following four scenarios:

1. Block coding only.
2. Convolutional coding only.
3. Block coding → convolutional coding.
4. Block coding → reordering/merge → convolutional coding.

To decode a reordered or merged data block, data have to be decoded by the Viterbi unit and sent back to the DSP unit. After the DSP unit virtually reordered the data, they will be sent to decoder unit for the cyclic decoder. The decoding unit need to have a configurable structure as shown in figure 3.1 to support the next three operation modes:

1. Cyclic decode only.
2. Viterbi decode only.
3. Viterbi decode → Cyclic decode.

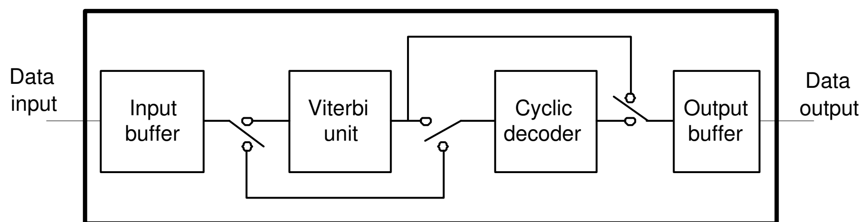


Fig. 3.1. The structure of decoder

A control signal from DSP unit will switch the decoder to different decode mode that is suitable for the current data block. This signal is called operation mode control. The operation mode control signal is a 2-bit signal that specifies which coding unit should be used for the current data

Operation mode control signal	Operation
0	Idle
1	Viterbi decoder only
2	Cyclic decoder only
3	Both Viterbi and cyclic decode

Tab. 3.1. The operation mode

block. Table 3.1 describes the relationship between the mode control signal and the operation mode of the decoder.

The DMA control is used in this design. The whole encoded data block is transmitted to the decoding unit before the decoding is performed. The encoded data block is stored in an input buffer residing in the decoder as shown is figure 3.1. After the whole data block is placed in the input buffer, the DSP unit will notify the decoder and initiate the decoding process. The decoder only needs to fetch the data from the input buffer, which could be realized by RAM structure. The decoded data is placed in an output buffer. After the whole data block is decoded, The decoder will send a signal to notify the DSP unit that the decoded data is ready in the output buffer.

For the Viterbi decoder, data that are encoded in one encoding stage will be received in parallel. This requires a data bus of certain width. If the data should only go through cyclic decoder, N coded bit will be sent to decoder in parallel, where N is the width of the data bus. The cyclic decoder is responsible for reading the N parallel input bits one by one in a pre-defined order.

Both Viterbi and cyclic decoder produce single-bit decoded data. Those sequenced single-bit output will be sent to DSP unit N bits at a time, which is similar to the way the N bit parallel input data for cyclic decoder is received. The order of the decoded output will be the same as that of the encoded input. The syndrome generated from the cyclic decoder will be concatenated to the decoded data.

3.1.2 Viterbi decoder

Introduction: The Viterbi unit is an adaptive decoding unit that can decode the data as encoded in [2]. Table 2.2 and 2.3 give a summary of the convolutional encoding polynomial in [2]. The encoded data is received as blocks with various lengths. The size of the data block will not have influence on the decoding process, but the size should be known to decoder. The input data format, the generator polynomials and the path prune threshold should be parameterized for each data block. However, those parameters should hold constant during the decoding process of a data block.

Data Input: The data input to the decoder is 4 or 5 bits soft-decision [19] coded data from input buffer. The data shall be depunctured and ordered correctly before they are stored in the input buffer. Depending on the encoder coding rate, each single bit data will be encoded into 2 to 6 coded data, in extreme case resulting in 30 soft-decision bits. These soft-decision coded data gen-

erated from one encoding stage should be stored in one address space as part of a word as shown in figure 3.2. The DSP unit is responsible to format the soft-decision data as described before the data is sent to the decoder.

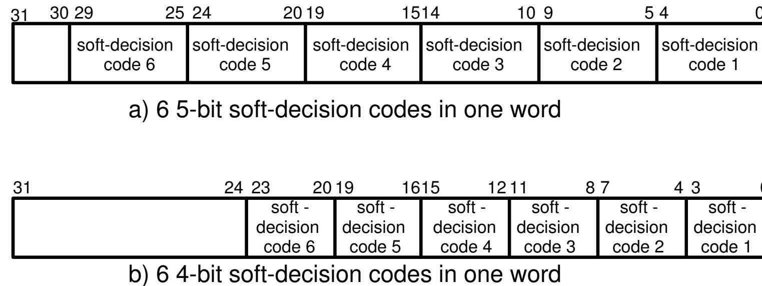


Fig. 3.2. Soft-decision coded data for each decoding stage contained in one word

Control input:

1. Decoding polynomial control: As shown in table 2.3, there are 13 different combinations of coding polynomials involved in GSM 3G coding standard. A 4-bit control signal will be needed to specify the decoding metric computation. The **ID** column in table 2.3 is used as the integer presentation of this control signal.

2. Soft bit control: As mentioned before, the soft-decision data could be 4 or 5 bits. A single-bit control signal will be needed to let the decoding unit understand the input data format and compute the branch metric properly. If the signal is binary 1, the encoded data is understood as 5-bit soft decision bits, otherwise, the encoded data is 4-bit soft decision bits.

3. Threshold control: The threshold for survivor pruning should be given to the decoder. Based on the estimation from [18] and the characteristic of 5-bit soft-decision bit, for coding constraint length of 7, a maximum threshold of 480 should lead to the coding quality that is close to the normal Viterbi algorithm. Thus, the threshold control signal is a 9-bit unsigned integer. The value of the threshold ranges from 0 to 511.

4. Block size: The block size is needed to control the number of path extension steps and trace back steps. The block size is a less-than-613 positive integer, thus it should be a 10-bit data.

Processing: The Viterbi decoding unit reads the data input received from the DSP unit and produces the decoded bits. The decoding process starts when a new data block is completely stored in the input buffer. A control signal must be received from DSP unit to indicate the completion of the data block transmission. The decoding consists of two steps: path extension and trace back. Path extension is the first step. It operates on all the coded data input as following:

1. Reads input data that corresponds to one decoding stage from input buffer.
2. Extends and merges the survivor paths.
3. Stores the trace back data into memory.
4. Prunes the paths that cannot pass the threshold.

After all the coded input data are processed by path extension, the decoder will start to trace back information. During the trace back, the decoded output along the best path from the last path metric update will be sent to data output buffer. After a block of data is decoded, the metric storage will be reset.

Output: The output from trace back will be a sequence of single bit. These data will be merged into words as shown in figure 4. Those words will be sent to the output buffer for further use.

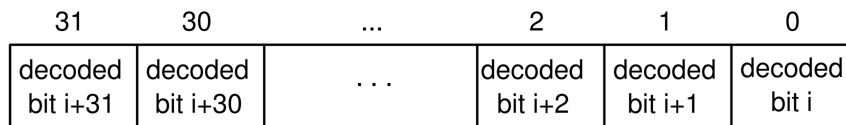


Fig. 3.3. 32 decoded bits merged in one word

3.1.3 Cyclic decoder

Introduction: the cyclic decoder is also an adaptive decoding unit. The decoder is a non-fixed-length shift register decoder with reconfigurable feedback loop. Table 2.1 gives a summary of the cyclic encoding polynomial in [2]. The encoded data is received as data blocks with various lengths. The length of data block will not have influence on the decoding process, but the size of the data block must be known to the decoder. The decoding configuration varies for each data block. The decoding shift register length and feedback polynomial could be parameterized for each data block. Those parameters should hold constant during the decoding process of each data block.

Data Input: The input to the cyclic decoder is correctly ordered single bits stream. The input could either be from the output of Viterbi unit or the DSP unit. Under both circumstances the input data should be received in words. Each input word is merged from 32 single bit coded input as shown in figure 3.4.

If the input is received from the DSP unit, the input data will be read from the input buffer. The DSP unit is responsible to merge the data into proper format before the data is sent to the input buffer. The data should be correctly ordered and the redundant 0-tail bits for RSC termination

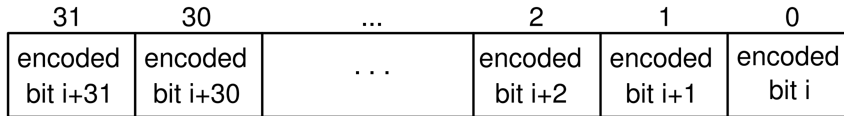


Fig. 3.4. 32 encoded bits merged in one word

should be removed by the DSP unit. If the input is from the Viterbi decoder output, the input data will be read from the output buffer. The data format should be guaranteed to be correct by Viterbi unit.

Control input: The control signal specifies which decoding polynomial should be used for each encoded data block. Table 2.1 lists all the GSM related cyclic coding polynomial. Since there are totally 9 coding polynomials, the control signal needs to be 4 bits. The **ID** column in table 2.1 is used as the integer presentation of this control signal. The block size is also needed for the counting.

Processing: The decoder reads the encoded data in series. The received data will be sent to the decoding shift register chain one bit at a time and clocked through the decoder. The decoded data will be sent to the output buffer. Once a block of data is decoded, the syndrome left in the shift register chain will also be sent to output buffer in series for further error analysis.

Output: Output from this unit will be single bit wide. Both the decoded data and syndrome will be stored in the output buffer. The syndrome will be concatenated to the decoded data while reserving the order that they are shifted out. The output data is merged into words as the output of the Viterbi unit

3.1.4 Data storage and control signal

As mentioned before, the input buffer can be implemented with a RAM. Even if the input soft-decision code will never be more than 30 bits in parallel, the data port is chosen to be 32 bit. The extra data storage will not be wasted, since the input buffer can be reused to store trace back decision bits. The size of the input buffer is decided by the length of the data block. The largest data block in [2] is known to be 612 bits. It could be seen that a memory of size 612-word by 32-bit will be sufficient, but it may not be the best choice.

To configure the decoding process, the decoding unit also need the following data: 2-bit operation mode control signal; 14-bit decoding parameters for Viterbi unit; 4-bit decoding parameters for cyclic decoder and 10 bits to specify the data block size. These data could be merged into a word as shown in figure 3.5 and sent to the input buffer as a data in order to reduce the interconnect between the decoder and DSP unit at the cost of 0.15% extra data storage.

Fig. 3.5. 30 bits of control data merged into one word

The input buffer can be implemented with a 613-word by 32-bit RAM. The encoded data must be stored between address 0 and 611. The data must be stored in successive addresses with the first data placed at address 0. Address 612 is reserved for the control data only.

The output buffer has the similar structure as the input buffer. As mentioned before, the single bit output of the decoder will be merged into words. The maximum number of bits need to be stored is 612 decoded data plus 12 bits syndrome. Since the input buffer is not used after the decoding is finished, the input buffer could be reused as output buffer.

The storage for the trace back information could also be implemented as RAM. Since the largest data block described in [2] is coded with a convolutional encoded that has the constraint length of 7, the size for such a RAM will be 612-word by 64-bit. Such a memory size is enough for any other coding schemes described in GSM standard. It could be possible to use input buffer as part of the trace back memory with some dedicated control.

3.1.5 Timing requirements

The system clock frequency is 100MHz. The data block of any size must be decoded in 300us. For the largest data block containing 619 encoded data, average number of clock cycle to decode one bit will be 48.5.

3.1.6 Interface design

The decoder unit only interfaces to the DSP unit. As shown in figure 3.6, the interface signal includes the following.

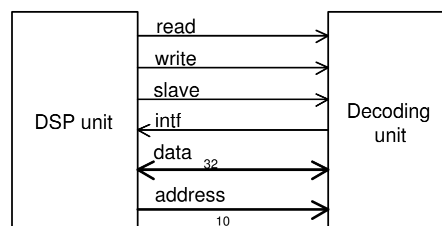


Fig. 3.6. The interface between decoder and DSP unit

1. The read data signal (read) asserts when DSP unit reads out the decoded data from decoder. The content in the output buffer will be sent to the data bus.
2. The write data signal (write) asserts when DSP unit sends the encoded data into the decoder. The data will be stored into the input buffer.
3. The slave signal (slave) asserts when DSP unit reads data from or writes data to the decoder. The decoder will do nothing when this signal is assert.
4. The finishing interrupt signal (intf) indicates the decoding of a data block is finished and calls the DSP unit to read the data.
5. Data bus (data) transmits the data and control signal between the DSP unit and decoder.
6. Address bus (address) specifies the address of the data that the DSP unit is operating on.

Figure 3.7 shows how the signals are changed during a decoding scenario.

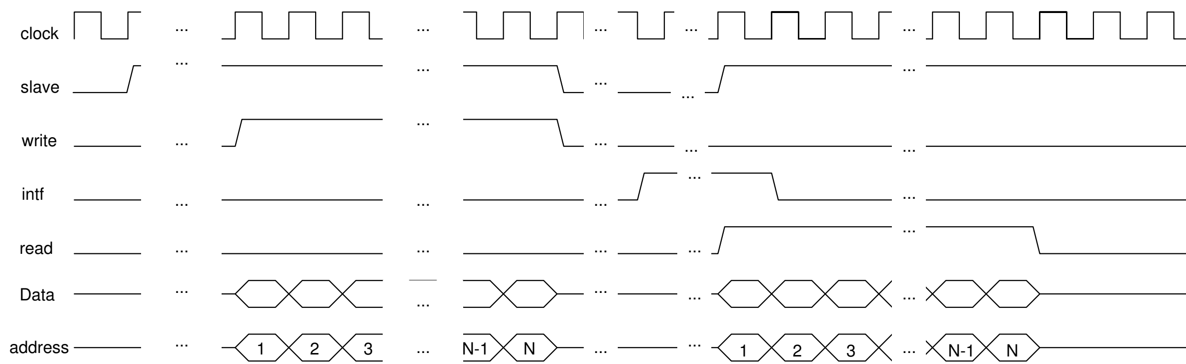


Fig. 3.7. The signal waveform of control signal

3.2 Decoder structure design

3.2.1 Structure design

The decoder should be able to do Viterbi decoding and CRC check for a data block. The encoded data are placed in an input buffer before a decoding process starts. After the encoded data are read by any of the decoder once, the encoded data are never reused, and the input buffer is never read again. The figure 3.1 suggests the use of a separate input buffer and an output buffer, but a more efficient design will be to use the same physical memory to store both the encoded data and decoded data. Such a structure also leads to a simplified control: under all operation modes, the two decoding units always get data from the same physical memory and store the decoded data

to the same memory. The structure of the decoder could then be summarized as in figure 3.8

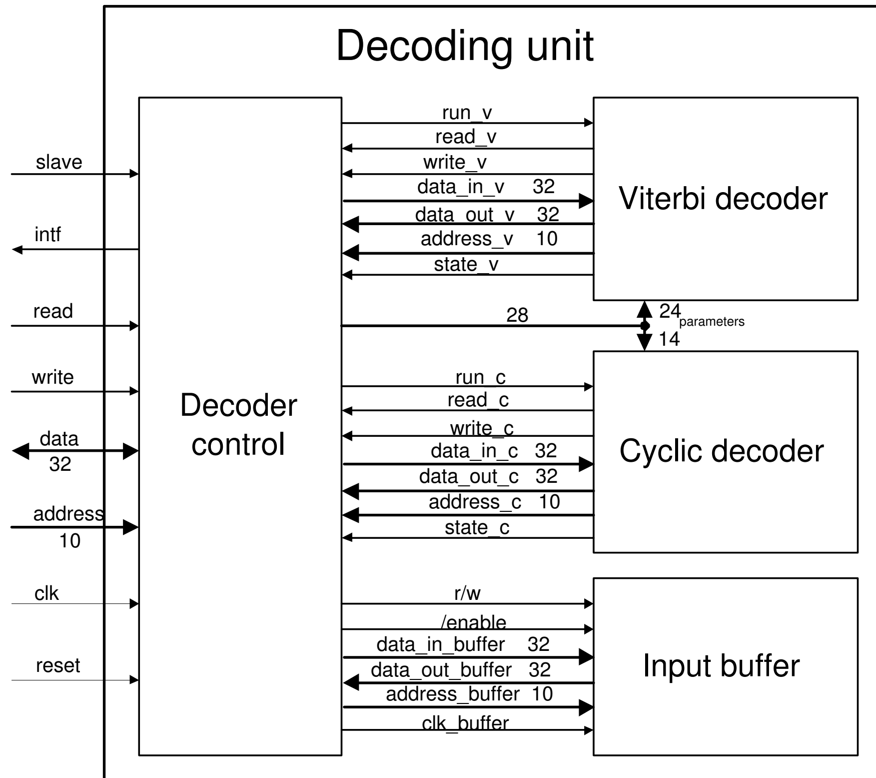


Fig. 3.8. The top level structure of the decoder

As shown in figure 3.8, there are four units in the decoder: a control unit, a Viterbi decoder, a cyclic decoder and a buffer memory. The input buffer memory should be able to contain the largest data block and the coding parameters, which require the size of 30*613. The actual size of the buffer is chosen to be 32*624, since it is the best available memory. The memory module is a synchronized SRAM that supports bitwise write. The data sheet of the memory is attached in the appendix B(GL00624032040). The DSP unit places the encoded data into the buffer and retrieves the decoded data with the remainder from buffer memory. The Viterbi decoder reads the encoded data from the buffer memory and puts the decoded single-bit string into the buffer. The cyclic decoder reads the output from Viterbi unit, which is placed in the buffer memory, and stores remainder generated from CRC check in the buffer memory. For any decoding scenario, if one of the three parts is using memory, the other two parts should be idle. The mutually exclusive access to memory could always be guaranteed by the decoding process. The control strategy of the decoder is simple: the control unit acts as a switch that connects the buffer memory to the Viterbi decoder, the cyclic decoder or the DSP unit. The design problem of the controller is to find out where the buffer memory should be connected to and at which moment.

3.2.2 State transition

The decoder has three operation modes: Viterbi decoding only, block decoding only and block decoding after Viterbi decoding. After the DSP unit places the data block in the input memory, the *slave* signal will be set to low to start the decoding. The parameter that specifies the operation mode is stored in the input buffer. To decode a data block, the decoder must first read from address 612 of the input buffer to find out the operation mode. After the operation mode is known, the decoder will perform the correct decoding process. When the decoding is finished, the *intf* signal will be set to high by the decoder. The signal *intf* will be held to high until *slave* signal becomes high again to acknowledge it. The DSP unit should hold *slave* signal high during the period it reads out the decoded output and places a new data block into the input buffer. Once the *slave* signal becomes low again, the decoder will assume a new round of decoding should be started. The state diagram of the control unit is shown in figure 3.9.

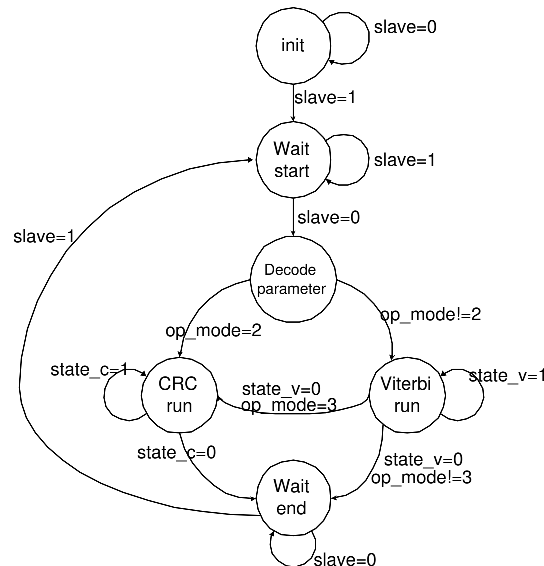


Fig. 3.9. The state diagram of the decoder

1. **Init**: The decoder goes into this state once it receives a *reset* signal. This state is expected to be reached only when the system is powered up. In this state, the decoder does nothing until it receives a *slave* signal from DSP unit. After that, the decoder will expect DSP unit to send data into the buffer memory. When the system is in this state, it prepares for the first write operation from DSP unit after the power up. If the first write operation occurs when the *slave* signal asserts, the data could still be stored correctly in the memory.

2. **Wait start**: The decoder works in the slave mode once it is in this state. The DSP unit will have full access to the buffer memory at this moment. The DSP unit will read out the data decoded from the last data block and place a new data block into the buffer memory. As long as the *slave* signal asserts, the decoder will be in this state.

3. **Decode parameter**: After the DSP unit finishes with the encoded data transmission, the decoder will read out the decoding parameter from the buffer memory. As described before, all the parameters needed in the decoding process are merged into one word and stored at address 612 of the buffer memory. In this state, the decoder reads out the parameters and latches them with flip-flops. Since the memory is clocked, the address bus of the buffer memory should be set to 612 before the decoder leaves the state **wait start**. Depending on the operation mode, the decoder might go into one of the two different states: **Viterbi run** or **CRC run**.

4. **Viterbi run**: If the operation mode requires Viterbi decoding, the decoder will reach this state. In this state, the Viterbi unit will have full access to the input buffer memory. The Viterbi decoder fetches the encoded data from the input memory and stores the decoded data in the buffer. When the decoder goes into this state, the controller will set signal *run_v* to high to wake up the Viterbi decoder. After Viterbi decoder finishes the decoding, the Viterbi unit will set signal *state_v* to low to notify the controller. When the Viterbi decoding is finished, the decoder might go to one of the two states: **CRC run** or **wait end**.

5. **CRC run**: If the operation mode requires CRC check, the decoder will reach this state. In this state, the cyclic decoder will have full access to the input buffer memory. The cyclic decoder fetches all the data from the input buffer. After the data block is sent through the block code decoder, the decoder stores the remainder into the buffer at the address next to the address where the last bit of decoded data is stored. When the decoder goes into this state, the controller will set signal *run_c* to high to wake up the cyclic decoder. After cyclic decoder finishes the decoding, the cyclic decoder will set signal *state_c* to low to notify the controller. When the remainder is found, the decoder goes to state **wait end**.

6. **Wait end**: The decoder sends a finishing interrupt signal to the DSP unit and waits for the DSP unit's acknowledgement. If the DSP unit acknowledges the interrupt by setting the *slave* signal to high, the decoder goes to the state **wait start**. When the system is in this state, it prepares for a read operation from DSP unit. If the first read operation occurs when the *slave* signal assert, the data could still be correctly fetched from the memory.

3.2.3 Function description

Decoder control

The decoder controller passes the access to the buffer memory among the DSP unit, the Viterbi decoder and the cyclic decoder as a single token. The signal *run_v, state_v, run_c, state_c, intf*

and *slave* make such a mechanism well organized. The interface between controller, Viterbi controller and the DSP unit is similar. Besides the above mentioned six synchronization signals, the other signals are the data buses, address buses and read/write signals. The data bus between the controller and DSP unit is bi-directional, thus it has to be implemented with tri-state buffer.

It could be seen that the clock signal of the input buffer *clk_buffer* is controlled by the control unit. When none of the decoder nor the DSP unit need to access the memory, the control unit will gate the memory's clock signal to save power. This also explains why the read and write signals are separate on the other units.

The decoded parameters are sent to the decoder units except for the operation mode control signal. The names and meaning of those signals are:

block_size: block size in figure 3.5.

poly_c: cyclic decoding polynomial in figure 3.5.

threshold: threshold control in figure 3.5.

soft_bit_no: soft bit control in figure 3.5.

poly_v: Viterbi decoding polynomial in figure 3.5.

Input buffer

The input buffer is a 10-bit 624-word static RAM. The RAM has separate input and output data buses. Data output can be latched, but either read or write operation can refresh the output. The memory is synchronized and has a cycle time of 1.62 ns. It supports bitwise write and many test functions. Since none of those functions is used in this design, they will not be described.

3.3 Viterbi unit design

The Viterbi unit is the major part of this design. As described in section 2.3 and 2.4, the decoding process is much more complicated than the cyclic decoding. From the memory requirement point of view, the Viterbi unit requires a memory unit to store the trace back decision information and a temporary storage to store the path metrics updated after every path extension. From the logic point of view, the Viterbi unit requires the butterfly units to do the path extension, a comparator to find out the best path, a circuit to prune the path that can not pass the threshold and special indexing circuits to perform dedicated memory access. The structure of the circuit is very flexible. Depending on the timing requirements, a fast K=7 decoder could be more than 10 times faster than slower designs.

For the cellular phone, the area of the circuit on chip is more important than power consumption. The first goal in this design is to find a decoder structure with lowest area cost without violating the timing requirements. After the structure is settled, the power consumption of each unit is explored and optimized. During the power optimization, the power/area trade-off is allowed as long as the area cost is acceptable.

3.3.1 Decoding scenario

The decoder starts when it receives a *run_v* signal from the top level controller. Due to the heavy data puncture, the trace-back depth is the same as the size of the data block, thus concurrent path extension and trace back is not necessary. Based on that, the decoding process could be divided into two phases: path extension (PE) phase and trace back (TB) phase.

Path extension

The path extension phase is initiated right after the decoding process starts. During the PE phase, the encoded data stored in the input buffer will be fetched. Every time one stage of encoded data is read, the decoder will perform the next few operations.

Soft-decision bit decoding: As shown in figure 3.2, the encoded data stored in the buffer could be presented with 4 or 5-bit soft-decision bits, thus needs decoding. The decoding parameters for the current data block are latched by the top level controller in its **Decode parameter** state. One of the parameters specifies the number of soft-decision bit for the current data block. With that parameter and one stage of encoded data, the soft-decision bit could be dissected into 2 to 6 signed integers.

Path prune: Before the paths are extended, they should be checked for qualification by being compared with an absolute threshold. The absolute threshold is found by subtracting the relative threshold from the metric of the best path that is found during the last round of path extension. The relative threshold is a positive integer latched by the top level controller. If a path cannot pass the threshold, later operation will not be performed on this path.

ACS: Paths will be processed in pairs in this stage. As shown in figure 2.15, path J and $J+2^{K-2}$ will be extended into four paths and merged into two new paths $2J$ and $2J+1$. During this step, every merge operation will produce 1 bit of decision bit for the future trace back. The trace back decision is stored in a memory called trace back memory. Since there is no data dependency between pairs of data, all the paths can be extended in parallel.

Best path searching: Since the current best path's metric is needed for the next round of path pruning, the best path need to be found and its metric should be saved in registers. In case the constraint length K is 7, totally 64 paths need to be compared. How to find the best path efficiently is an important issue.

Store the decision bits: In case K is 7, 64 decision bits need to be stored in a memory for each path extension stage.

After all the above 5 operations are finished for the current stage, a new round of path extension could be started.

trace back

Trace back starts when all the encoded data are processed in the path extension phase. Starting from the best path of the last path extension, the trace back process performs the following operations:

Trace back data fetch: The trace back decision bit is stored in the trace back memory. Every time the decoder travels backward in the decoding trellis for one stage, a stage of trace back memory need to be read out.

Find indexed decision bits: Depending on the value of K, 64 or 16 decision bits in an array will be read out of the memory. The index that points out the correct decision bit on the best path among those 64/16 bits is calculated at the previous trace back stage. The first trace back stage is exceptional: the index of the best path metric is used instead. The index actually also indicates which state in this coding stage is the part of the best path.

Find correct edge: With the current state specified by the index and the decision bits correctly indexed, the decoding edge on the trellis could be found. Once the edge is found, the single bit output could be found.

Find next index: The index used in the next round is calculated here. It is basically the reversed operation of the encoding.

After the above operations are done, the decoder traces back one stage backward in the trellis and decodes the previous input.

3.3.2 Decoder structure

The structure of the decoder is shown in figure 3.10. In this part of the design there are totally 5 units. The function of each part is briefly introduced below. The more detailed explanation will be given in later sections.

Viterbi control unit is a Mealy machine that takes control of the decoder. The state diagram of the Mealy machine is discussed in the next subsection. This control unit is the only entity that has access to the trace back memory (TB memory). It is also the only unit that has communication with the top level controller, thus can indirectly access the input buffer. Like the top level controller, this decoder also gates the clock of the trace back memory in order to save power.

Soft bit decoder decodes the encoded data fetched from the input buffer into 2-6 integers. It is a simple combinational circuit implemented with some MUX circuits. The decoded integer is always 5-bit signed integer. The input *soft bit coded* is the encoded data from the input buffer. The input *soft bit No* is the decoding parameter received from the top level controller.

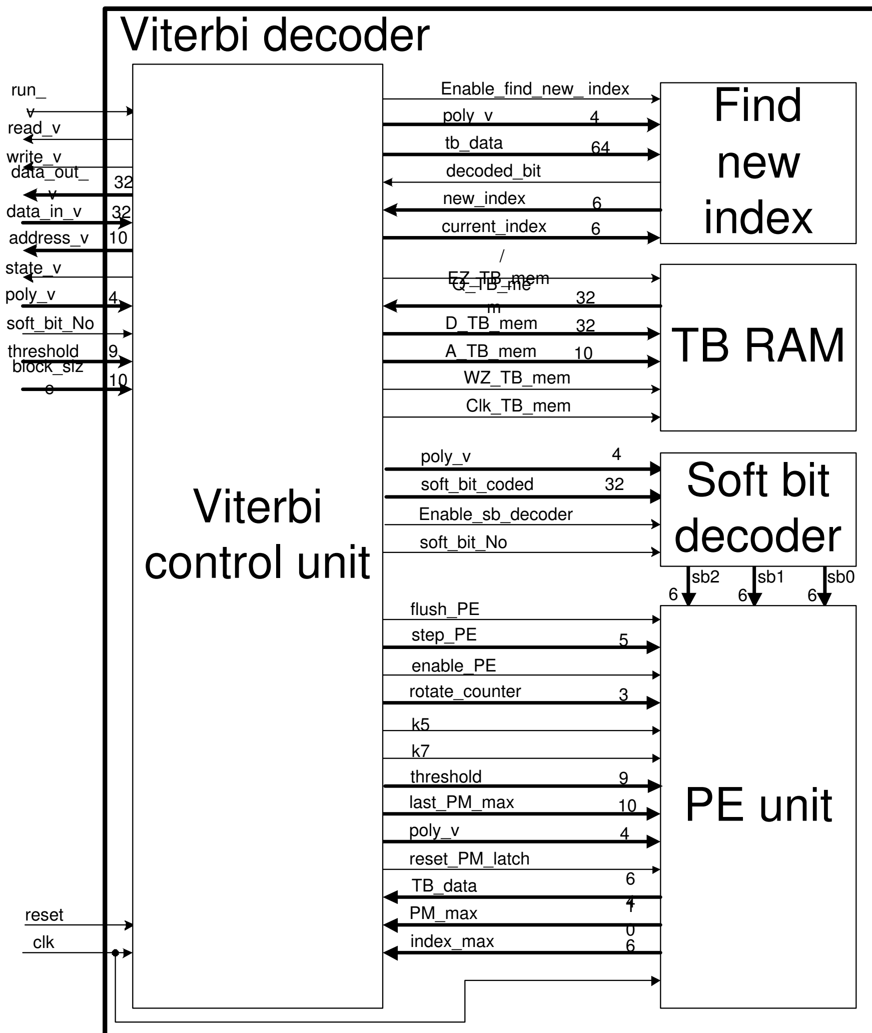


Fig. 3.10. The structure of the Viterbi decoder

The input signal *enable sb decoder* is an enable signal. If this unit is disabled, the output of this unit will be constant, thus reduces the unnecessary activity on the other circuits. The decoding algorithm is shortly explained in the algorithm subsection. The connection in the figure shows that the output is not 6 5-bit integer but 3 6-bit data. Also, this unit has an input *poly_v* which is the coding polynomial specifier. The reason why these signals are present will be explained in the algorithm subsection.

PE unit performs the actual path extension. Due to the complexity of this unit, it will be explained in another subsection. Path extension is the most expensive operation to perform. It is the main target to achieve the low power and low area design.

TB RAM has the same structure as the input buffer. As the name suggests, it is used to store the trace back decision bits. Since the size of the memory is 32*624, it is not big enough to store all the trace back data. This memory is actually only used to store the merge decision of path 0-31. The other half of the decision bits are stored in the input buffer. Since the encoded data is only used by the *soft bit decoder*, there is no reason that the encoded data cannot be overwritten by the trace back data. Notice that the input buffer is only used to store the trace back data when the constraint length K is 7. If K is 5, even the trace back memory is more than enough.

Find new index unit performs most of the trace back operation. Given a stage of trace back data(*tb data*), the index calculated from the last trace back stage(*current index*) and the encoding polynomial(*poly_v*), this unit can find the index for the next decoding stage(*new index*) and the decoded bit(*decoded bit*) of this stage. This unit is also a combinational circuit.

3.3.3 State diagram and control

To carry out the decoding scenario, the Viterbi control unit needs to realize the state diagram as shown in figure 3.11. The operations the decoding performs in each state are explained below.

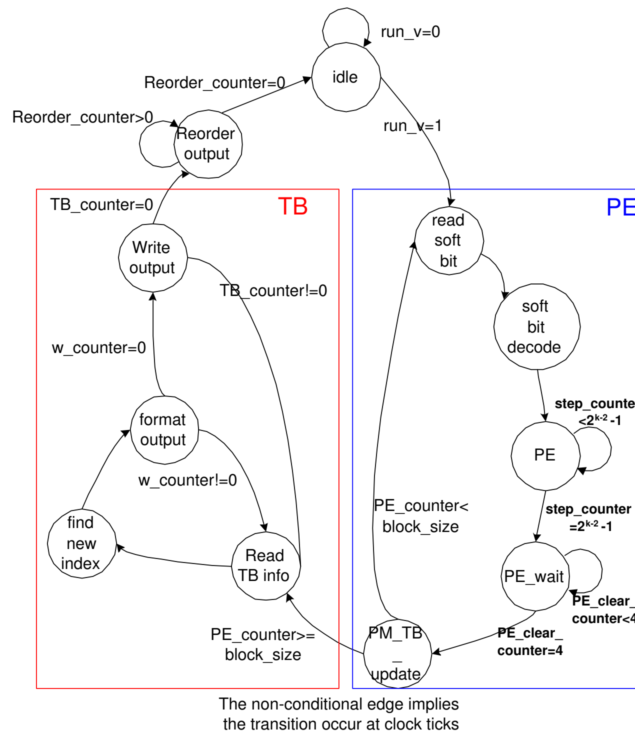


Fig. 3.11. The state diagram of the Viterbi decoder

1. **Idle:** The Viterbi unit goes into this state when it is reset or a data block is decoded. The Viterbi unit waits in this state until the overall controller sets the signal **run_v** to high. In this state, all the subunits in the Viterbi unit are disabled and all the counters are reset to 0. Once the signal *run_v* goes to high, the decoder goes into the path extension phase.

2. **Read soft bit:** The Viterbi unit starts each round of path extension by reading one word of the soft-decision data from the input buffer. Since the memory is clocked, the decoded cannot get the data immediately. In this state, the Viterbi unit specifies the address where the data are stored and leaves the current state at the next clock tick. The address is derived from the counter *PE counter* in the figure. The soft-decision data should be received from the top level controller at the next clock cycle.

3. **Soft bit decode:** In this state, the encoded data should be received from the input buffer. The encoded data will be sent to the **soft bit decoder** for further use.

4. **PE:** Depending on the constraint length K, the decoder will stay in this state for 2^{K-2} clock cycles. The decoder will do one stage of path extension for all the 2^{K-1} survivor paths. While the decoder stays in this state, the counter *step counter* will be increased by 1 in each clock cycle, starting from 0. The value of the counter is sent through the signal *step_PE* to the PE unit for control purpose. The signal *enable_PE* only asserts in this state.

5. **PE wait:** The PE unit is actually a pipelined unit. In this state, the decoder waits until all the signal in the pipeline stabilize. For the controller, the path extension is as simple as waiting.

6. **PM TB update:** As one stage of path extension is finished, the decision bits should be stored in the trace back memory and the input buffer. These operations are done in this state. The decision bits of path 0-31 are stored in the trace back memory, while the decision bits of path 32-63 are stored in the input buffer. Besides the trace back information, the current best path's metric should be stored to form the threshold of the next stage of path extension. If the PE phase is not over, the decoder goes back to state **read soft bit** and increases *PE counter* by 1. If all the input data are processed, the decoder need to go into trace back phase. Before the decoder goes into trace back phase, the value of *PE counter* should be copied to the *TB counter*, the best path's index should be stored in the flip-flop and the value of *W counter* should be initialized as "*PE counter* mod 32".

7. **Read TB info:** The first thing to do during a trace back stage is to read out the trace back decision data stored in the trace back memory and the input buffer. Similarly, the address of the memory cell(s) is derived from the value of the *TB counter*. The trace back data should be received from the memory at the next clock cycle. Since the index of the decision bit is found by the previous trace back or the last stage of path extension, the value of index could be used to save some power. In more details, if the index is less than 32, the decision bit must be stored in the trace back memory, thus the input buffer need not to be accessed and vice versa.

8. **Find new index:** The trace back data and the current index of the decision bit will be sent to the **find new index** unit. The **find new index** unit should produce the index for the next trace back stage and one bit of decoded data.

9. **Format output:** The figure 3.3 suggests that the decoded data should be grouped into 32-bit words before it is placed in the memory. There are two ways to do so. Firstly, the buffer memory supports bitwise write. Every time a bit is decoded, it could be stored in a word without overwriting the other bits. This can cost some power dissipation on the buffer memory. Besides the memory access, an index circuit is needed to perform the bitwise write masking. Secondly, a 32-bit buffer could be used. Every time a bit is decoded, it will be placed in the buffer. When the buffer is full, the 32-bit data in the buffer could be stored in the memory together. Two possible implementations of the 32 buffer are either a 32-bit shift register or 32 indexed registers. Experiments show that the 32-bit registers cost approximately 160 gates and the index circuit costs 70 gates. The shift register need to exercise the whole register chain at the most of the time, but the indexed circuit only exercises the index circuit and 1 register. From the efficiency point of view, the 32-bit shift register costs less area than the indexed registers, but uses more power. In this design, the indexed 32 registers is used since 70 gates takes up a small area. In this state, the decoder puts the decoded bit produced from the **find new index** unit into the buffer. The free space of the 32-bit buffer is recorded by the *W counter*. If the buffer is full, the decoder goes into **write output** state, the *W counter* is set to 31, otherwise, it goes into **read TB info** state and reduces *W counter* by 1. In either case, the *TB counter* will be reduced by 1.

10. **Write output:** If the 32-bit buffer is full, the decoded data should be stored into the input buffer memory. In case the trace back is over, the decoder goes into the **Reorder output** state, otherwise, the decoder goes to **Read TB info** state.

11. **Reorder output:** In case the decoder finishes the trace back from stage 63 to 31, a word of decoded data should be stored into the input buffer at address 1. But it will cause errors if the address 1 of the input buffer contains the trace back data for stage 2, which will be used later. A way to resolve this problem is to store the decoded data at address 32 of the trace back memory during the **write output** state. After the trace back is over, the decoded data is distributed in the trace back memory. In this state, the decoded data are copied from the trace back memory to the proper places in the input memory. Apparently, the initial value of the *reorder counter* depends on the size of the data block.

3.3.4 Path extension unit

Timing analysis

The area optimization of a circuit mainly depends on the time slacks. To know how many parallelism is necessary, the design timing must be explored. It is known that the given time to decode a data block is 300μs, or 30000 clock cycles. The biggest data block described in [2]

contains 612 data. In average, 49 clock cycles are given for each data. For an encoded data, each of **read soft bit**, **soft bit decode**, **PM TB update**, **read TB info**, **find new index** and **format output** operation will use 1 clock cycle. The operation **write output** or **reorder output** only uses 1/32 of a clock for each data. The number of clock cycles it takes for the cyclic decoding is equal to the size of the data block if the control overhead is ignored. On average, it costs 1 clock cycle to perform the cyclic decoding. Taking all these into account, the time left for one stage of path extension is 49-7.1=41.9.

The path extension is a complicated process. Without special optimization, a path extension would normally take 3-4 clock cycles. As the decoding trellis shows, there is no data dependency between paths in an extension stage. In such a situation, the most straight forward implementation is to use 4 butterfly units in parallel to process 8 paths concurrently. For K=7 case, only 4 clock cycle* 64 path/8 path=32 clock cycles is needed to finish a path extension stage. Even if it seems to be a possible implementation, it can hardly make good use of the threshold. To fetch 8 path metrics from a data storage is also expensive and complicated.

Another possible implementation is the pipelined single butterfly unit. Undoubtedly, it is the smallest possible structure for the given requirements. The timing requirements could be fulfilled as long as the number of pipeline stage is less than 9. With the presence of pipeline registers, the activity of the combinational circuit can be better manipulated. Since the number of concurrently processed path is reduced to 2, RAM could be used as the storage of the path metrics. As long as the power and area overhead is acceptable, pipelined structure has many more advantages over several parallel butterfly units. This design is implemented in the project.

Interface

The interface of the path extension unit is shown in figure 3.10. and figure 3.12. The functions of the signals are described below.

Flush signal is used to clear all the content in the pipeline registers. This signal asserts when the Viterbi decoder is in the **Idle** state and **read soft bit** state. Every time one stage of path extension is over, the register content need to be reset.

Step signal is an integer that ranges from 0 to 31. Every time a path extension stage starts, its value will be increased by 1 in every clock cycle, starting from 0. Once its value reaches 2^{K-2}, the path extension is finished. This value is propagated through the pipeline stages.

Enable signal enables the activity of this unit. It's also propagated through the pipeline stages.

Rotate counter is a control signal that is used to calculate the index to the memory. The index algorithm will be described in the algorithm subsection.

K5 asserts if current data block is encoded with an encoder of constraint length 5. It's derived from the value of *poly_v* by the Viterbi control unit.

K7 asserts if current data block is encoded with an encoder of constraint length 7. It's derived as *K5*.

Threshold is the relative threshold for the current data block. It's given by the DSP unit.

Last PM max is the metric of the last path extension stage's best path. Together with *threshold*, they will form the absolute threshold of the current stage.

Poly_v is the specifier of the coding polynomials. It's given by the DSP unit.

Reset PM latch is used to clear the stored values of the path metrics. Such an operation is needed before each data block arrives.

TB data is the decision information generated from the merge process. Controller should store them in the memory properly.

PM max is the metric of the current stage's best path.

Index max is the index of the current stage's best path.

Sb0, sb1 and sb2 are the soft decision bits from the **soft bit decoder** unit.

Structure design

The structure of the path extension unit is as shown in figure 3.12. As shown in the figure, the path extension unit is 3-stage pipelined, thus the timing of the design is guaranteed.
The function of each block in the figure are described below.

Even memory: This memory stores half amount of the path metrics. The memory cell is a 10-bit by 32-word register file. The data sheet of this cell could be found in the appendix B(MG00032010020). The memory has one read port and one write port. The read is asynchronous, while the write is clocked. This memory can store 32 paths metric, each of which is 10-bit. The selection of this memory is based on two reasons. Firstly, the relative threshold is an integer in the range from 0 to 511. After every path extension, the path could get at most 6*16=96 points of credits or penalty, thus the biggest possible difference between the highest metric and the lowest metric can never exceed 511 + 2*96=703. If the metrics are normalized properly, 10 bits of storage for the metric should be enough. Secondly, the total number of paths is 64. Since the paths are extended in pairs, the ideal choice of memory is a dual-read-port and dual-write-port 10-bit by 64-word memory. Unfortunately, such a memory is not available, thus an alternative must be found. By carefully manipulating the index, it is possible to use two single-port-read and single-port-write 10-bit by 32-word memories to store the metrics.

Odd memory: This memory is the same as the **even memory**. It stores another half of the path metric.

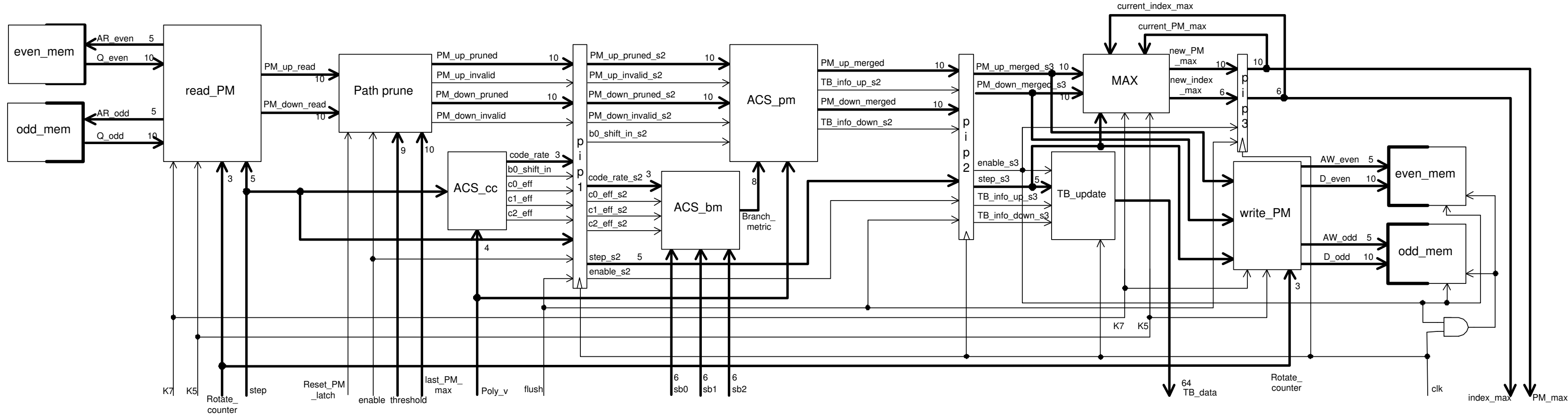


Fig. 3.12. The pipelined path extension unit

Read PM: This is the index circuit that fetches the path metrics from the **even memory** and the **odd memory**. This unit uses signal *step* as the value of J in the figure 2.15 to fetch path metric J and $J+2^{K-2}$ from the memory and send them to the **path prune** unit. It is basically an indexing and multiplexing circuit.

Path prune: This unit takes the *threshold* and the *last PM max* as input to find the absolute threshold for this PE stage. If the path J or $J+2^{K-2}$ cannot pass the threshold, the corresponding output metric will be replaced with an invalid value -512. As a consequence, an output flag will be set to 1. The flag will be used by the next two pipeline stages for power saving.

ACS cc: Each edge on a trellis diagram is related to a speculated encoded input. The input is needed for the branch metric calculation. The function of this unit is to find the input C' that corresponds to one of the edges on the butterfly unit. Although there are another 3 edges on a butterfly unit, their corresponding inputs could be derived from the output of this unit.

Pip1: This pipeline register passes input to the next stage if the enable signal is high. It is also used to isolate the inputs to the second pipeline stage circuit. The two input path metrics could only be passed to the next stage if their corresponding flag signals specify that they are valid, otherwise, the register will keep the previous metric value. If both metrics are invalid, the registers will hold two path metrics and the five outputs from the **ACS cc** unit. If this happens, most of the arithmetic units in the second pipeline stage will have no activity.

ACS bm: This unit reads the corresponding input of edge $J \rightarrow 2J$ and the soft-decision bit decoded by **soft bit decoder**. The output of this unit is the branch metric of this edge. This unit will not consume power if neither of the paths is valid.

ACS pm: By using the branch metric of path $J \rightarrow 2J$ and the two input metric, this unit can calculate the path metrics of all the four paths in the butterfly unit and merge them. The output of this unit is the metrics of the winning path during the path merge and two trace back decision bits. If one of the input metric is invalid, the path merge's winner is apparent. If both input paths are invalid, both output metrics will be the invalid metric value -512, and the trace back decision bits could be random.

Pip2: The second pipeline register simply passes input to the next pipeline stage if enabled. No operand isolation is done by this unit.

TB update: This unit is a 64-register buffer. In every clock cycle two decision bits will be saved in this unit. When one stage of path extension is over, the latched 2^{K-1} bits will be stored in the trace back memory and input buffer.

MAX: This unit has three metric inputs: two metrics from the **ACS pm** unit and one metric latched at this unit's output. Basically this unit outputs the best path's metric and index among the three inputs. Together with the latch **pip3**, the actual output after one stage of path extension is the best path's metric and index of this PE stage.

Write PM: This unit's function is almost the same as that of **read PM**. The only difference between them is that the **read PM** performs the asynchronous read, while the **write PM** performs the synchronous write. Otherwise, the index calculation and multiplexing circuits are the same.

Pip3: This unit is a storage of the temporary best path rather than a pipeline register. The latched metric is reset to -512 every time a path extension stage starts. The input is latched at the output only if the enable signal is high.

Pipelined operation

The operation performed in each pipeline stage could be called **read and prune**, **path merge** and **store path metric**. The input signal *step* from the Viterbi control unit is used as value of J in the butterfly unit. To properly control the pipeline, the signal *step* is fed into the first pipeline stage and propagated through the pipeline circuit. As described before, the value of signal *step* is increased by one in every clock, thus the operations the pipeline perform could be described as in figure 3.13.

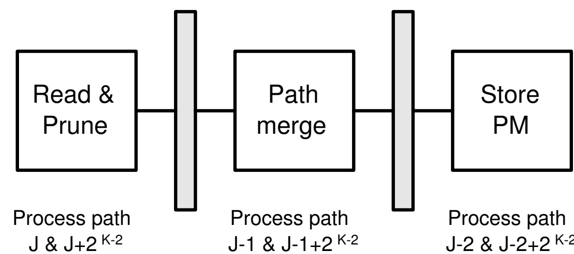


Fig. 3.13. The pipelined operation

3.3.5 Algorithms

By now, the complete architecture of the Viterbi unit has been introduced. The critical part of the design **path extension** unit is implemented with a low area pipelined structure. Due to the existence of pipeline registers, the activity in the circuit could be controlled by the threshold. However, all the units are introduced as black box. This subsection will detail the algorithm of the Viterbi unit.

As shown in figure 2.18, the algorithm of the design is also a major factor to reduce the power consumption of the circuit. In this subsection, the algorithms of the most important units are described. Some units' algorithm has taken the power consumption as an issue, while the others are decided by the structure. To make them easier to understand, they are introduced in the order of the decoding process.

reset PE memory and pipeline registers

When one stage of path extension starts, the previous path metric is read from the **odd memory** and **even memory**. But if the decoder is doing the path extension for the first time, the output of the memory should be an initial value. Thus, the content in the **odd memory** and **even memory** should be reset before a data block is decoded. One possible way to reset the memory is to add a **reset memory** state in the Viterbi state diagram between *idle* state and **read soft bit** state. The decoder could use 32 clock cycles to clear the content in the memory in that state. If the memory is reset by another unit besides the **write PM** circuit in the path extension unit, 20 2-to-1 MUX must be connected to the write data bus of the memory units. If the reset is performed by the **write PM** circuit, the controlling circuit will be more complicated than the existing circuit. In this design, the memory reset is not really performed on the memory, but on some other unit.

The other way to resolve this problem is to replace the data read from the memory at the first path extension stage. As shown in figure 3.12, a signal called *reset PM latch* is connected to the **path prune** unit. This signal is controlled by the **Viterbi control unit**. If PE counter in figure 3.11 is 0, which means the first path extension stage is not finished, the *reset PM latch* will be set to 1. While this signal is 1, the path prune unit's output metric will be set to -256, which is the minimum metric of a survivor path. This memory reset is easier to implement, but cost 20 2-to-1 MUX. The power consumption of such a small circuit could be neglected.

Note that this reset method makes all the paths to be possible initial state of the decoding. If the initial decoding state should be a certain state, e.g. state 0, this reset method can still work with following modification. The **path prune** unit can read signal *step* for more information. When the decoding starts and the path metric need to be reset, and if the signal *step* is 0, the **path prune** unit knows the output is the path 0 and 2^{K-2}, thus it sets the output *PM up pruned* to -256 but *PM down pruned* to invalid number -512. If the *step* is not 0, while the signal *reset PM latch* is still 1, the path prune unit sets both output metric as invalid.

The content latched on the pipeline registers need to be reset before every path extension stage. This could be done when the Viterbi is in the *read soft bit* or the *soft bit decode* state. Reset operation is done by using the signal *flush*. This signal becomes 1 when the Viterbi control unit is in the *read soft bit* state. The pipeline registers are implemented with asynchronously-reset flip-flops, so the flush signal should be protected with a flip-flop to prevent spurious transitions. The protected flush signal is set to 1 when the Viterbi unit is in the *soft bit decode* state, and becomes 0 when the path extension starts.

Soft-decision bit decoding

As shown in figure 3.2, there are two possible formats of the soft-decision bits. If the encoded data are represented by 5-bit integer, the decoding will only be a direct bit-mapping. If the encoded data is 4-bit integer, the decoded integer's lowest 4 bits will be the copy of the encoded integer, while the decoded interger's MSB should be duplicated from the MSB of the encoded data. E.g a 4-bit signed integer *abcd* is equal to a 5-bit signed integer *aabcb*, while the *a* is the value of the 4-bit integer's MSB. This transformation holds because the VHDL uses two's-complement format to represent the signed integer.

Combining soft-decision bits

The table 2.3 shows that the code rate(r) of GSM convolutional code varies from 1/2 to 1/6. Even if the code rate is sometimes lower than 1/3, the actual generator polynomial used in a coding scheme never exceeds 3. E.g. the 12th convolutional code on the table has a code rate of 1/5, but the second encoded bit(G4/G6) is duplicated from the first one(G4/G6), while the last encoded bit(1) is duplicated from the fourth one(1). For each code rate, the duplication method is fixed. The table 3.2 shows the relationship between the code rate r and the duplication method.

code rate	output code
1/2	X,Y
1/3	X,Y,Z
1/4	X,Y,Z,Z
1/5	X,X,Y,Z,Z
1/6	X,Y,Z,X,Y,Z

Tab. 3.2. Code rate and code duplication

This special characteristic could be used to reduce some cost of the circuit. As described in section 2.4.1, conversion from soft-decision bits to credits requires a sign comparison. If all the 6 soft-decision bits are generated from different polynomials, totally 6 sign comparison is needed. But if 2 soft-decision data are generated from the same polynomial, they can be combined into one integer for only one sign comparison.

The **soft bit decoder** takes decoding parameter *poly_v* as an input. With this parameter, the code rate could be known by the **soft bit decoder**. Once the code rate is known, the soft bit decoder will use table 3.2 to combine the soft-decision data into 3 output. These three integers are the outputs that are sent to the **path extension** unit. The local code generation circuit **ACS cc** in the **path extension** unit generates the encoded input. By using the same principle, the output of the **ACS cc** unit could be reduced from 6 bits to 3 bits, thus saves 3 pipeline registers.

To reduce the unnecessary activity in the circuit, especially the adder, the circuit in the figure 3.14 is used to combine the soft-decision data. This circuit uses one adder if the code rate is 4, or two adders if the code rate is 5 or all three adders if the code rate is 6.

This circuit is placed in the **soft bit decoder** unit instead of being placed in the **ACS bm** to remove the critical timing path in the second pipeline stage. Experiments have shown that the power and area cost could be reduced if this circuit is placed outside of path extension unit.

Encoded input calculation

The encoded data on an trellis edge is calculated by **ACS cc** unit. This unit takes the control signal *step*, which is used as the value of J, and *poly_v* as input and generates the encoded data. By using the soft-decision bit combination principle, at most only three outputs are needed for the later processes. The **ACS cc** unit only calculates the encoded data for one of the edge on the butterfly

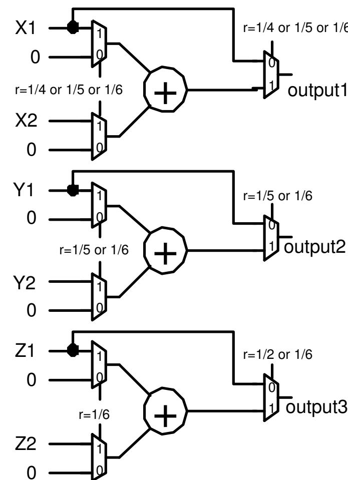


Fig. 3.14. The combination circuit for soft-decision bit data

unit. This edge starts from the node J, and its U' is 0(see section 2.3.3). If the convolutional encoder is an RSC class encoder, the edge could end at state 2J when the feed back is 0 or at state 2J+1 when the feedback is 1. The edge will be call **B0** in the following descriptions.

The encoding process has the following four steps. The first step is to find the input of the encoder register chain. For non-RSC coding(figure 2.6), this value for the edge B0 is U', which is 0. For RSC coding(figure 2.7), this value for the edge B0 is R, which is U' XOR-ed with the feedback. Since the value of U of edge B0 is 0, the value of R is exactly equal to the feedback. In GSM standard, the coding polynomials used to construct feedback are G0, G3, G4 and G6. To implement all these polynomials for the edge B0, totally 7 XOR gates are needed. Based on the value of *poly_v*, the decoder will choose one of the output from those 4 feedback circuits that is suitable for current coding scheme to calculate the value of R. The result of the first step is also assigned to the output signal *B0 shift in* and sent to the **ACS PM** unit for further use.

The second step is to calculate the output of all the 8 GSM generator polynomials. This part costs totally 19 XOR gates.

The third step is to choose some of the coding polynomial's output as the output for the current coding schemes. The selection is also based on the value of *poly_v* and table 2.3.

The last step is to combine the output into 3 coded bits. The selection is based on the value of *poly_v* and table 3.2.

The key design issue of this circuit is the fixed configuration of the code generation circuit. In all the circumstances, the output of all the feedback and generator polynomials are calculated. Even if it is possible to implement a dynamically configurable encoder with less XOR gates and more complicated control circuit, it might not be worthwhile due to the small size of the circuit.

Branch metric calculation

There are two major operations in this process: To convert the soft-decision data into credits and to accumulate the credits into the branch metric.

The conversion from soft-decision data to credit is described as in figure 3.15. In the diagram the signal *MSB(soft-decision data)* is the sign bit of the soft-decision data. If the sign bit is equal to the coded data, which implies that the sign of the soft decision data does not match the coded data, the negated absolute value of the soft decision data is used as credit. Otherwise, the absolute value of the soft decision data is used as credit.

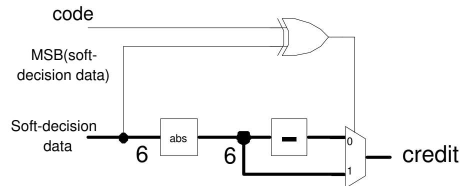


Fig. 3.15. Calculation of credit

The credit accumulation circuit is shown in the figure 3.16. If the code rate is 2, the right-sided adder has constant input, thus consumes little dynamic power.

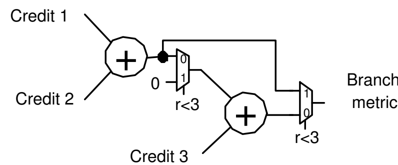


Fig. 3.16. Branch metric calculation

Path merge

As mentioned before, the output of **ACS bm** is only the branch metric of edge B0. To perform the path merge on all the four branches, the branch metric for each edge on the butterfly unit must be known. Fortunate enough, the other branches' metrics could be derived from the metric of B0.

To derive the metrics of the other three metric, the characterizes of each edge must be studied. Here the edges on the butterfly unit are named B0, B1, B2 and B3. The characteristics for each edge are:

B0 starts from state *J*. The value of U' on this edge is 0.

B1 starts from state *J*. The value of U' on this edge is 1.

B2 starts from state $J+2^{K-2}$. The value of U' on this edge is 0.

B3 starts from state $J+2^{K-2}$. The value of U' on this edge is 1.

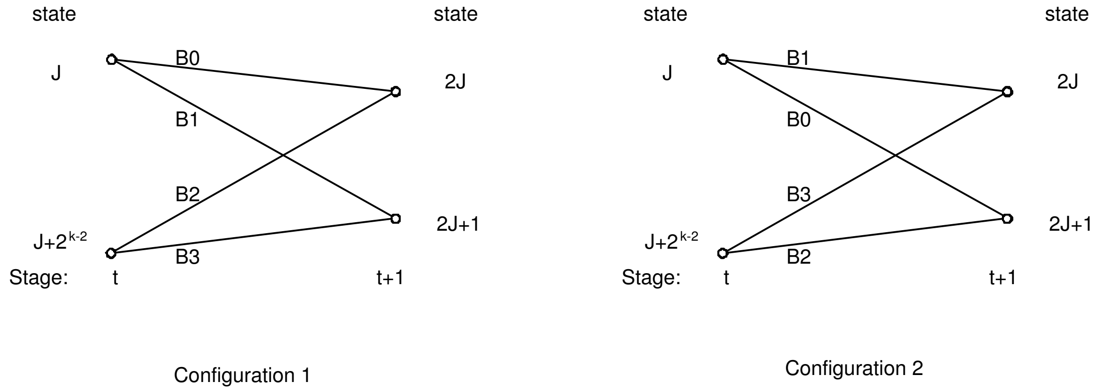


Fig. 3.17. Butterfly unit configurations

As shown in figure 3.17, there are two possible configurations of the butterfly unit. The first configuration suits for all the non-RSC butterfly units. If the B0 edge's register chain input(R) of an RSC butterfly unit is 0, the first configuration is suitable for this butterfly unit. Otherwise, the second configuration is suitable for this butterfly unit. For non-RSC codes, all the butterfly units use the first configuration. For any RSC code, half amount of the butterfly units use the first configuration while the other half use the second configuration. The output signal *B0 shift in* of *ACS cc* unit can indicate the current butterfly unit's configuration.

Taking the edge B1 as an example, its speculated encoded input C'_{B1} could be derived from C'_{B0} as follows. The value of C' is XOR-ed from the encoder shift register chain content and the register chain input (U' or R). Since the content in the register chain for B0 and B1 are both J, but register chain input are opposite, the C'_{B1} should be exactly the same as $\overline{C'_{B0}}$. By using the similar approach, the value of C'_{B2} could be found as $\overline{C'_{B0}}$ and the value of C'_{B3} could be found as C'_{B0}. The table 3.3 summarizes the relationship of all these four edges.

Edge	Register chain input	Right-most bit in the register chain	Encoded data	Branch metric
B0	x	0	C'	bm
B1	$\overline{x}$	0	$\overline{C'}$	$\overline{bm}$
B2	x	1	$\overline{C'}$	$\overline{bm}$
B3	$\overline{x}$	1	C'	bm

Tab. 3.3. Branch metrics derived from B0

The path merge is performed by the **ACS pm** unit. The input of the **ACS pm** includes:

PM up pruned s2 is the metric of current path J.

PM up invalid s2 specifies whether the current path J has passed the threshold.

PM down pruned s2 is the metric of current path $J+2^{K-2}$.

PM down invalid s2 specifies whether the current path $J+2^{K-2}$ has passed the threshold.

B0 shift in s2 specifies which configuration of butterfly unit should be used.

The output of the **ACS pm** includes:

PM up merged is the metric of path $2J$.

TB info up s2 is the trace back decision bit of state $2J$.

PM down merged is the metric of path $2J+1$.

TB info down s2 is the trace back decision bit of state $2J+1$.

To make good use of table 3.3, the unit **ACS pm** should have the structure as in figure 3.18. Although the trace back decision bits are not shown in the figure, they are simply 2 bits generated from the selection step. If the path extended from the state J is the winner, the trace back data will be 0, otherwise, it will be 1. These two decision bits will be propagated to the output in the same way as the metrics are propagated. The four extra MUX circuits are needed to deal with the two possible configuration of the butterfly units and the invalid paths.

Path prune and normalization

In order to prevent the path metric's storage overflow, the metrics must be normalized in each path extension stage. The normalization in this design is done together with the path pruning.

The normalization is a short but tricky process. It is closely related to the relative threshold (Threshold), which is given as a decoding parameter, and the best path's metric of last path extension stage(last PM max). In the normalization process, the normalized metric is found by the equation

$$Normalized_metric = input_metric - (last_PM_max + 256 - threshold) \tag{3.1}$$

To understand why the normalization is carried out in this way, the path pruning should be explained first. The path prune is simply a comparison between the normalized metric and the constant -256. If the normalized metric is larger than -256, the path is considered to have passed the threshold. Why does this mechanism work? An example is demonstrated as following.

In extreme case (r=1/6, 5-bit soft-decision bits) the maximum number of credit a path can accumulate from one stage of extension is 96. Thus, the input metric could be represented by the equation

$$input_metric = last_PM_max - m \qquad m \in [0, threshold + 2*96] \tag{3.2}$$

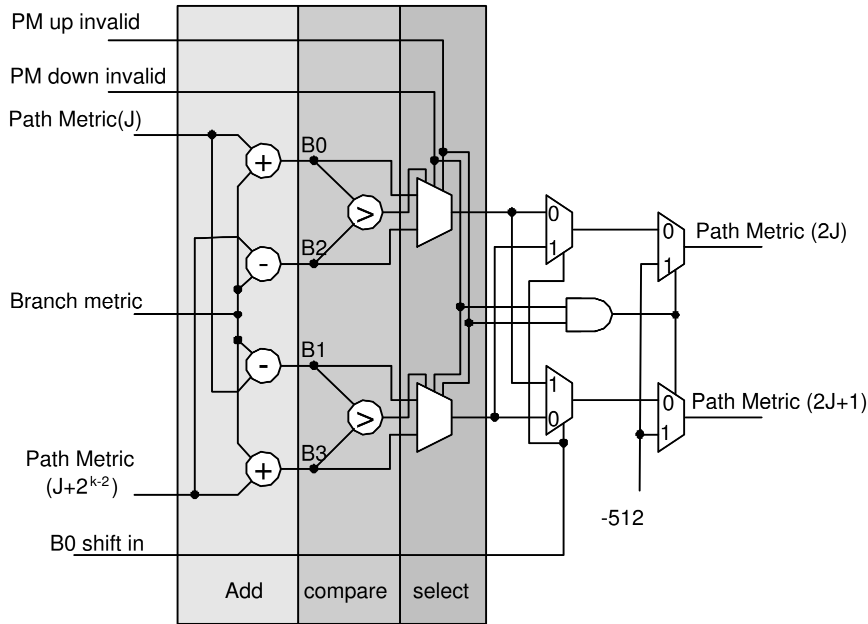


Fig. 3.18. Path merge unit configurations

Substitute equation 3.2 into 3.1, the normalized path metric could be represented as

$$Normalized_metric = Threshold - 256 - m. \quad (3.3)$$

When the normalized threshold is compared with -256 as shown in equation 3.4, the comparison is actually made between *threshold* and *m*.

$$Threshold - 256 - m \geq -256 \Longrightarrow Threshold \geq m \quad (3.4)$$

When the path metric is normalized, its value is in the range of -256 and $-256+threshold$. After the path is extended, the metric is in the range of $-256-branch_metric$ and $-256+threshold+branch_metric$. In extreme case when the threshold is 511 and the branch metric is 96, the path metric is in the range of -352 and 351. In this way the overflow is prevented.

The activity in the path pruning the normalization is carefully considered. The calculation $lastPMmax + 256 - threshold$ is made only once per path extension stage, since the value of *last PM max* is only updated once per extension stage. The comparison for the path pruning is made between a variable metric and the constant -256, thus it only costs one inverter and one AND gate. The only arithmetic unit that consumes power should be the subtraction unit used for the inevitable metric normalization.

Storage indexing

As discussed before, the path metric storage is two 10-bit * 32 word register files. To effectively use these two memory cells, the indexing problem must be solved. The storage indexing has two

problems: in-place storing problem and dual access problem. The solutions of these two problems are briefly discussed in [15]. Here the details of these indexing techniques are discussed. The indexing methods described here are used in both the **read PM** and the **write PM** unit.

The in-place storing problem can be resolved with an index rotation technique. Assume that a dual-read and dual-write memory is available. As shown in figure 2.15, the butterfly unit reads the path metric J and $J+2^{K-2}$ and produces the metric $2J$ and $2J+1$. While the output is produced, they cannot be stored into the memory address $2J$ and $2J+1$, since these spaces may be occupied by the resulting metric of the last path extension stage. The simplest way to resolve this problem is to use a redundant memory, but not necessary the only solution.

As shown in figure 3.19, if the butterfly unit places the output metric $2J$ into the physical address where the input J was stored, and the output metric $2J+1$ into the physical address where the input $J+2^{K-2}$ was stored, the content of the path metric memory evolves in a cyclical pattern.

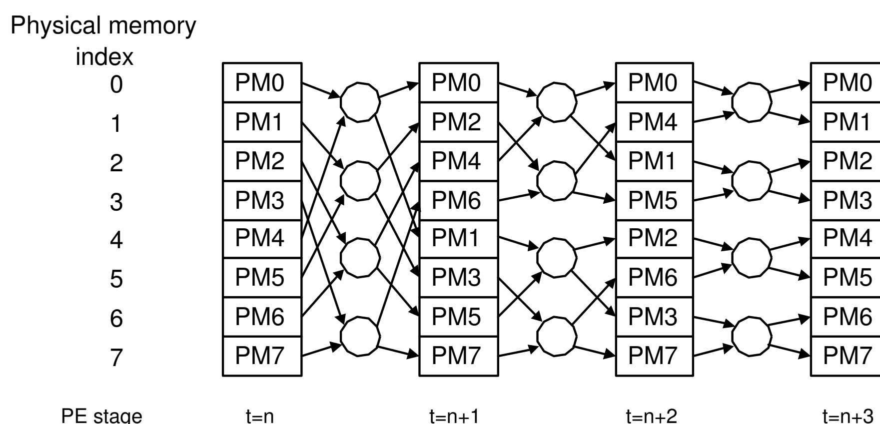


Fig. 3.19. Content evolution of path metric memory

At path extension stage n, the index of the physical memory matches the index of the path metric. After one path extension stage, the index of the physical memory cell could be found by rotating the path metrics' index one bit to the right. Similarly, at the next path extension stage, the index of the physical memory cell could be found by rotating the path metrics' index 2 bits to the right. In general, the number of rotation needed to get the proper memory cell index is "n mod K-1", where n is the number of path extension passed and K is the constraint length of the decoder. This indexing technique is used in this design.

Due to the lack of the dual-read dual-write port memory, some alternative must be used. The simplest method to deal with this problem is to use two equal-sized single-read single-write port memory. The design issue is to figure out which metric should be stored in which memory so that the butterfly unit can operate on two metrics all the time. By studying the binary presentation of the index J and $J+2^{K-2}$, one can conclude that their parity must be different since they differs only on the MSB. The same conclusion can be made on the binary presentation of $2J$ and $2J+1$. To fetch or store path metric all the time, the metrics with an even-parity index should be stored in

one memory, while those metrics with odd-parity index should be stored in another. This explains why the storages for path metrics are called even memory and odd memory.

The signal *rotate counter* that is sent to the **read PM** and the **write PM** unit specifies the number of bits the index need be to rotated. This signal is generated by a counter, which is increased after each path extension stage.

Decision bits storage

The decision bit storage could be stored with either 2*32 shift registers or 64 indexed flip-flops. Experiments shows the index circuit's power consumption could be as low as that of 2 registers, therefore the indexed flip-flops are used.

The storage is built in the **TB update** unit. It is constructed with a clock gating circuit and 64 registers. Since the path metrics are stored in pairs, totally only 32 clock gating signals are needed.

The **TB update** unit should be placed in the third pipeline stage in stead of the second stage in order to reduce the timing requirements for the **ACS bm** and **ACS PM** units. Even if this will cost 2 pipeline registers, the power consumption of the **TB update** unit can be reduced by 90%, and the area can be reduced by half.

Counter sharing in control unit

As shown in figure 3.11, there are totally 6 counters used. Since the maximum number of counter needed at the same time is 2, some of those counters can be shared. The implementation of the controller only need two counters. The PE counter and the TB counter share a 10-bit counter, while the other counters share a 5-bit counter.

Find new index

The **find new index** circuit uses an index and 64-bit trace back data to find out the index for the next trace back stage and decode one bit of output. The input index points out which state on the current trace back stage is a node of the best path, while the output index serves for the same purpose for the next decoding stage.

The correct decision bit in the 64-bit trace back data could be directly found with the index. To find the index for the next trace back stage, the butterfly unit need to be backward traversed. By using the current index as $2J$ or $2J+1$, if the decision bit is 0, the next index could be found as J, otherwise, the next index is $J+2^{K-2}$.

Due to the use of RSC code, the decoding requires more work. Firstly, the value of J in the butterfly unit must be found. This can be done by discarding the LSB of the current input index. The second step is to find out the input bit (R) to the encoding register chain when data is encoded on edge B0 of the butterfly unit(see figure 3.17). This step is exactly the same as the first step the **ACS CC** unit carries out to find the value of *B0 shift in*. The last step is to find the decoded bit. If the encoding scheme is not RSC code, the decoded bit is simply the LSB of the current index. If the encoding scheme is RSC code and the input bit R to the encoding register chain is 0, the decoded bit is the current index's LSB XOR-ed with the currently indexed trace back decision bit.

If the encoding scheme is RSC code and the input bit R to the encoding register chain is 1, the decoded bit is the index's LSB XNOR-ed with the currently indexed trace back decision bit.

3.4 CRC unit design

3.4.1 State transition of the cyclic decoder

The cyclic decoder should generate the remainder of the decoded data block with various dividing polynomials listed in table 2.1. The decoding process could be described with a state diagram as shown in figure 6. The operation performed in each state is explained as following.

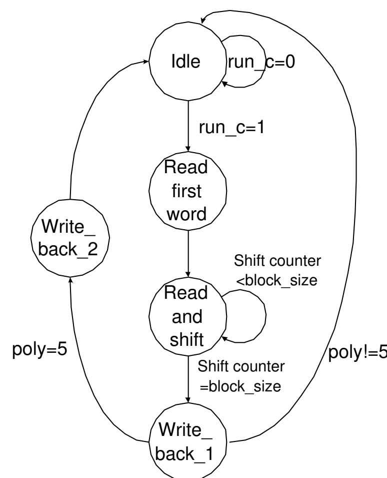


Fig. 3.20. State diagram of the cyclic decoder

1. **Idle**: The cyclic decoder goes into this state when it is reset or it finishes the syndrome generation process of the last data block. The cyclic decoder will wait in this state until the overall controller sets the signal *run_c* to high. Once the signal *run_c* goes to high, the decoder goes to the **read first word** state.

2. **Read first word**: The decoder reads the first encoded data out of the buffer memory. Since the memory is clocked, the data cannot be delivered by the buffer memory until the next clock cycle.

3. **Read and shift**: In this state, the decoder reads the data from the buffer memory and sends the data into a decoding circuit one bit per clock cycle. Since the data delivered by the buffer memory is 32 bits in parallel, the decoder only need to access the memory every 32 clock cycles.

4. **write back 1**: After all the data are sent to the decoder, the remainder of the division should be stored in the memory. The decoder can store as much as 32 bits of the remainder into the buffer memory in this state. If the remainder is less than 32 bits long, the decoding process is finished and the decoder goes to the **idle** state. Otherwise the decoder goes into state **write back 2** and continues with the remainder write back.

5. **write back 2**: In case the decoder need to store more than 32 bits of the remainder into the buffer(poly_c=5), the decoder goes into this state. After the decoder stores the rest of the data into the memory, the decoder will go into state **idle**.

3.4.2 Structure of the cyclic decoder

Figure 3.21 shows the structure of the cyclic decoder. The **control** unit realizes the state transition described in figure 3.20 and transmits the data to buffer memory. The **shift chain** unit is a configurable division circuit.

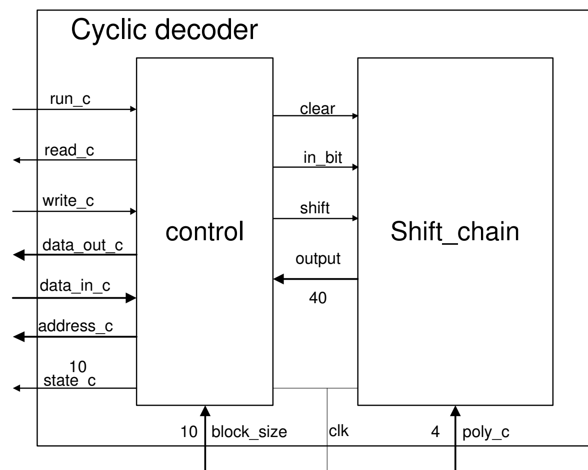


Fig. 3.21. Structure of the cyclic decoder

The division circuit **shift chain** interfaces to the control unit through the following signals.

Clear signal resets the content latched in the division register chain.

Shift signal enables the division circuit. If this signal is set to 0, all the registers' clock signal in the division circuit is gated.

In bit is the data input to the decoding circuit. The control unit sends the data block to the division circuit one bit per clock cycle.

Output is the output of all the registers in the division circuit. By using the parallel connection, all the contents latched by the shift registers can be read concurrently.

3.4.3 Structure of the configurable decoder

The **shift chain** unit is constructed of 40 stages of shift registers with xor gates and multiplexors. Figure 3.22 shows the structure of this decoder. For a certain decoding polynomial *poly_c*, if a certain stage in the register chain should not take feedback as an input, the MUX on the top will pass a zero to the xor gate. The xor gate will then act as a transparent buffer. Another input to the xor gate is chosen between the output of the previous stage and the input bit (*in_bit*) sent from the controller. In case this stage is currently configured as the first stage of the dividing polynomial, it will take *in_bit* as the input. Otherwise, the input will be connected from the output of the previous stage. The remainder of the division is not delivered to the control unit one bit every clock cycle but in parallel. The controller is responsible to find out which stages' output is the part of the syndrome for the current configuration.

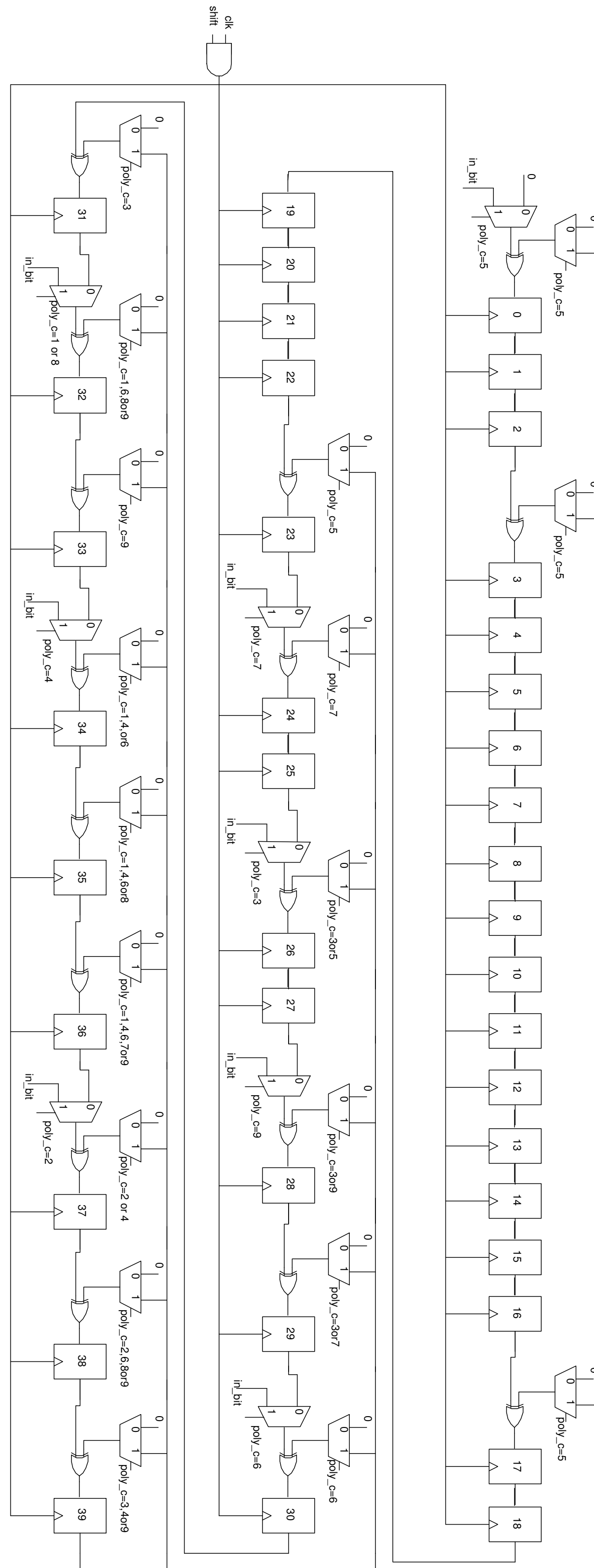


Fig. 3.22. Implementation of the shift chain unit

4 Design evaluation

4.1 Test bench

The implemented test bench can simulate all the cyclic code encoding and convolutional encoding methods described in [2]. Besides the functional test, the test bench is also needed for the power evaluation of the design. The symbolic structure of the test signal stimulator is shown in the figure 4.1. The source code of the test bench is attached in the appendix A.

The encoding part of the test bench has an interleaved structure that is similar to the decoding unit. Such a structure can support various operation modes. The cyclic encoder in figure 4.1 is similar to the **shift chain** unit in figure 3.22. The convolutional encoder is similar to the **ACS cc** unit in the pipelined path extension unit. Since the **ACS cc** unit and the **shift chain** unit are introduced in the earlier sections, the algorithm of the encoders in the test bench will not be repeated here.

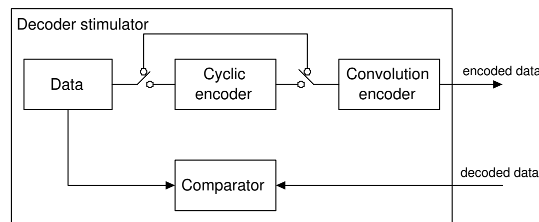


Fig. 4.1. Test bench structure

The user of the decoder only need to specify the operation mode, the cyclic and convolution coding method, the decoding threshold, and the data block and data block's size. The generation of the encoded data is automated. During the test data generation, no noise is added into the data, nor the encoded data are punctured.

The interface between the test stimuli and the decoder is identical to the connection on figure 3.6. During the test scenario, the interface signals interact as on figure 3.7. When the test bench receives the decoded data, they will be compared with the encoded data. If the decoded data matches the encoded data perfectly, a flag *success* will be set. The syndrome of CRC check is read from the decoder into the test bench, but not checked. The value of the syndrome can easily be read from the wave form of the interface signal, since it is presented by the last 2 data on the data bus.

4.2 Function verification

Because the decoder is highly configurable, the test for all the cases is difficult to make. To cover most of the single cases and some of the mixed decoding schemes, the following 20 cases are tested.

Case	Op_mode	*poly_v*	*poly_c*	Soft decision	Threshold	Data block size	Result
1	Viterbi only	1	-	4-bits	200	30	Correct
2	Viterbi only	3	-	4-bits	200	100	Correct
3	Viterbi only	5	-	4-bits	200	200	Correct
4	Viterbi only	7	-	4-bits	200	300	Correct
5	Viterbi only	9	-	4-bits	200	400	Correct
6	Viterbi only	11	-	4-bits	200	600	Correct
7	CRC only	-	1	-	-	25	Correct
8	CRC only	-	3	-	-	300	Correct
9	CRC only	-	5	-	-	400	Correct
10	CRC only	-	7	-	-	500	Correct
11	CRC only	-	9	-	-	600	Correct
12	Both	2	2	5-bits	100	30	Correct
13	Both	4	4	5-bits	150	100	Correct
14	Both	6	6	5-bits	200	200	Correct
15	Both	8	8	5-bits	250	300	Correct
16	Both	10	9	5-bits	300	400	Correct
17	Both	12	9	5-bits	350	500	Correct
18	Both	13	9	5-bits	400	600	Correct
19	Both	12	9	5-bits	450	600	Correct
20	Both	12	9	5-bits	500	600	Correct

Tab. 4.1. The test cases and results

The test is made on both the RTL description and the gate level model with no delay added. The biggest data block (600 bit data + 12 bit remainder) could be decoded in 280 μs

Two cases are not tested: the extremely low threshold case and the very small data block case. In low threshold case, if the threshold is low enough and the data contain noise so that the correct path could be pruned, the decoded data could be wrong. This is the normal decoding behavior, therefore it is not considered as an implementation error.

In small data block case, since the Viterbi decoder assumes no initial state, the first few round of path extension cannot guarantee that the correct path could be found. Thus trace back is made on the wrong path. Experiments show that to guarantee to find the best path, at least 12 stages of path extension are required. Since all the data blocks described in the GSM standard are much larger than 12 bits, this problem is not considered as a potential error. The problem can be solved by using the modified path metric reset method discussed in section 3.3.5.

4.3 Timing verification

4.3.1 Gate level delay

The gate level delay information could normally be extracted from the synthesized gate level description of the circuit. By using SYNOPSYS, the delay information of gate level model could be saved in Standard Delay Format(SDF) and exported to ModelSim for timing simulation.

The library GS50's vendor, Texas Instrument(TI), offers an alternative to do the gate level verification. The packages delivered by TI include a VITAL library for behavioral verification, a SYNOPSYS model for gate level simulation and a PRIME TIME model for gate level delay verification. Both the VITAL model and the SYNOPSYS model has timing generic described, but they are not consistent. E.g. the VITAL model of the input buffer memory has 98 timing generics, but the SYNOPSYS model has more than 1200 timing generics. As a consequence, the SDF files generated by the SYNOPSYS cannot be used for timing verification with the VITAL model. In order to create a usable SDF file, the PRIME TIME compiler(pt_shell) must be used. By using the given PRIME TIME model and the gate level model generated from SYNOPSYS, the pt_shell should be able to create a SDF file that is compatible with the VITAL model.

Since the pt_shell is unavailable at this moment, the delay verification cannot be done in this project. The timing information of each sub-unit is reported and recorded in the following sections as a compromise.

4.3.2 Layout level delay

Due to the lack of layout route tool, the layout verification is not made for this design. The timing information of each sub-unit is reported and recorded in the following sections as a compromise.

4.3.3 Unit delay

The unit delay from the synthesized gate level circuit is listed in table 4.2. When the circuit is synthesized, the synthesis effort is medium and no power/area optimization constraint is set. The only timing constraint added is the 100MHz clock cycle.

4.4 Area cost

The area cost of the circuit is listed in the table 4.3.

As shown in the table, the buffer memory, trace back memory and the storages for the path metrics cost 84.8% of total area. The logic is less important an issue comparing to the memory. Since the memory size influences the decoding quality, it will be improper to reduce the memory size.

Unit	Delay (ns)
Decoder	9.65
Decoder control unit	3.9
Input buffer	1.43
Viterbi decoder	4.20
Viterbi Control unit	5.13
Trace back memory	1.45
Soft bit decoder	3.95
Find new index	1.86
Path extension	0.34
Even memory	0.97
Odd memory	0.97
Read PM	3.22
Path prune	4.41
ACS cc	3.50
Pipeline 1	0.16
ACS bm	3.07
ACS pm	3.62
pipeline 2	0.16
TB update	0.23
MAX	4.58
write PM	3.22
pipeline 3	0.16
Cyclic decoder	4.22
Cyclic decoder control	4.21
Shift chain	0.68

Tab. 4.2. The unit delay

4.5 Power consumption VS Threshold

4.5.1 SYNOPSYS power compiler

The SYNOPSYS model the power dissipation as described in section 2.6.1. The power compiler of the SYNOPSYS can extract almost all the parameters from the standard cell library to estimate the power dissipation in the circuit except one information: the circuit activity. Since SYNOPSYS design compiler and power compiler have no insight of the circuit dynamic behavior, some other tools are required.

The SYNOPSYS solves the problem by using a Switching Activity Interchange Format(SAIF) file to exchange data with RTL simulators like VSS or ModelSim. The ModelSim(later than

Unit	area (# of gates)	area (mm^2)	%
Decoder	35757	0.238	100
Decoder control unit	405	0.0027	1.1326
Input buffer	13501	0.09	37.76
Viterbi decoder	20995	0.1399	58.72
Viterbi Control unit	1258	0.0084	3.52
Trace back memory	13501	0.09	37.76
Soft bit decoder	309	0.00206	0.864
Find new index	144	0.00096	0.4027
Path extension	5777	0.03851	16.156
Even memory	1675	0.0112	4.6843
Odd memory	1675	0.0112	4.6843
Read PM	177	0.00118	0.495
Path prune	260	0.001733	0.7271
ACS cc	102	0.00068	0.2852
Pipeline 1	215	0.00143	0.6012
ACS bm	244	0.001627	0.6823
ACS pm	465	0.0031	1.300
pipeline 2	169	0.00113	0.473
TB update	359	0.0024	1.00
MAX	163	0.001086	0.456
write PM	175	0.00117	0.489
pipeline 3	97	0.00064	0.2713
Cyclic decoder	856	0.00571	2.3939
Cyclic decoder control	487	0.003246	1.3619
Shift chain	368	0.00245	1.029

Tab. 4.3. The circuit area cost

version 5.5) used in this project can interface with SYNOPSYS by using a Foreign Language Interface(FLI) called DPFLI. The DPFLI should normally be installed under the SYNOPSYS's auxiliary component directory when the SYNOPSYS is installed to the system. The exchange of switching activity is a long process. The ModelSim script and dc_shell script used in this project are attached in the appendix C. The RTL level activity extraction process is shown in figure 4.2.

Before starting the ModelSim, the RTL description of the decoder should be analyzed and elaborated with the dc_shell. After the RTL description is checked for error, a forward SAIF file should be generated. The forward SAIF is used as a table that lists all the signals to be monitored during the activity collection. Then the RTL model should be stimulated by the decoder test bench with the forward SAIF file for a certain duration. During the decoding process, the activities of the signals listed in the forward SAIF files will be monitored and stored in a backward SAIF file. For

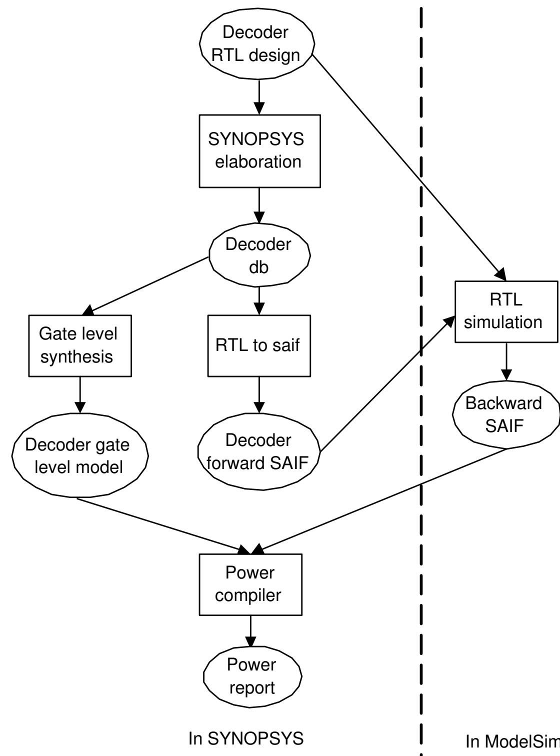


Fig. 4.2. SAIF RTL design flow

each signal, 4 timing parameters are noted done, namely the number of toggle, time in logic 1, time in logic 0 and the time in logic X. The backward SAIF signal is then sent back to SYNOPSYS power compiler for power analysis. This is the complete RTL power analysis. To do a gate level power analysis, a state-and path library must be used. Since the current version of ModelSim does not support this kind of library yet, the power analysis can only be done at RTL model in the project.

There are many issues in the analysis process. First, the registers' output MUST NOT be connected to the entities' output. If some entity's output is latched, e.g. like pipeline registers, a dummy signal must be used as a buffer. Second, the naming rules of the technique library MUST NOT be used. The default name rules must be used to make the SAIF and RTL model consistent. Third, the instances' names in the design should NOT contain capital letters. The third point is not confirmed in this project, but the case sensitivity problems have been reported by other users. Fourth, the external boundary cell should be excluded when generating the power report. If not, the power dissipation of the external boundary cell will be added into the design as an overhead.

4.5.2 Decoding dynamics

To see how the threshold affects the power consumption of the decoder, the decoding power analysis is done with one coding scheme but various threshold. The operation mode is set to both Viterbi and CRC code; the threshold varies from 511 to 100; the poly_v is set to 12; the poly_c is set to 9; the data block size is 600+12 and the soft-decision bit data is in 5-bit format. The result of the test is summarized in the table 4.4.

Threshold	511	500	450	400	350	300	250	100
pruned path	7766	8438	19718	25646	31305	34938	36237	38332
pair of path pruned	1	148	3581	7268	12064	15633	16892	18842
survivor rate %	80.2	78.5	49.6	34.5	20	10.8	7.5	2.1
Power consumption in μW								
Decoder	1300	1300	1310	1200	1090	838	768	661
Decoder control unit	8.28	8.28	8.067	7.875	7.605	7.358	7.246	7.077
Input buffer	21.3	21.27	21.28	21.307	21.296	21.211	21.136	21.015
Viterbi decoder	1170	1170	1190	1070	963	715	645	539
Viterbi Control unit	179	179	179	178	178	177	177	176
Trace back memory	11.6	11.6	11.6	11.6	11.4	11.2	11.07	10.9
Soft bit decoder	7.26	7.12	7.10	7.10	7.16	7.28	7.31	7.15
Find new index	1.22	1.20	1.15	1.11	1.04	0.93	0.82	0.71
Path extension	974	974	986	875	766	519	449	343
Even memory	177	176	196	177	159	90	71	44
Odd memory	180	180	200	181	163	91	72	44
Read PM	16.1	16.1	17.5	16.1	14.9	10.0	8.8	6.6
Path prune	31.3	39.0	46.0	44.6	44.8	27.1	24.8	9.1
ACS cc	12.2	12.2	12.1	12.0	12.2	12.2	12.1	12.2
Pipeline 1	88.5	86.8	81.6	74.0	66.2	59.9	57.2	54.2
ACS bm	42.5	43.0	35.8	29.5	23.7	15.2	13.8	10.2
ACS pm	194.0	189.3	155.4	120.3	83.0	50.3	38.6	23.4
pipeline 2	108.4	107.1	114.5	102.2	88.7	70.0	63.3	59.0
TB update	16.2	16.2	16.1	15.5	14.6	13.4	13.0	12.6
MAX	37.4	36.8	40.2	32.2	26.7	15.7	12.0	6.8
write PM	39.5	39.8	39.9	38.6	37.1	34.6	33.8	33.2
pipeline 3	28.0	27.6	27.4	27.9	27.2	25.9	24.7	23.7
Cyclic decoder	94.1	94.1	94.1	94.1	94.1	94.1	94.1	94.1
Cyclic control	33	33	33	33	33	33	33	33
Shift chain	61	61	61	61	61	61	61	61

Tab. 4.4. The power statistics

Since the Decoded data block is a K=7 612-bit stream, there will be totally 39168 paths that have ever been extended or pruned. To measure the survivor rate, two counters are built in the pipeline registers. One of the counter is used to measure the amount of paths that are pruned, while the other counter measures how many times both path J and $J+2^{K+2}$ are pruned. The pipelined path extension unit is designed in a way that if and only if neither input paths to the **ACS PM** unit is valid can the activity in the **ACS PM** unit and the **ACS BM** unit be reduced. The power reduction should be related to the pair-prune counter instead of the single-prune counter. The values of these two counters are shown in the second and third row of the table 4.4. The fourth row in the table shows the percentage of the paths which pass the threshold. As stated in [9] and [18], the survivor rate should be controlled between 50% and 25% to achieve the highest efficiency.

As the threshold decreases, the power consumption of the decoder does decrease, but only if the survivor rate is very low. Figure 4.3 shows the power consumption of decoder and path extension unit versus the survivor rate. From figure 4.3, it can be concluded the most power deduction is achieved from the pipelined path extension unit. It could be understood that the power consumption of the cyclic decoder, the input buffer, the trace back memory, the soft bit decoder and the control units are not influenced by the threshold. The power dissipation of these units may be considered as control and data storage overhead. The unit **even memory**, **odd memory** and the **ACS PM** unit use much more power than the other units at the ideal operation range. The possible power reduction method for these three units are discussed in the later chapter.

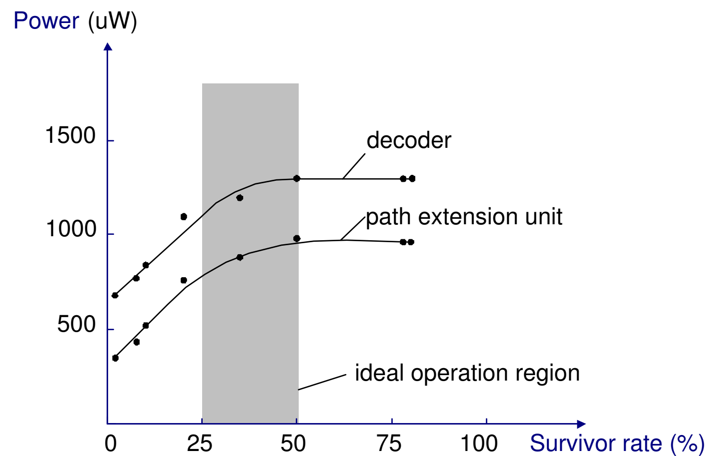


Fig. 4.3. Threshold and power consumption

5 Future work

5.1 Path metric storage power optimization

5.1.1 Structure modification

The power measurement in the section 4.5.2 shows that the metric storages use more power than any other unit in the path extension unit in the ideal operation region. This is because the metric storages are accessed all the time, even if the stored metrics in that address is invalid. To reduce the number of access to the path metric memory, the flags that indicate the validity of the metric should be stored in some memory. When the **read PM** unit tries to access the memory, if the flag indicates one or both of the metrics are invalid, the path metric memory need not to be accessed. To build the flag storage into the path extension unit, the first stage of pipeline must be modified as in figure 5.1.

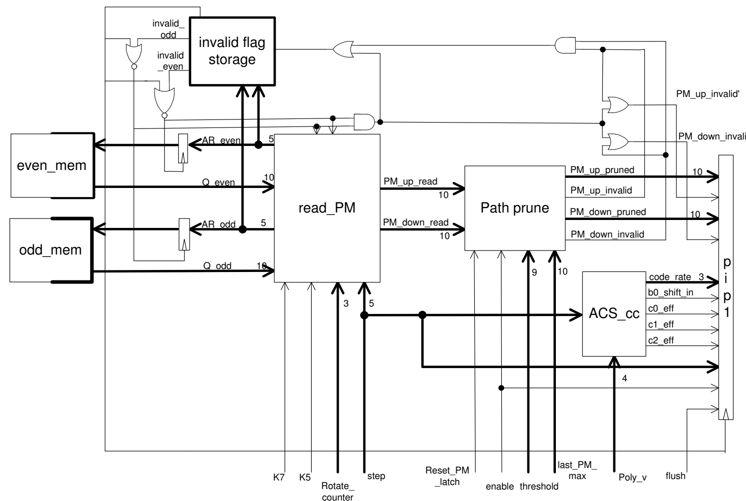


Fig. 5.1. Modified pipeline stage 1

As shown in the figure, a special 64-bit storage is added to the circuit. The invalid flags are stored into this storage at the positive clock transition. The stored invalid flag is not related to path J and $J+2^{k-2}$ but the resulting path $2J$ and $2J+1$ of the extension process. As the figure suggests, the flag for path $2J$ and $2J+1$ should always be the same. The path $2J$ and $2J+1$ could only be predicated to be both invalid if neither path J nor path $J+2^{k-2}$ is valid. There are now two possible sources of invalid flags: the **path prune** unit and the invalid flags read from the 64-bit storage. The **path prune** unit acts as described before, but the invalid flag generated from it is not sufficient. Because the output of the metric memory could be latched, the input of the prune unit could be some previous value, thus it cannot be trusted. By using the current invalid bit

fetched from the flag storage and the output of **path prune** unit together, the value of the invalid flag should always be correct.

This storage has the same indexing and replacement issue as the path metric storage, so the address value of the **odd mem** and **even mem** could be used as the index of this storage. To isolate the address bus of the path metric memory and the **read PM** unit, two flip-flops are added between the path metric storage and the **read PM** unit. The flip-flops update the latched address at the negative clock transition only if the invalid bit is 0.

The circuit should act in the following order. When a path extension is started, the **read PM** unit calculates the index and reads the invalid flags during the first half clock cycle. If any of the path is invalid, the output of the corresponding metric memory is held to the old value by the flip-flops. Depending on the output of the **path prune** unit or the flags currently fetched from the flag storage, the validity of the extended path could be predicted. The predicted invalid flags are stored into address where the flags of path J and $J+2^{k-2}$ are stored when the second half clock cycle is over.

The signal *PM up invalid'* and *PM down invalid'* are the invalid flags for path J and $J+2^{k-2}$. They are still needed for the computation of the next few pipeline stages. To correctly produce these two signals with the circuit in the figure, the flags fetched from the storage must be known by the **read PM** unit. If the paths are pruned by the **path prune** unit, the invalid flags will be set by the **path prune** unit. In case both paths are invalid as the flags in the storage indicate, the invalid flags will be set by the value fetched from the flag storage. If one of the metric is invalid, the **read PM** should feed the **path prune** unit with a metric value that is lower than the threshold to set the output flag of **path prune** unit to 1.

The path metric memories' write address bus should also be isolated with flip-flops in the last pipeline stage as for the read bus. To gate the flip-flops' clock signal, the invalid flags of path $2J$ and $2J+1$ must be propagated to the last stage.

5.1.2 Gray code indexing

The power efficiency of the extra circuit should be much less than that of the path metric storage in order to make it worthwhile. The above modification requires 64-bit registers, multiplexing circuit and demultiplexing circuit for the memory access. Since only 2 of the 64 registers are accessed at the same time, the most of the power cost should be on the multiplexing and demultiplexing circuit. The Gray code is a famous coding technique for circuit activity deduction. In order to see whether the Gray code can reduce some power for the multiplexing and demultiplexing circuit in the flag storage, an experiment is carried out. As shown in figure 5.2, a test environment of the multiplexing and demultiplexing circuit is set up. This circuit could be used as a very naive implementation for the invalid flag storage with some modification. Since the index rotation and parity check are not considered at the time the circuit is designed, the circuit uses 2*64 registers in order to prevent the replacement problem. The block **storage 1** and **storage 2** are the two 64-bit registers. These two storage are used alternatively, depending on the *selection* signal.

Each block in the circuit is explained as following:

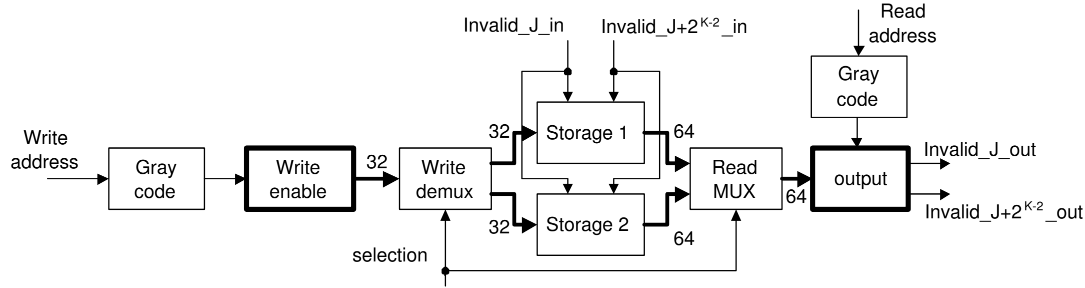


Fig. 5.2. Gray code experiment

Gray code units are used to convert the binary code into the gray code.

Write enable unit is a 1-to-32 demultiplexing circuit. The output of this circuit is used as the clock gating signal of the registers. Since the registers are updated in pairs, 32 gates should be enough. This unit is predicted as one of the most power consuming units.

Write demux unit has 32 1-to-2 demultiplexors. It switches the clock gate signals between the two storage units.

Storage 1 is one of the 64-register memory. It is in read mode when the signal *selection* is 1 and in write mode when the signal *selection* is 0.

Storage 2 is one of the 64-register memory. It is in read mode when the signal *selection* is 0 and in write mode when the signal *selection* is 1.

Read mux includes 64 2-to-1 multiplexors. It switches the output of the two storages.

Output is a two-bit 32-to-1 multiplexor. Since the output are in pairs, the multiplexor should select pairs of the latched bits as output. This unit is predicted as another most power consuming unit.

The circuit is stimulated by the increasing address signals and random input invalid bits. By using the SYNOPSYS power compiler and design compiler, the area and power cost of the circuit are measured. To compare the Gray code with the normal binary code, another set of data is measured from the circuit 5.2 but with both gray code units removed. Table 5.1 shows the measurement results.

The measurement results show that the binary code's power cost is 30% lower than that of the Gray code. The two possible explanation for this are the inaccuracy of the power compiler and the uncontrollable synthesize process. Due to the time limitation, this problem is not studied in depth or concluded.

	Gray code		Binary code	
unit	area(# of gate)	power(μW)	area(# of gate)	power(μW)
Gray code(read)	10	7.56	-	-
Gray code(write)	10	7.02	-	-
Write enable	42	15.3	42	12.6
Write demux	71	2.58	71	2.58
Storage 1	360	2.93	360	2.96
Storage 2	360	2.92	360	2.91
Read mux	140	0.87	140	0.87
Output	87	24.1	87	20.7
Total	1082	63.3	1061	42.5

Tab. 5.1. The cost of Gray code

5.2 Redundant path merge unit

The path merge unit **ACS pm** is another unit that consumes much power. Unlike the path metric memory, the power consumption of this unit is controlled by the threshold. Since the first stage pipeline register can isolate the operand of this unit, the power consumption reduces as survivor rate decreases. As shown in table 4.4, the power deduction of this unit increases very linearly as the amount of pairs of pruned paths increases. This happens because all the inputs of the **ACS PM** unit will be isolated if both input metrics are invalid. In case one of the input metric is valid, the path merge still need to be done. In order to make the path merge process more sensitive to the survivor rate, some redundance should be added to the circuit. The figure 5.3 shows the modified first and second pipeline stages of the path extension unit.

On the figure, there are two path extension units. The **ACS PM pair** unit is toggled if both paths are valid, while the **ACS PM single** unit is toggled if one of the path is valid. The **select** unit in the previous stage is used to select the valid metric as the input of the **ACS PM single** unit. Depending on the invalid flag generated from the first pipelined stage, the inputs of the ACS PM units will be isolated by the pipeline registers. The size of the **ACS PM pair** unit should be close to that of the original **ACS pm** unit, while the size of the **ACS PM single** unit should be close to half of the **ACS PM pair** unit.

Another modification on the figure is the presence of two flip-flops that isolate the branch metric. Those flip-flops are latched at the falling edge of the clock cycle. To enable these flip-flops, the corresponding input metrics of the ACS pm units should be valid.

5.3 Reduced temporary storage

The design has used two temporary storage as buffers. The first one is the unit **TB update** in the path extension unit. It is required to collect the 64-bit trace back information of one path

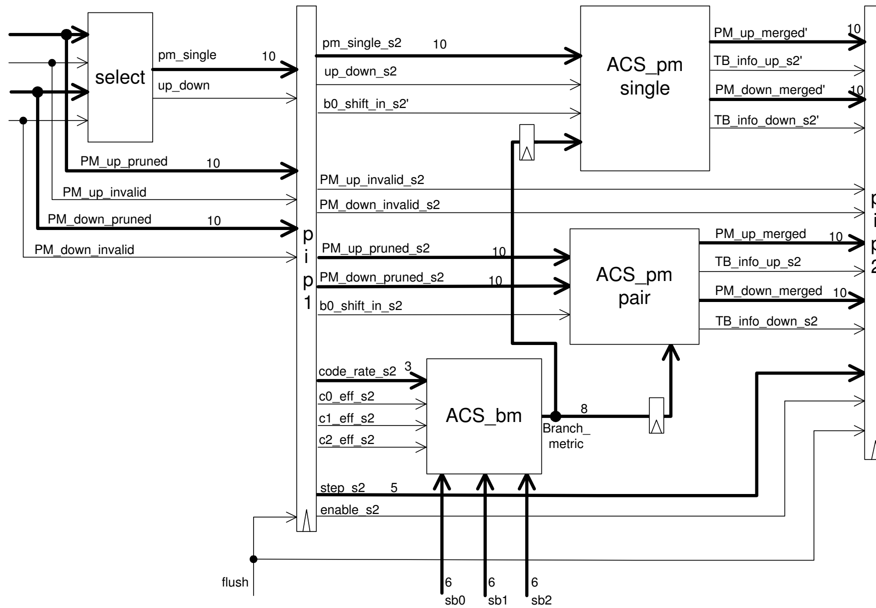


Fig. 5.3. Redundant ACS unit

extension. The second one is the 32-bit buffer in the Viterbi control unit. It is required to buffer the decoded bits during the trace back process. Those two buffers are never used at the same time, thus they could be shared.

Since the input buffer and the trace back memory support bitwise write, the size of these two memories could be reduced, e.g. to 16-bit. This may reduce the size of the circuit by 200-300 gates, but costs more energy on the memory. The balanced point may be found by making more experiments.

6 Conclusion

The current implementation of the decoder is not very susceptible to the change of the threshold. Partly this is because the decoding process for the block based data transmission is more complicated than that of sequential transmission. The CRC check, soft-decision bit handling, various coding scheme, complicated control scheme and RAM based data storage add considerable overhead to the power consumption. The original T-algorithm proposed in [4] is designed for non-punctured sequential data transmission. For their application, the trace back depth of the decoder is short enough so that all the data could be stored in the registers and accessed in parallel. For GSM decoder, many nice features of the original T-algorithm do not apply due to the bottleneck caused by the limited access port of the RAM. However, if the algorithm is carefully designed, the concept of the threshold based path pruning should be able to reduce at least 30 percent of the power.

The area cost of the design is mainly decided by the requirements of the memory. Since the data puncturing is an essential process in the GSM channel coding, the reduction of the circuit size is difficult, if not impossible. The logic design takes relatively less chip area, thanks to the timing requirements.

Due to the lack of tools, the timing verification and layout verification can not be done. The timing information can only be estimated from the timing report generated from the SYNOPSYS design compiler. In case there are timing violations, they could be fixed either by adding more pipeline stages or by optimizing the design for timing with some cost. The gate level power analysis of the design has not been made due to the limitation of the tools, but the RTL model power analysis supported by the SYNOPSYS appears to be a good tool for iterative design optimization. At the later stage of this project, many decoding activities' cost has been revealed by the power analyzer. It is a pity that the discovered targets of power optimization cannot be studied in more details.

Based on the verification of the functional test, the current design could be concluded to have achieved all the functional requirements described in the [25]. The design costs 142.8 transistors, which is equivalent to approximately 0.24 mm^2 for the given GS50 technology. Based on the RTL power analysis result reported by the SYNOPSYS, the power dissipation of the decoder is 1.3 to 1.1 mW. Consider the power consumption of a mobile phone in the stand-by mode is several mW, this unit is rather costly. The power loss in the four RAM modules is 0.150nW/MHz/gate, while the power loss in the rest of the circuit is 1.59nW/MHz/gate. These numbers are much smaller than the average 6nW/MHz/gate reported by TI. Since no other convolutional decoder of the same class has ever been found, the efficiency comparison cannot be made.

APPENDIX

A Source code

A.1 Decoder source code

A.1.1 Decoder unit

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--structure design of the decoder

library ieee;
library ti_custom;

use ieee.std_logic_1164.all;
use ti_custom.all;

entity decoder is
port(
    reset       : in std_logic;
    clk     : in std_logic;

    slave       : in std_logic;
    read        : in std_logic;
    write       : in std_logic;
    address     : in std_logic_vector(9 downto 0);

    intf        : out std_logic;

    data        : inout std_logic_vector(31 downto 0)



);
end decoder;

architecture structural of decoder is

component decoder_control_unit
port(
    reset       : in std_logic;
    clk         : in std_logic;

    slave       : in std_logic;
    read        : in std_logic;
    write       : in std_logic;
    address     : in std_logic_vector(9 downto 0);

    read_v      : in std_logic;
    write_v     : in std_logic;
    data_out_v  : in std_logic_vector(31 downto 0);
    address_v   : in std_logic_vector(9 downto 0);
    state_v     : in std_logic;


    read_c      : in std_logic;
    write_c     : in std_logic;
```

```
    data_out_c  : in std_logic_vector(31 downto 0);
    address_c   : in std_logic_vector(9 downto 0);
    state_c     : in std_logic;

    data_out_buff   : in std_logic_vector(31 downto 0);


    intf        : out std_logic;

    run_v       : out std_logic;
    data_in_v   : out std_logic_vector(31 downto 0);

    run_c       : out std_logic;
    data_in_c   : out std_logic_vector(31 downto 0);

    rw_buff     : out std_logic;
    en_buff     : out std_logic;
    data_in_buff    : out std_logic_vector(31 downto 0);
    address_buff    : out std_logic_vector(9 downto 0);
    clk_buff    : out std_logic;

        poly_v          : out std_logic_vector(3 downto 0);
        poly_c          : out std_logic_vector(3 downto 0);
        soft_bit_no     : out std_logic;
        threshold       : out integer range 0 to 511;
        block_size      : out integer range 0 to 612;


        TA      : out  STD_LOGIC_VECTOR( 9 downto 0)  ;
        TD          : out  STD_LOGIC_VECTOR( 31 downto 0) ;
        TEZ         : out  STD_LOGIC ;
        TWZ         : out  STD_LOGIC ;

        WRENZ       : out  STD_LOGIC_VECTOR( 31 downto 0) ;
        SCAN        : out  STD_LOGIC ;
        TM          : out  STD_LOGIC ;
        ATPGM       : out  STD_LOGIC ;
        CSCAN       : out  STD_LOGIC ;

        SI          : out  STD_LOGIC ;

        DFTREAD     : out  STD_LOGIC ;

        data        : inout std_logic_vector(31 downto 0)


);

end component;

component  viterbi_unit
port
(
    reset : in std_logic;
    clk   : in std_logic;

    run_v : in std_logic;
    data_in_v : in std_logic_vector(31 downto 0);
    poly_v  : in std_logic_vector(3 downto 0);
    soft_bit_no : in std_logic;
    threshold : in integer range 0 to 511;
    block_size :  in integer range 0 to 612;
```

```
    read_v : out std_logic;
    write_v : out std_logic;
    state_v : out std_logic;
    data_out_v : out std_logic_vector(31 downto 0);
    address_v : out std_logic_vector(9 downto 0)

);
end component;

component buffer_ram  port(

    A   :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TA  :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    D   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    TD  :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

    CLK  :   in  STD_LOGIC ;

      EZ      :   in  STD_LOGIC ;
      TEZ     :   in  STD_LOGIC ;

      WZ      :   in  STD_LOGIC ;
      TWZ     :   in  STD_LOGIC ;

      WRENZ   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

      SCAN    :   in  STD_LOGIC ;
      TM      :   in  STD_LOGIC ;
      ATPGM   :   in  STD_LOGIC ;
      CSCAN   :   in  STD_LOGIC ;

      SI      :   in  STD_LOGIC ;


      DFTREAD :   in  STD_LOGIC ;

      Q       :   out STD_LOGIC_VECTOR( 31 downto 0);
      SO      :   out std_logic
    );


end component;

component cyclic_decoder
port(
    reset       : in std_logic;
    clk     : in std_logic;

    run_c       : in std_logic;
    data_in_c   : in std_logic_vector(31 downto 0);
    block_size  : in integer range 0 to 612;
    poly_c      : in std_logic_vector(3 downto 0);

    read_c      : out std_logic;
    write_c     : out std_logic;
    data_out_c  : out std_logic_vector(31 downto 0);
    address_c   : out std_logic_vector(9 downto 0);

    state_c     : out std_logic
);
end component;

signal  read_v      :  std_logic;
```

```
signal  write_v     :  std_logic;
signal  data_out_v  :  std_logic_vector(31 downto 0);
signal  address_v   :  std_logic_vector(9 downto 0);
signal  state_v     :  std_logic;

signal  read_c      :  std_logic;
signal  write_c     :  std_logic;
signal  data_out_c  :  std_logic_vector(31 downto 0);
signal  address_c   :  std_logic_vector(9 downto 0);
signal  state_c     :  std_logic;

signal  data_out_buff   :  std_logic_vector(31 downto 0);

signal  run_v       :  std_logic;
signal  data_in_v   :  std_logic_vector(31 downto 0);

signal  run_c       :  std_logic;
signal  data_in_c   :  std_logic_vector(31 downto 0);

signal  rw_buff     :  std_logic;
signal  en_buff     :  std_logic;
signal  data_in_buff    :  std_logic_vector(31 downto 0);
signal  address_buff    :  std_logic_vector(9 downto 0);
signal  clk_buff    :  std_logic;

signal  poly_v          :  std_logic_vector(3 downto 0);
signal  poly_c          :  std_logic_vector(3 downto 0);
signal  soft_bit_no     :  std_logic;
signal  threshold       :  integer range 0 to 511;
signal  block_size      :  integer range 0 to 612;

signal  TA      :  STD_LOGIC_VECTOR( 9 downto 0)  ;
signal  TD          :  STD_LOGIC_VECTOR( 31 downto 0) ;
signal  TEZ         :  STD_LOGIC ;
signal  TWZ         :  STD_LOGIC ;

signal  WRENZ       :  STD_LOGIC_VECTOR( 31 downto 0) ;
signal  SCAN        :  STD_LOGIC ;
signal  TM          :  STD_LOGIC ;
signal  ATPGM       :  STD_LOGIC ;
signal  CSCAN       :  STD_LOGIC ;

signal  SI          :  STD_LOGIC ;

signal  DFTREAD     :  STD_LOGIC ;

signal  SO          :  STD_LOGIC ;

begin
controller: decoder_control_unit port map(reset,clk,
slave,read,write,address,read_v,write_v,data_out_v,
address_v,state_v,read_c,write_c,data_out_c,address_c,
state_c,data_out_buff,intf,run_v,data_in_v,run_c,
data_in_c,rw_buff,en_buff,data_in_buff,address_buff,
clk_buff, poly_v, poly_c, soft_bit_no,  threshold,
block_size,TA,TD, TEZ, TWZ,WRENZ,SCAN,TM,ATPGM,CSCAN,
SI, DFTREAD,data );

trellis: viterbi_unit port map ( reset,clk,run_v,
data_in_v,poly_v,soft_bit_no,  threshold,block_size,
read_v,write_v,state_v,data_out_v, address_v);

crc_check: cyclic_decoder port map ( reset,clk,
```

```
run_c,data_in_c,block_size,poly_c,read_c,write_c,
data_out_c,address_c,  state_c);

input_buffer: buffer_ram  port map (address_buff,
TA,data_in_buff,TD,clk_buff,en_buff,TEZ,rw_buff,
TWZ, WRENZ,SCAN,TM,ATPGM,CSCAN,SI,DFTREAD,
data_out_buff,SO);



end structural;
```

A.1.2 Decoder control unit

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--overall control unit


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity decoder_control_unit is
port(
    reset       : in std_logic;
    clk         : in std_logic;

    slave       : in std_logic;
    read        : in std_logic;
    write       : in std_logic;
    address     : in std_logic_vector(9 downto 0);

    read_v      : in std_logic;
    write_v     : in std_logic;
    data_out_v  : in std_logic_vector(31 downto 0);
    address_v   : in std_logic_vector(9 downto 0);
    state_v     : in std_logic;


    read_c      : in std_logic;
    write_c     : in std_logic;
    data_out_c  : in std_logic_vector(31 downto 0);
    address_c   : in std_logic_vector(9 downto 0);
    state_c     : in std_logic;

    data_out_buff   : in std_logic_vector(31 downto 0);


    intf        : out std_logic;

    run_v       : out std_logic;
    data_in_v   : out std_logic_vector(31 downto 0);

    run_c       : out std_logic;
    data_in_c   : out std_logic_vector(31 downto 0);

    rw_buff     : out std_logic;
    en_buff     : out std_logic;
```

```
    data_in_buff    : out std_logic_vector(31 downto 0);
    address_buff    : out std_logic_vector(9 downto 0);
    clk_buff    : out std_logic;

        poly_v          : out std_logic_vector(3 downto 0);
        poly_c          : out std_logic_vector(3 downto 0);
        soft_bit_no     : out std_logic;
        threshold       : out integer range 0 to 511;
        block_size      : out integer range 0 to 612;


        TA      : out  STD_LOGIC_VECTOR( 9 downto 0)  ;
        TD          : out  STD_LOGIC_VECTOR( 31 downto 0) ;
        TEZ         : out  STD_LOGIC ;
        TWZ         : out  STD_LOGIC ;

        WRENZ       : out  STD_LOGIC_VECTOR( 31 downto 0) ;
        SCAN        : out  STD_LOGIC ;
        TM          : out  STD_LOGIC ;
        ATPGM       : out  STD_LOGIC ;
        CSCAN       : out  STD_LOGIC ;

        SI          : out  STD_LOGIC ;

        DFTREAD     : out  STD_LOGIC ;

        data        : inout std_logic_vector(31 downto 0)


);

end decoder_control_unit;


architecture behavioral of decoder_control_unit is
type state_type is (init, wait_start, decode_param, CRC_run, viterbi_run, wait_end);

signal current_state, next_state : state_type;
signal op_mode: std_logic_vector(1 downto 0);
begin

control_logic : process (clk,slave,read, write,address,read_v,write_v,
data_out_v,address_v,state_v,read_c,write_c,data_out_c,address_c,
state_c,data_out_buff,current_state, data,op_mode )

constant param_address : std_logic_vector(9 downto 0):="1001100101";
begin

-- disable test
TA <="0000000000";
TWZ <= '0';
TEZ<= '1';
WRENZ<="00000000000000000000000000000000";
TD<= "00000000000000000000000000000000";

DFTREAD<='0';
SCAN<='0';
TM<='0';
CSCAN<='0';
ATPGM<='0';

SI<='0';
```

```
--tri-state buffer inference
data<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

case current_state is
    when init =>
--idle
        run_v<='0';
        run_c<='0';
        intf<='0';
        data_in_v<="00000000000000000000000000000000";
        data_in_c<="00000000000000000000000000000000";
        data_in_buff<=data;
        address_buff<=address;
--prepare for write
        if write='1' then
            rw_buff<='0';
            en_buff<='0';
            clk_buff<=clk;
        else
            rw_buff<='0';
            en_buff<='1';
            clk_buff<='1';
        end if;

        if slave='0' then
            next_state<=init;
        else
            next_state<=wait_start;
        end if;

    when wait_start =>
        run_v<='0';
        run_c<='0';
        intf<='0';
        data_in_v<="00000000000000000000000000000000";
        data_in_c<="00000000000000000000000000000000";

        if slave='1' then
--wait
            address_buff<=address;
            if write = '1' then
--prepare for write
                data_in_buff<=data;
                rw_buff<='0';
                en_buff<='0';
                clk_buff<=clk;
            elsif read = '1' then
--prepare for read
                data<=data_out_buff;
                data_in_buff<="00000000000000000000000000000000";
                rw_buff<='1';
                en_buff<='0';
                clk_buff<=clk;
            else
                data_in_buff<="00000000000000000000000000000000";
                rw_buff<='0';
                en_buff<='1';
                clk_buff<='1';
            end if;
            next_state<=wait_start;

        else
--start decoding
```

```
            address_buff<=param_address;
            data_in_buff<="00000000000000000000000000000000";
            rw_buff<='1';
            en_buff<='0';
            clk_buff<=clk;
            next_state<=decode_param;

        end if;
    when decode_param =>
--decode parameter
        op_mode<=data_out_buff(1 downto 0);
        poly_v<=data_out_buff(5 downto 2);
        soft_bit_no<=data_out_buff(6);
        threshold<= conv_integer(unsigned(data_out_buff(15 downto 7)));
        poly_c<=data_out_buff(19 downto 16);
        block_size<=conv_integer(unsigned(data_out_buff(29 downto 20)));

        intf<='0';
        data_in_v<="00000000000000000000000000000000";
        data_in_c<="00000000000000000000000000000000";
        data_in_buff<="00000000000000000000000000000000";
        rw_buff<='0';
        en_buff<='1';
        clk_buff<='1';
        address_buff<="0000000000";
        if op_mode="10" then
-- to CRC
            run_c<='1';
            run_v<='0';
            next_state<=CRC_run;
        else
--tp Viterbi
            run_c<='0';
            run_v<='1';
            next_state<=viterbi_run;
        end if;

    when CRC_run =>
        intf<='0';
        run_v<='0';
        run_c<=state_c;
        data_in_v<="00000000000000000000000000000000";
        address_buff<=address_c;
        data_in_c<=data_out_buff;
        data_in_buff<=data_out_c;
--handle memory access from CRC unit
        if read_c='1' then
            rw_buff<='1';
            en_buff<='0';
            clk_buff<=clk;
        elsif write_c='1' then
            rw_buff<='0';
            en_buff<='0';
            clk_buff<=clk;
        else
            rw_buff<='0';
            en_buff<='1';
            clk_buff<='1';
        end if;

        if state_c='1' then
            next_state<=CRC_run;
        else
```

```
            next_state<=wait_end;
        end if;


    when viterbi_run =>

        intf<=’0’;
        run_v<=state_v;
        data_in_c<="00000000000000000000000000000000";
        address_buff<=address_v;
        data_in_v<=data_out_buff;
        data_in_buff<=data_out_v;
--handle memory access from vitrbi unit
        if read_v=’1’ then
            rw_buff<=’1’;
            en_buff<=’0’;
            clk_buff<=clk;
        elsif write_v=’1’ then
            rw_buff<=’0’;
            en_buff<=’0’;
            clk_buff<=clk;
        else
            rw_buff<=’0’;
            en_buff<=’1’;
            clk_buff<=’1’;
        end if;

        if state_v=’1’ then
            run_c<=’0’;
            next_state<=viterbi_run;

        else
            if op_mode="11" then
                next_state<=CRC_run;
                run_c<=’1’;
            else
                next_state<=wait_end;
                run_c<=’0’;
            end if;
        end if;

    when wait_end =>
-- wait for interrupt acknowledgement
        run_v<=’0’;
        run_c<=’0’;

        intf<=’1’;

        data_in_v<="00000000000000000000000000000000";
        data_in_c<="00000000000000000000000000000000";

        address_buff<=address;

        data_in_buff<="00000000000000000000000000000000";

        if read=’1’ then
            rw_buff<=’1’;
            en_buff<=’0’;
            clk_buff<=clk;
        else
            rw_buff<=’1’;
            en_buff<=’1’;
            clk_buff<=’1’;
```

```
        end if;

        if slave = '1' then
            next_state<= wait_start;
        else
            next_state<=wait_end;
        end if;

end case;
end process;

state_transition: process(clk, reset)

begin

if reset= '1' then
    current_state<=init;
elsif clk'event and clk='1' then
    current_state<=next_state;-- after 1 ns;
end if;


end process;

end behavioral;
```

A.1.3 Input buffer

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- input buffer
-- capsulated memory

library ieee;
library ti_custom;

use ieee.std_logic_1164.all;
use ti_custom.all;

entity buffer_ram is
port(

    A   :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TA  :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    D   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    TD  :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

    CLK     :   in  STD_LOGIC ;
    EZ      :   in  STD_LOGIC ;
    TEZ     :   in  STD_LOGIC ;
    WZ      :   in  STD_LOGIC ;
    TWZ     :   in  STD_LOGIC ;
    WRENZ   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    SCAN    :   in  STD_LOGIC ;
    TM      :   in  STD_LOGIC ;
    ATPGM   :   in  STD_LOGIC ;
    CSCAN   :   in  STD_LOGIC ;
    SI      :   in  STD_LOGIC ;
    DFTREAD :   in  STD_LOGIC ;
```

```
    Q       :   out STD_LOGIC_VECTOR( 31 downto 0);
    SO      :   out std_logic

);
end buffer_ram;

architecture structural of buffer_ram is

component GL00624032040  port(

    A   :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TA  :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    D   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    TD  :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

    CLK     :   in  STD_LOGIC ;
    EZ      :   in  STD_LOGIC ;
    TEZ     :   in  STD_LOGIC ;
    WZ      :   in  STD_LOGIC ;
    TWZ     :   in  STD_LOGIC ;
    WRENZ   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    SCAN    :   in  STD_LOGIC ;
    TM      :   in  STD_LOGIC ;
    ATPGM   :   in  STD_LOGIC ;
    CSCAN   :   in  STD_LOGIC ;
    SI      :   in  STD_LOGIC ;
    DFTREAD :   in  STD_LOGIC ;

    Q       :   out STD_LOGIC_VECTOR( 31 downto 0);
    SO      :   out std_logic
    );


end component;

begin
ram_core: GL00624032040  port map (A,TA,D,TD,CLK,EZ,TEZ,WZ,TWZ,
WRENZ,SCAN,TM,ATPGM,CSCAN,SI,DFTREAD,Q,SO);
end structural;
```

A.1.4 Viterbi decoder

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--structure of the viterbi unit

library ieee;
library ti_custom;

use ieee.std_logic_1164.all;
use ti_custom.all;

entity viterbi_unit is
port
(
    reset : in std_logic;
    clk   : in std_logic;
```

```
    run_v : in std_logic;
    data_in_v : in std_logic_vector(31 downto 0);
    poly_v  : in std_logic_vector(3 downto 0);
    soft_bit_no : in std_logic;
    threshold : in integer range 0 to 511;
    block_size :  in integer range 0 to 612;

    read_v : out std_logic;
    write_v : out std_logic;
    state_v : out std_logic;
    data_out_v : out std_logic_vector(31 downto 0);
    address_v : out std_logic_vector(9 downto 0)

);
end viterbi_unit;

architecture structural of viterbi_unit is

component tb_ram  port(

    A   :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TA  :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    D   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    TD  :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

    CLK  :   in  STD_LOGIC ;

    EZ      :   in  STD_LOGIC ;
    TEZ     :   in  STD_LOGIC ;

    WZ      :   in  STD_LOGIC ;
    TWZ     :   in  STD_LOGIC ;

    WRENZ   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

    SCAN    :   in  STD_LOGIC ;
    TM      :   in  STD_LOGIC ;
    ATPGM   :   in  STD_LOGIC ;
    CSCAN   :   in  STD_LOGIC ;

    SI      :   in  STD_LOGIC ;


    DFTREAD :   in  STD_LOGIC ;

    Q       :   out STD_LOGIC_VECTOR( 31 downto 0);
    SO      :   out std_logic
    );


end component;

component soft_bit_decoder
port(
    soft_bit_coded      : in    std_logic_vector (31 downto 0);
    enable_sb_decoder   : in    std_logic;
    soft_bit_no     : in    std_logic;
    poly_v  : in    std_logic_vector(3 downto 0);

    sb0_eff         : out   std_logic_vector(5 downto 0);
    sb1_eff         : out   std_logic_vector(5 downto 0);
    sb2_eff         : out   std_logic_vector(5 downto 0)
);
```

```
end component;

component find_new_index
port(
    enable_find_new_index   : in    std_logic;
    poly_v          : in    std_logic_vector(3 downto 0);
    TB_data         : in    std_logic_vector(63 downto 0);
    current_index       : in    integer range 0 to 63;

    decoded_bit     : out   std_logic;
    new_index       : out   integer range 0 to 63

);
end component;

component PE_unit
port(

    clk     : in std_logic;
    flush           : in std_logic;

    step    : in integer range 0 to 31;
    enable  : in std_logic;

    rotate_counter  : in integer range 0 to 7;
    k5      : in std_logic;
    k7      : in std_logic;
    threshold   : in integer range 0 to 511;
    last_PM_max : in integer range -512 to 511;

    sb0     : in    std_logic_vector(5 downto 0);
    sb1     : in    std_logic_vector(5 downto 0);
    sb2     : in    std_logic_vector(5 downto 0);

    poly_v  : in    std_logic_vector(3 downto 0);

    reset_PM_latch  : in    std_logic;

    TB00_latched        : out   std_logic;
    TB01_latched        : out   std_logic;
    TB02_latched        : out   std_logic;
    TB03_latched        : out   std_logic;
    TB04_latched        : out   std_logic;
    TB05_latched        : out   std_logic;
    TB06_latched        : out   std_logic;
    TB07_latched        : out   std_logic;
    TB08_latched        : out   std_logic;
    TB09_latched        : out   std_logic;
    TB10_latched        : out   std_logic;
    TB11_latched        : out   std_logic;
    TB12_latched        : out   std_logic;
    TB13_latched        : out   std_logic;
    TB14_latched        : out   std_logic;
    TB15_latched        : out   std_logic;
    TB16_latched        : out   std_logic;
    TB17_latched        : out   std_logic;
    TB18_latched        : out   std_logic;
    TB19_latched        : out   std_logic;
    TB20_latched        : out   std_logic;
    TB21_latched        : out   std_logic;
    TB22_latched        : out   std_logic;
    TB23_latched        : out   std_logic;
    TB24_latched        : out   std_logic;
```

```
    TB25_latched        : out   std_logic;
    TB26_latched        : out   std_logic;
    TB27_latched        : out   std_logic;
    TB28_latched        : out   std_logic;
    TB29_latched        : out   std_logic;
    TB30_latched        : out   std_logic;
    TB31_latched        : out   std_logic;
    TB32_latched        : out   std_logic;
    TB33_latched        : out   std_logic;
    TB34_latched        : out   std_logic;
    TB35_latched        : out   std_logic;
    TB36_latched        : out   std_logic;
    TB37_latched        : out   std_logic;
    TB38_latched        : out   std_logic;
    TB39_latched        : out   std_logic;
    TB40_latched        : out   std_logic;
    TB41_latched        : out   std_logic;
    TB42_latched        : out   std_logic;
    TB43_latched        : out   std_logic;
    TB44_latched        : out   std_logic;
    TB45_latched        : out   std_logic;
    TB46_latched        : out   std_logic;
    TB47_latched        : out   std_logic;
    TB48_latched        : out   std_logic;
    TB49_latched        : out   std_logic;
    TB50_latched        : out   std_logic;
    TB51_latched        : out   std_logic;
    TB52_latched        : out   std_logic;
    TB53_latched        : out   std_logic;
    TB54_latched        : out   std_logic;
    TB55_latched        : out   std_logic;
    TB56_latched        : out   std_logic;
    TB57_latched        : out   std_logic;
    TB58_latched        : out   std_logic;
    TB59_latched        : out   std_logic;
    TB60_latched        : out   std_logic;
    TB61_latched        : out   std_logic;
    TB62_latched        : out   std_logic;
    TB63_latched        : out   std_logic;

    PM_max_out          : out   integer range -512 to 511;
    index_max_out       : out   integer range 0 to 63

);
end component;

component viterbi_control_unit
port(
    reset           : in std_logic;
    clk         : in std_logic;

    run_v           : in std_logic;
    data_in_v       : in std_logic_vector(31 downto 0);
    poly_v          : in std_logic_vector (3 downto 0);
    block_size      : in integer range 0 to 612;
    data_in_TB_mem      : in std_logic_vector(31 downto 0);

    --trace back data
    decoded_bit     : in std_logic;
    new_index       : in integer range 0 to 63;


    --from max
```

```
index_max       : in integer range 0 to 63;

--from PE
PM_max          : in integer range -512 to 511;


TB_info_00      : in std_logic;
TB_info_01      : in std_logic;
TB_info_02      : in std_logic;
TB_info_03      : in std_logic;
TB_info_04      : in std_logic;
TB_info_05      : in std_logic;
TB_info_06      : in std_logic;
TB_info_07      : in std_logic;
TB_info_08      : in std_logic;
TB_info_09      : in std_logic;

TB_info_10      : in std_logic;
TB_info_11      : in std_logic;
TB_info_12      : in std_logic;
TB_info_13      : in std_logic;
TB_info_14      : in std_logic;
TB_info_15      : in std_logic;
TB_info_16      : in std_logic;
TB_info_17      : in std_logic;
TB_info_18      : in std_logic;
TB_info_19      : in std_logic;

TB_info_20      : in std_logic;
TB_info_21      : in std_logic;
TB_info_22      : in std_logic;
TB_info_23      : in std_logic;
TB_info_24      : in std_logic;
TB_info_25      : in std_logic;
TB_info_26      : in std_logic;
TB_info_27      : in std_logic;
TB_info_28      : in std_logic;
TB_info_29      : in std_logic;

TB_info_30      : in std_logic;
TB_info_31      : in std_logic;
TB_info_32      : in std_logic;
TB_info_33      : in std_logic;
TB_info_34      : in std_logic;
TB_info_35      : in std_logic;
TB_info_36      : in std_logic;
TB_info_37      : in std_logic;
TB_info_38      : in std_logic;
TB_info_39      : in std_logic;

TB_info_40      : in std_logic;
TB_info_41      : in std_logic;
TB_info_42      : in std_logic;
TB_info_43      : in std_logic;
TB_info_44      : in std_logic;
TB_info_45      : in std_logic;
TB_info_46      : in std_logic;
TB_info_47      : in std_logic;
TB_info_48      : in std_logic;
TB_info_49      : in std_logic;

TB_info_50      : in std_logic;
TB_info_51      : in std_logic;
```

```
    TB_info_52      : in std_logic;
    TB_info_53      : in std_logic;
    TB_info_54      : in std_logic;
    TB_info_55      : in std_logic;
    TB_info_56      : in std_logic;
    TB_info_57      : in std_logic;
    TB_info_58      : in std_logic;
    TB_info_59      : in std_logic;

    TB_info_60      : in std_logic;
    TB_info_61      : in std_logic;
    TB_info_62      : in std_logic;
    TB_info_63      : in std_logic;


    read_v          : out std_logic;
    write_v         : out std_logic;
    data_out_v      : out std_logic_vector(31 downto 0);
    address_v       : out std_logic_vector(9 downto 0);
    state_v         : out std_logic;

    en_TB_mem       : out std_logic;
    rw_TB_mem       : out std_logic;
    data_out_TB_mem     : out std_logic_vector(31 downto 0);
    address_TB_mem      : out std_logic_vector(9 downto 0);
    clk_TB_mem      : out std_logic;

    soft_bit_coded      : out std_logic_vector(31 downto 0);
    enable_sb_decoder   : out std_logic;

--from PE
    enable_PE       : out std_logic;
    step_PE         : out integer range 0 to 31;
    flush_PE        : out std_logic;
    rotate_counter  : out integer range 0 to 7;
    last_PM_max     : out integer range -512 to 511;
    k5              : out std_logic;
    k7              : out std_logic;


    reset_PM_storage  : out std_logic;

    enable_find_new_index   : out std_logic;
    tb_data                 : out std_logic_vector(63 downto 0);
    current_index           : out integer range 0 to 63;

    TA              : out  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TD              : out  STD_LOGIC_VECTOR( 31 downto 0) ;
    TEZ             : out  STD_LOGIC ;
    TWZ             : out  STD_LOGIC ;

    WRENZ           : out  STD_LOGIC_VECTOR( 31 downto 0) ;
    SCAN            : out  STD_LOGIC ;
    TM              : out  STD_LOGIC ;
    ATPGM           : out  STD_LOGIC ;
    CSCAN           : out  STD_LOGIC ;

    SI              : out  STD_LOGIC ;

    DFTREAD         : out  STD_LOGIC

);
end component;
```

```
signal      A_TB_mem    : STD_LOGIC_VECTOR( 9 downto 0)  ;
signal      TA_TB_mem   : STD_LOGIC_VECTOR( 9 downto 0)  ;
signal      D_TB_mem    : STD_LOGIC_VECTOR( 31 downto 0) ;
signal      TD_TB_mem   : STD_LOGIC_VECTOR( 31 downto 0) ;
signal      clk_TB_mem  : STD_LOGIC ;
signal      EZ_TB_mem      : STD_LOGIC ;
signal      TEZ_TB_mem     : STD_LOGIC ;
signal      WZ_TB_mem      : STD_LOGIC ;
signal      TWZ_TB_mem     : STD_LOGIC ;
signal      WRENZ_TB_mem   : STD_LOGIC_VECTOR( 31 downto 0) ;
signal      SCAN_TB_mem    : STD_LOGIC ;
signal      TM_TB_mem      : STD_LOGIC ;
signal      ATPGM_TB_mem   : STD_LOGIC ;
signal      CSCAN_TB_mem   : STD_LOGIC ;
signal      SI_TB_mem      : STD_LOGIC ;
signal      DFTREAD_TB_mem : STD_LOGIC ;
signal      Q_TB_mem       : STD_LOGIC_VECTOR( 31 downto 0);
signal      SO_TB_mem      : std_logic;

signal  soft_bit_coded      : std_logic_vector(31 downto 0);
signal  enable_sb_decoder   : std_logic;
signal  sb0         :  std_logic_vector(5 downto 0);
signal  sb1         :  std_logic_vector(5 downto 0);
signal  sb2         :  std_logic_vector(5 downto 0);

signal  enable_find_new_index   : std_logic;
signal  tb_data         : std_logic_vector(63 downto 0);
signal  current_index       : integer range 0 to 63;
signal  decoded_bit : std_logic;
signal  new_index   : integer range 0 to 63;

signal      flush_PE            : std_logic;
signal      step_PE             : integer range 0 to 31;
signal      enable_PE           : std_logic;
signal      rotate_counter      : integer range 0 to 7;
signal      k5                  : std_logic;
signal      k7                  : std_logic;
signal      last_PM_max         : integer range -512 to 511;
signal      reset_PM_latch      : std_logic;

signal  TB00        : std_logic;
signal  TB01        : std_logic;
signal  TB02        : std_logic;
signal  TB03        : std_logic;
signal  TB04        : std_logic;
signal  TB05        : std_logic;
signal  TB06        : std_logic;
signal  TB07        : std_logic;
signal  TB08        : std_logic;
signal  TB09        : std_logic;
signal  TB10        : std_logic;
signal  TB11        : std_logic;
signal  TB12        : std_logic;
signal  TB13        : std_logic;
signal  TB14        : std_logic;
signal  TB15        : std_logic;
signal  TB16        : std_logic;
signal  TB17        : std_logic;
signal  TB18        : std_logic;
signal  TB19        : std_logic;
signal  TB20        : std_logic;
signal  TB21        : std_logic;
```

```
signal  TB22        : std_logic;
signal  TB23        : std_logic;
signal  TB24        : std_logic;
signal  TB25        : std_logic;
signal  TB26        : std_logic;
signal  TB27        : std_logic;
signal  TB28        : std_logic;
signal  TB29        : std_logic;
signal  TB30        : std_logic;
signal  TB31        : std_logic;
signal  TB32        : std_logic;
signal  TB33        : std_logic;
signal  TB34        : std_logic;
signal  TB35        : std_logic;
signal  TB36        : std_logic;
signal  TB37        : std_logic;
signal  TB38        : std_logic;
signal  TB39        : std_logic;
signal  TB40        : std_logic;
signal  TB41        : std_logic;
signal  TB42        : std_logic;
signal  TB43        : std_logic;
signal  TB44        : std_logic;
signal  TB45        : std_logic;
signal  TB46        : std_logic;
signal  TB47        : std_logic;
signal  TB48        : std_logic;
signal  TB49        : std_logic;
signal  TB50        : std_logic;
signal  TB51        : std_logic;
signal  TB52        : std_logic;
signal  TB53        : std_logic;
signal  TB54        : std_logic;
signal  TB55        : std_logic;
signal  TB56        : std_logic;
signal  TB57        : std_logic;
signal  TB58        : std_logic;
signal  TB59        : std_logic;
signal  TB60        : std_logic;
signal  TB61        : std_logic;
signal  TB62        : std_logic;
signal  TB63        : std_logic;

signal    PM_max          : integer range -512 to 511;
signal    index_max       : integer range 0 to 63;

begin

tb_memory : tb_ram port map(A_TB_mem, TA_TB_mem,D_TB_mem,TD_TB_mem,
clk_TB_mem, EZ_TB_mem,TEZ_TB_mem,WZ_TB_mem,TWZ_TB_mem,WRENZ_TB_mem,
SCAN_TB_mem,TM_TB_mem,ATPGM_TB_mem,CSCAN_TB_mem,SI_TB_mem,
DFTREAD_TB_mem,Q_TB_mem,SO_TB_mem);

sb_decoder: soft_bit_decoder port map ( soft_bit_coded,
enable_sb_decoder,soft_bit_no, poly_v,sb0,sb1,sb2);

update_index: find_new_index port map( enable_find_new_index, poly_v,
TB_data, current_index, decoded_bit, new_index);

pe_pipelined: PE_unit port map( clk,flush_PE,step_PE,enable_PE,
rotate_counter,k5,k7,threshold,last_PM_max,sb0,sb1,sb2,poly_v,
reset_PM_latch,TB00,TB01,TB02,TB03,TB04,TB05,TB06,TB07,TB08,TB09,
TB10,TB11,TB12,TB13,TB14,TB15,TB16,TB17,TB18,TB19,TB20,TB21,TB22,
```

```
TB23,TB24,TB25,TB26,TB27,TB28,TB29,TB30,TB31,TB32,TB33,TB34,TB35,
TB36,TB37,TB38,TB39,TB40,TB41,TB42,TB43,TB44,TB45,TB46,TB47,TB48,
TB49,TB50,TB51,TB52,TB53,TB54,TB55,TB56,TB57,TB58,TB59,TB60,TB61,
TB62,TB63,PM_max,index_max);

controller: viterbi_control_unit port map (reset,clk,run_v,data_in_v,
poly_v,block_size,Q_TB_mem,decoded_bit,new_index,index_max,PM_max,TB00,
TB01,TB02,TB03,TB04,TB05,TB06,TB07,TB08,TB09,TB10,TB11,TB12,TB13,TB14,
TB15,TB16,TB17,TB18,TB19,TB20,TB21,TB22,TB23,TB24,TB25,TB26,TB27,TB28,
TB29,TB30,TB31,TB32,TB33,TB34,TB35,TB36,TB37,TB38,TB39,TB40,TB41,TB42,
TB43,TB44,TB45,TB46,TB47,TB48,TB49,TB50,TB51,TB52,TB53,TB54,TB55,TB56,
TB57,TB58,TB59,TB60,TB61,TB62,TB63,read_v,write_v,data_out_v,address_v,
state_v,EZ_TB_mem,WZ_TB_mem,D_TB_mem,A_TB_mem,clk_TB_mem,soft_bit_coded,
enable_sb_decoder,enable_PE,step_PE,flush_PE,rotate_counter,last_PM_max,
k5,k7,reset_PM_latch,enable_find_new_index,tb_data,current_index,TA_TB_mem,
TD_TB_mem,TEZ_TB_mem,TWZ_TB_mem,WRENZ_TB_mem,SCAN_TB_mem,TM_TB_mem,
ATPGM_TB_mem,CSCAN_TB_mem,SI_TB_mem,DFTREAD_TB_mem);

end structural;
```

A.1.5 Viterbi control unit

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--controller of Viterbi unit

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity viterbi_control_unit is
port(
    reset           : in std_logic;
    clk         : in std_logic;

    run_v           : in std_logic;
    data_in_v       : in std_logic_vector(31 downto 0);
    poly_v          : in std_logic_vector (3 downto 0);
    block_size      : in integer range 0 to 612;
    data_in_TB_mem      : in std_logic_vector(31 downto 0);

    --trace back data
    decoded_bit     : in std_logic;
    new_index       : in integer range 0 to 63;


    --from max
    index_max       : in integer range 0 to 63;

    --from PE
    pm_max          : in integer range -512 to 511;

    -- from TB_update
    TB_info_00      : in std_logic;
    TB_info_01      : in std_logic;
    TB_info_02      : in std_logic;
    TB_info_03      : in std_logic;
    TB_info_04      : in std_logic;
```

```
    TB_info_05      : in std_logic;
    TB_info_06      : in std_logic;
    TB_info_07      : in std_logic;
    TB_info_08      : in std_logic;
    TB_info_09      : in std_logic;

    TB_info_10      : in std_logic;
    TB_info_11      : in std_logic;
    TB_info_12      : in std_logic;
    TB_info_13      : in std_logic;
    TB_info_14      : in std_logic;
    TB_info_15      : in std_logic;
    TB_info_16      : in std_logic;
    TB_info_17      : in std_logic;
    TB_info_18      : in std_logic;
    TB_info_19      : in std_logic;

    TB_info_20      : in std_logic;
    TB_info_21      : in std_logic;
    TB_info_22      : in std_logic;
    TB_info_23      : in std_logic;
    TB_info_24      : in std_logic;
    TB_info_25      : in std_logic;
    TB_info_26      : in std_logic;
    TB_info_27      : in std_logic;
    TB_info_28      : in std_logic;
    TB_info_29      : in std_logic;

    TB_info_30      : in std_logic;
    TB_info_31      : in std_logic;
    TB_info_32      : in std_logic;
    TB_info_33      : in std_logic;
    TB_info_34      : in std_logic;
    TB_info_35      : in std_logic;
    TB_info_36      : in std_logic;
    TB_info_37      : in std_logic;
    TB_info_38      : in std_logic;
    TB_info_39      : in std_logic;

    TB_info_40      : in std_logic;
    TB_info_41      : in std_logic;
    TB_info_42      : in std_logic;
    TB_info_43      : in std_logic;
    TB_info_44      : in std_logic;
    TB_info_45      : in std_logic;
    TB_info_46      : in std_logic;
    TB_info_47      : in std_logic;
    TB_info_48      : in std_logic;
    TB_info_49      : in std_logic;

    TB_info_50      : in std_logic;
    TB_info_51      : in std_logic;
    TB_info_52      : in std_logic;
    TB_info_53      : in std_logic;
    TB_info_54      : in std_logic;
    TB_info_55      : in std_logic;
    TB_info_56      : in std_logic;
    TB_info_57      : in std_logic;
    TB_info_58      : in std_logic;
    TB_info_59      : in std_logic;

    TB_info_60      : in std_logic;
    TB_info_61      : in std_logic;
```

```
    TB_info_62      : in std_logic;
    TB_info_63      : in std_logic;


    read_v          : out std_logic;
    write_v         : out std_logic;
    data_out_v      : out std_logic_vector(31 downto 0);
    address_v       : out std_logic_vector(9 downto 0);
    state_v         : out std_logic;

    en_TB_mem       : out std_logic;
    rw_TB_mem       : out std_logic;
    data_out_TB_mem     : out std_logic_vector(31 downto 0);
    address_TB_mem      : out std_logic_vector(9 downto 0);
    clk_TB_mem      : out std_logic;

    soft_bit_coded      : out std_logic_vector(31 downto 0);
    enable_sb_decoder   : out std_logic;

    enable_PE       : out std_logic;
    step_PE         : out integer range 0 to 31;
    flush_PE        : out std_logic;
    rotate_counter  : out integer range 0 to 7;
    last_pm_max     : out integer range -512 to 511;
    k5              : out std_logic;
    k7              : out std_logic;

    reset_PM_storage  : out std_logic;

    enable_find_new_index   : out std_logic;
    tb_data         : out std_logic_vector(63 downto 0);
    current_index       : out integer range 0 to 63;

    TA              : out  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TD              : out  STD_LOGIC_VECTOR( 31 downto 0) ;
    TEZ             : out  STD_LOGIC ;
    TWZ             : out  STD_LOGIC ;

    WRENZ           : out  STD_LOGIC_VECTOR( 31 downto 0) ;
    SCAN            : out  STD_LOGIC ;
    TM              : out  STD_LOGIC ;
    ATPGM           : out  STD_LOGIC ;
    CSCAN           : out  STD_LOGIC ;

    SI              : out  STD_LOGIC ;

    DFTREAD         : out  STD_LOGIC

);
end viterbi_control_unit;



architecture behavioral of viterbi_control_unit is


type state_type is (idle, read_soft_bit, soft_bit_decode, PE,
wait_PE, TB_update, read_TB_info, find_new_index, format_output,
write_decoded_data, reorder_output);

signal current_state : state_type;
signal next_state : state_type;
```

```
signal current_outer_counter : integer range 0 to 1023;
signal next_outer_counter : integer range 0 to 1023;

signal current_inner_counter : integer range 0 to 31;
signal next_inner_counter : integer range 0 to 31;

signal current_index_internal : integer range 0 to 63;
signal next_index_internal : integer range 0 to 63;
signal current_decoded_output : std_logic_vector(31 downto 0);

signal next_rotate_counter : integer range 0 to 7;
signal current_rotate_counter : integer range 0 to 7;

signal next_pm_max : integer range -512 to 511;
signal current_pm_max : integer range -512 to 511;

signal clk_TB_mem_en: std_logic;

signal enable_latch_decoded_bit: std_logic;
signal flush_PE_internal: std_logic;

begin

reset_PE_mem: process(clk)
begin
-- reset the pipeline registers

if clk'event and clk='1' then
flush_PE<=flush_PE_internal;
end if;

end process;

mem_clock_gate: process(clk, clk_TB_mem_en)

begin

--gate the TB memory clock
    clk_TB_mem<=not (clk or clk_TB_mem_en);
end process;

latch_decoded_bit: process(clk, reset)
begin
-- 32 bit buffer for trace back data
if reset='1' then
    current_decoded_output<="00000000000000000000000000000000";
elsif clk 'event and clk='1' then
    if enable_latch_decoded_bit='1' then
        current_decoded_output(current_inner_counter)<=decoded_bit;
    end if;
end if;
end process;

control_logic : process(run_v, data_in_v, poly_v, block_size, data_in_TB_mem,
pm_max, new_index,TB_info_00,TB_info_01,TB_info_02,TB_info_03,TB_info_04,
TB_info_05,TB_info_06,TB_info_07,TB_info_08,TB_info_09,TB_info_10,TB_info_11,
TB_info_12,TB_info_13,TB_info_14,TB_info_15,TB_info_16,TB_info_17,TB_info_18,
TB_info_19,TB_info_20,TB_info_21,TB_info_22,TB_info_23,TB_info_24,TB_info_25,
TB_info_26,TB_info_27,TB_info_28,TB_info_29,TB_info_30,TB_info_31,TB_info_32,
TB_info_33,TB_info_34,TB_info_35,TB_info_36,TB_info_37,TB_info_38,TB_info_39,
TB_info_40,TB_info_41,TB_info_42,TB_info_43,TB_info_44,TB_info_45,TB_info_46,
TB_info_47,TB_info_48,TB_info_49,TB_info_50,TB_info_51,TB_info_52,TB_info_53,
```

```
TB_info_54,TB_info_55,TB_info_56,TB_info_57,TB_info_58,TB_info_59,TB_info_60,
TB_info_61,TB_info_62,TB_info_63, current_outer_counter,current_index_internal,
current_decoded_output,current_state, current_inner_counter,current_rotate_counter,
current_pm_max,index_max)

variable PE_counter_vector: std_logic_vector(9 downto 0);
variable TB_counter_vector: std_logic_vector(9 downto 0);
variable block_size_vector: std_logic_vector(9 downto 0);

variable write_counter: integer range 0 to 31;
variable K5_int, K7_int: std_logic;

begin

block_size_vector:=conv_std_logic_vector(block_size, 10);

write_counter:=conv_integer(unsigned(block_size_vector(9 downto 5)));

-- disable memory test signals
TA <="0000000000";
TWZ <= '0';
TEZ<= '1';
WRENZ<="00000000000000000000000000000000";
TD<= "00000000000000000000000000000000";

DFTREAD<='0';
SCAN<='0';
TM<='0';
CSCAN<='0';
ATPGM<='0';

SI<='0';

if poly_v="0001" or poly_v="0010" or poly_v="0011"or poly_v="0111" or
poly_v="1000" or  poly_v="1001" or  poly_v="1010" then
     --      K=5;
     K5<='1';
     K5_int:='1';
     K7<='0';
     K7_int:='0';
else
     --      K=7;
     K5<='0';
     K5_int:='0';
     K7<='1';
     K7_int:='1';
end if;

if current_outer_counter = 0 then
-- reset path metric storage
    reset_PM_storage<='1' ;
else
    reset_PM_storage<='0' ;
end if;


current_index<=current_index_internal;

rotate_counter<=current_rotate_counter;
last_pm_max<=current_pm_max;

PE_counter_vector:=conv_std_logic_vector(current_outer_counter, 10);
```

```
if current_outer_counter = 0 then
TB_counter_vector:="0000000000";
else
TB_counter_vector:=conv_std_logic_vector(current_outer_counter+1, 10);
end if;

   case current_state is

    when idle =>

-- do nothing
        read_v<='0';
        write_v<='0';
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector((current_outer_counter),10);
        state_v<='0';

        en_TB_mem<='1';
        rw_TB_mem<='0';
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
        clk_TB_mem_en<='1';

        soft_bit_coded<="00000000000000000000000000000000";
        enable_sb_decoder<='0';


        enable_PE<='0';
        flush_PE_internal<='1';
        step_PE<=0;

        enable_find_new_index<='0';

        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";

        next_outer_counter<=current_outer_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<= '0';

        next_inner_counter<=0;
        next_rotate_counter<=0;
        next_pm_max<=-256;

        if run_v='0' then
            next_state<=current_state;
        else
            next_state<=read_soft_bit;
        end if;

    when read_soft_bit =>
--read buffer
        read_v<='1';
        write_v<='0';
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector((current_outer_counter),10);
        state_v<='1';

        en_TB_mem<='1';
        rw_TB_mem<='0';
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
        clk_TB_mem_en<='1';
```

```
        soft_bit_coded<="00000000000000000000000000000000";
        enable_sb_decoder<=’0’;

        enable_PE<=’0’;
        flush_PE_internal<=’1’;
        step_PE<=0;

        enable_find_new_index<=’0’;
        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";

        next_outer_counter<=current_outer_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<= ’0’;

        next_inner_counter<=current_inner_counter;
        next_rotate_counter<=current_rotate_counter;
        next_pm_max<=current_pm_max;

        next_state<=soft_bit_decode;

    when soft_bit_decode =>
-- decode soft-decision data
        read_v<=’0’;
        write_v<=’0’;
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector(current_outer_counter, 10);
        state_v<=’1’;

        en_TB_mem<=’1’;
        rw_TB_mem<=’0’;
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
        clk_TB_mem_en<=’1’;

        soft_bit_coded<=data_in_v;
        enable_sb_decoder<=’1’;

        enable_PE<=’0’;
        flush_PE_internal<=’0’;
        step_PE<=0;

        enable_find_new_index<=’0’;

        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";


        next_outer_counter<=current_outer_counter;
        next_inner_counter<=current_inner_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<=’0’;

        next_rotate_counter<=current_rotate_counter;
        next_pm_max<=current_pm_max;

        next_state<=PE;


    when PE=>

        read_v<=’0’;
        write_v<=’0’;
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector(current_outer_counter, 10);
```

```
        state_v<=’1’;

        en_TB_mem<=’1’;
        rw_TB_mem<=’0’;
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
        clk_TB_mem_en<=’1’;

        soft_bit_coded<=data_in_v;
        enable_sb_decoder<=’1’;

        enable_PE<=’1’;
        flush_PE_internal<=’0’;
        step_PE<=current_inner_counter;

        enable_find_new_index<=’0’;

        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";


        next_outer_counter<=current_outer_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<=’0’;

        next_rotate_counter<=current_rotate_counter;
        next_pm_max<=current_pm_max;

        if k5_int=’1’ and k7_int=’0’ then
-- 8 clock cycles of PE
            if current_inner_counter=7 then
                next_inner_counter<=0;
                next_state<=wait_PE;
            else
                next_inner_counter<=current_inner_counter+1;
                next_state<=PE;
            end if;
        else
-- 32 clock cycles of PE
            if current_inner_counter=31 then
                next_inner_counter<=0;
                next_state<=wait_PE;
            else
                next_inner_counter<=current_inner_counter+1;
                next_state<=PE;
            end if;
        end if;


    when wait_PE=>
-- wait for a few clock cycles
        read_v<=’0’;
        write_v<=’0’;
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector(current_outer_counter, 10);
        state_v<=’1’;

        en_TB_mem<=’1’;
        rw_TB_mem<=’0’;
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
        clk_TB_mem_en<=’1’;

        soft_bit_coded<=data_in_v;
```

```
        enable_sb_decoder<='1';

        enable_PE<='0';
        flush_PE_internal<='0';
        step_PE<=0;


        enable_find_new_index<='0';

        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";

        next_outer_counter<=current_outer_counter;
        next_inner_counter<=current_inner_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<= '0';

        next_rotate_counter<=current_rotate_counter;
        next_pm_max<=current_pm_max;

        if current_inner_counter=3 then
            next_inner_counter<=0;
            next_state<=TB_update;
        else
            next_inner_counter<=current_inner_counter+1;
            next_state<=wait_PE;
        end if;


    when TB_update =>
        if k5_int='0' and k7_int='1' then
            read_v<='0';
            write_v<='1';
-- decision bits stored in the input buffer
            data_out_v(0)<=TB_info_32;
            data_out_v(1)<=TB_info_33;
            data_out_v(2)<=TB_info_34;
            data_out_v(3)<=TB_info_35;
            data_out_v(4)<=TB_info_36;
            data_out_v(5)<=TB_info_37;
            data_out_v(6)<=TB_info_38;
            data_out_v(7)<=TB_info_39;
            data_out_v(8)<=TB_info_40;
            data_out_v(9)<=TB_info_41;


            data_out_v(10)<=TB_info_42;
            data_out_v(11)<=TB_info_43;
            data_out_v(12)<=TB_info_44;
            data_out_v(13)<=TB_info_45;
            data_out_v(14)<=TB_info_46;
            data_out_v(15)<=TB_info_47;
            data_out_v(16)<=TB_info_48;
            data_out_v(17)<=TB_info_49;
            data_out_v(18)<=TB_info_50;
            data_out_v(19)<=TB_info_51;


            data_out_v(20)<=TB_info_52;
            data_out_v(21)<=TB_info_53;
            data_out_v(22)<=TB_info_54;
            data_out_v(23)<=TB_info_55;
            data_out_v(24)<=TB_info_56;
            data_out_v(25)<=TB_info_57;
```

```
            data_out_v(26)<=TB_info_58;
            data_out_v(27)<=TB_info_59;
            data_out_v(28)<=TB_info_60;
            data_out_v(29)<=TB_info_61;

            data_out_v(30)<=TB_info_62;
            data_out_v(31)<=TB_info_63;

            address_v<=conv_std_logic_vector(current_outer_counter, 10);
            state_v<='1';

        else
            read_v<='0';
            write_v<='0';
            data_out_v<="00000000000000000000000000000000";
            address_v<=conv_std_logic_vector(current_outer_counter,10);
            state_v<='1';

        end if;

        en_TB_mem<='0';
        rw_TB_mem<='0';
-- decision bit stored in the TB memory
        data_out_TB_mem(0)<=TB_info_00;
        data_out_TB_mem(1)<=TB_info_01;
        data_out_TB_mem(2)<=TB_info_02;
        data_out_TB_mem(3)<=TB_info_03;
        data_out_TB_mem(4)<=TB_info_04;
        data_out_TB_mem(5)<=TB_info_05;
        data_out_TB_mem(6)<=TB_info_06;
        data_out_TB_mem(7)<=TB_info_07;
        data_out_TB_mem(8)<=TB_info_08;
        data_out_TB_mem(9)<=TB_info_09;

        data_out_TB_mem(10)<=TB_info_10;
        data_out_TB_mem(11)<=TB_info_11;
        data_out_TB_mem(12)<=TB_info_12;
        data_out_TB_mem(13)<=TB_info_13;
        data_out_TB_mem(14)<=TB_info_14;
        data_out_TB_mem(15)<=TB_info_15;
        data_out_TB_mem(16)<=TB_info_16;
        data_out_TB_mem(17)<=TB_info_17;
        data_out_TB_mem(18)<=TB_info_18;
        data_out_TB_mem(19)<=TB_info_19;

        data_out_TB_mem(20)<=TB_info_20;
        data_out_TB_mem(21)<=TB_info_21;
        data_out_TB_mem(22)<=TB_info_22;
        data_out_TB_mem(23)<=TB_info_23;
        data_out_TB_mem(24)<=TB_info_24;
        data_out_TB_mem(25)<=TB_info_25;
        data_out_TB_mem(26)<=TB_info_26;
        data_out_TB_mem(27)<=TB_info_27;
        data_out_TB_mem(28)<=TB_info_28;
        data_out_TB_mem(29)<=TB_info_29;

        data_out_TB_mem(30)<=TB_info_30;
        data_out_TB_mem(31)<=TB_info_31;

        address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
        clk_TB_mem_en<= '0';

        soft_bit_coded<=data_in_v;
```

```
        enable_sb_decoder<='1';

        enable_PE<='0';
        flush_PE_internal<='0';
        step_PE<=0;

        enable_find_new_index<='0';
        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";

        enable_latch_decoded_bit<='0';

--update rotation counter

        if K5_int='1' and K7_int='0' then
            if current_rotate_counter=3 then
                next_rotate_counter<=0;
            else
                next_rotate_counter<=current_rotate_counter+1;
            end if;
        elsif K5_int='0' and K7_int='1' then
            if current_rotate_counter=5 then
                next_rotate_counter<=0;
            else
                next_rotate_counter<=current_rotate_counter+1;
            end if;
        else
            next_rotate_counter<=current_rotate_counter;
        end if;

        next_pm_max<=pm_max;

        if current_outer_counter>=(block_size-1) then
--continue PE
            next_outer_counter<=current_outer_counter;
            next_inner_counter<=conv_integer(unsigned(PE_counter_vector(4 downto 0)));
            next_index_internal<=index_max;
            next_state<=read_TB_info;
        else
--start Trace back
-- set up counters
            next_outer_counter<=current_outer_counter+1;
            next_inner_counter<=current_inner_counter;
            next_index_internal<=current_index_internal;
            next_state<=read_soft_bit;
        end if;

    when read_TB_info =>
        if current_index_internal>31 then
--enable input buffer
            read_v<='1';
            write_v<='0';
            data_out_v<="00000000000000000000000000000000";
            address_v<=conv_std_logic_vector(current_outer_counter,10);
            state_v<='1';
--disable TB memory
            en_TB_mem<='1';
            rw_TB_mem<='0';
            data_out_TB_mem<="00000000000000000000000000000000";
            address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
            clk_TB_mem_en<='1';


        else
```

```
--disable input buffer
            read_v<='0';
            write_v<='0';
            data_out_v<="00000000000000000000000000000000";
            address_v<=conv_std_logic_vector(current_outer_counter,10);
            state_v<='1';

--enable TB memory
            en_TB_mem<='0';
            rw_TB_mem<='1';
            data_out_TB_mem<="00000000000000000000000000000000";
            address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
            clk_TB_mem_en<='0';
        end if;


        soft_bit_coded<="00000000000000000000000000000000";
        enable_sb_decoder<='0';

        enable_PE<='0';
        flush_PE_internal<='0';
        step_PE<=0;

        enable_find_new_index<='0';
        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";

        next_outer_counter<=current_outer_counter;
        next_inner_counter<=current_inner_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<='0';

        next_rotate_counter<=0;
        next_pm_max<=-256;

        next_state<=find_new_index;

    when find_new_index =>
-- decode data


        read_v<='0';
        write_v<='0';
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector(current_outer_counter,10);
        state_v<='1';

        en_TB_mem<='1';
        rw_TB_mem<='0';
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=conv_std_logic_vector(current_outer_counter,10);
        clk_TB_mem_en<='1';

        soft_bit_coded<="00000000000000000000000000000000";
        enable_sb_decoder<='0';

        enable_PE<='0';
        flush_PE_internal<='0';
        step_PE<=0;


        enable_find_new_index<='1';
        tb_data<=data_in_v&data_in_TB_mem;
```

```
        next_outer_counter<=current_outer_counter;
        next_inner_counter<=current_inner_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<='0';

        next_rotate_counter<=0;
        next_pm_max<=-256;

        next_state<=format_output;

    when format_output=>
--latch the decoded data in the buffer

        read_v<='0';
        write_v<='0';
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector(current_outer_counter,10);
        state_v<='1';

        en_TB_mem<='1';
        rw_TB_mem<='0';
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=conv_std_logic_vector(current_outer_counter, 10);
        clk_TB_mem_en<='1';

        soft_bit_coded<="00000000000000000000000000000000";
        enable_sb_decoder<='0';

        enable_PE<='0';
        flush_PE_internal<='0';
        step_PE<=0;

        enable_find_new_index<='1';
        tb_data<=data_in_v&data_in_TB_mem;


        if current_outer_counter >0 then
            next_outer_counter<=current_outer_counter-1;
        else
            next_outer_counter<=1023;
        end if;

        if current_inner_counter = 0 then
            next_inner_counter<=31;
        else
            next_inner_counter<=current_inner_counter-1;
        end if;

        next_index_internal<=new_index;

        enable_latch_decoded_bit<='1';

        next_rotate_counter<=0;
        next_pm_max<=-256;

        if current_inner_counter=0 then
            next_state<=write_decoded_data;
        else
            next_state<=read_TB_info;
        end if;

    when write_decoded_data =>
```

```
--store decoded data in the TB memory


        read_v<='0';
        write_v<='0';
        data_out_v<="00000000000000000000000000000000";
        address_v<=conv_std_logic_vector(current_outer_counter,10);
        state_v<='1';

        en_TB_mem<='0';
        rw_TB_mem<='0';
        data_out_TB_mem<=current_decoded_output;
        address_TB_mem<=TB_counter_vector;
        clk_TB_mem_en<='0';

        soft_bit_coded<="00000000000000000000000000000000";
        enable_sb_decoder<='0';

        enable_PE<='0';
        flush_PE_internal<='0';
        step_PE<=0;

        enable_find_new_index<='0';
        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";

        next_outer_counter<=current_outer_counter;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<='0';

        next_rotate_counter<=0;
        next_pm_max<=-256;


        if current_outer_counter=1023 then
            next_state<=reorder_output;
            next_inner_counter<=0;
        else
            next_state<=read_TB_info;
            next_inner_counter<=current_inner_counter;
        end if;

    when reorder_output =>

-- move the decoded data from the TB memory to input buffer

        read_v<='0';
        write_v<='1';
        data_out_v<=data_in_TB_mem;
        address_v<=conv_std_logic_vector(current_inner_counter,10);
        state_v<='1';

        en_TB_mem<='0';
        rw_TB_mem<='1';
        data_out_TB_mem<="00000000000000000000000000000000";
        address_TB_mem<=(conv_std_logic_vector(current_inner_counter,5))&"00000";
        clk_TB_mem_en<='0';

        soft_bit_coded<="00000000000000000000000000000000";
        enable_sb_decoder<='0';

        enable_PE<='0';
        flush_PE_internal<='0';
        step_PE<=0;
```

```
        enable_find_new_index<='0';
        tb_data<="0000000000000000000000000000000000000000000000000000000000000000";

        next_outer_counter<=current_outer_counter;
        next_inner_counter<=current_inner_counter+1;
        next_index_internal<=current_index_internal;
        enable_latch_decoded_bit<='0';

        next_rotate_counter<=0;
        next_pm_max<=-256;


        if current_inner_counter>=write_counter then
            next_state<=idle;
        else
            next_state<=reorder_output;
        end if;


   end case;
end process;

state_transition: process (clk, reset)
begin

if reset='1' then
        current_outer_counter<=0;      --big H
        current_inner_counter<=0;       --small H
        current_index_internal<=0;      --other
        current_state<=idle;            --other
        current_rotate_counter<=0;      --small H
        current_pm_max<=-256;           --other
elsif clk'event and clk='1' then
        current_outer_counter<=next_outer_counter;-- after 1 ns;
        current_inner_counter<=next_inner_counter;-- after 1 ns;
        current_index_internal<=next_index_internal;-- after 1 ns;
        current_state<=next_state;-- after 1 ns;
        current_rotate_counter<=next_rotate_counter;
        current_pm_max<=next_pm_max;
end if;

end process;



end behavioral;
```

A.1.6 Trace back memory

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- capsulated memory
--trace back decision bit memory

library ieee;
library ti_custom;

use ieee.std_logic_1164.all;
```

```
use ti_custom.all;

entity tb_ram is
port(

    A   :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TA  :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    D   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    TD  :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

    CLK  :   in  STD_LOGIC ;

      EZ      :   in  STD_LOGIC ;
      TEZ     :   in  STD_LOGIC ;

      WZ      :   in  STD_LOGIC ;
      TWZ     :   in  STD_LOGIC ;

      WRENZ   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

      SCAN    :   in  STD_LOGIC ;
      TM      :   in  STD_LOGIC ;
      ATPGM   :   in  STD_LOGIC ;
      CSCAN   :   in  STD_LOGIC ;

      SI      :   in  STD_LOGIC ;


      DFTREAD :   in  STD_LOGIC ;

      Q       :   out STD_LOGIC_VECTOR( 31 downto 0);
      SO      :   out std_logic

);
end tb_ram;

architecture structural of tb_ram is

component GL00624032040  port(

    A   :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    TA  :   in  STD_LOGIC_VECTOR( 9 downto 0)  ;
    D   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;
    TD  :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

    CLK  :   in  STD_LOGIC ;

      EZ      :   in  STD_LOGIC ;
      TEZ     :   in  STD_LOGIC ;

      WZ      :   in  STD_LOGIC ;
      TWZ     :   in  STD_LOGIC ;

      WRENZ   :   in  STD_LOGIC_VECTOR( 31 downto 0) ;

      SCAN    :   in  STD_LOGIC ;
      TM      :   in  STD_LOGIC ;
      ATPGM   :   in  STD_LOGIC ;
      CSCAN   :   in  STD_LOGIC ;

      SI      :   in  STD_LOGIC ;
```

```
      DFTREAD :   in  STD_LOGIC ;

      Q       :   out STD_LOGIC_VECTOR( 31 downto 0);
      SO      :   out std_logic
    );


end component;

begin
ram_core: GL00624032040  port map (A,TA,D,TD,CLK,EZ,TEZ,WZ,TWZ,WRENZ,
SCAN,TM,ATPGM,CSCAN,SI,DFTREAD,Q,SO);
end structural;
```

A.1.7 Soft bit decoder

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--soft decision decoding and combination

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity soft_bit_decoder is
port(
    soft_bit_coded      : in    std_logic_vector (31 downto 0);
    enable_sb_decoder   : in    std_logic;
    soft_bit_no     : in    std_logic;
    poly_v  : in    std_logic_vector(3 downto 0);

    sb0_eff         : out   std_logic_vector(5 downto 0);
    sb1_eff         : out   std_logic_vector(5 downto 0);
    sb2_eff         : out   std_logic_vector(5 downto 0)
);
end soft_bit_decoder;



architecture behavioral of soft_bit_decoder is
begin

process(soft_bit_coded, enable_sb_decoder, soft_bit_no, poly_v)
variable    sb0 :    std_logic_vector(4 downto 0);
variable    sb1 :    std_logic_vector(4 downto 0);
variable    sb2 :    std_logic_vector(4 downto 0);
variable    sb3 :    std_logic_vector(4 downto 0);
variable    sb4 :    std_logic_vector(4 downto 0);
variable    sb5 :    std_logic_vector(4 downto 0);
variable    code_rate: integer range 0 to 7;
variable combined_sb0, combined_sb1, combined_sb2: integer range -32 to 31;
variable sb_to_combine_0, sb_to_combine_1, sb_to_combine_2, sb_to_combine_3,
sb_to_combine_4, sb_to_combine_5: integer range -16 to 15;

begin

  if enable_sb_decoder = '1' then
      if soft_bit_no = '0' then   --4-bit code
```

```
--decoding 4-bit data
          sb0:=soft_bit_coded(3)&soft_bit_coded(3 downto 0);
          sb1:=soft_bit_coded(7)&soft_bit_coded(7 downto 4);
          sb2:=soft_bit_coded(11)&soft_bit_coded(11 downto 8);
          sb3:=soft_bit_coded(15)&soft_bit_coded(15 downto 12);
          sb4:=soft_bit_coded(19)&soft_bit_coded(19 downto 16);
          sb5:=soft_bit_coded(23)&soft_bit_coded(23 downto 20);

      else
-- 5bit
          sb0:=soft_bit_coded( 4 downto  0);
          sb1:=soft_bit_coded( 9 downto  5);
          sb2:=soft_bit_coded(14 downto 10);
          sb3:=soft_bit_coded(19 downto 15);
          sb4:=soft_bit_coded(24 downto 20);
          sb5:=soft_bit_coded(29 downto 25);
      end if;
  else

  sb0:="00000";
  sb1:="00000";
  sb2:="00000";
  sb3:="00000";
  sb4:="00000";
  sb5:="00000";
  end if;

--calculate coderate

if poly_v="0001" then
    code_rate:=2;
elsif poly_v="0010" then
    code_rate:=3;
elsif poly_v="0011" then
    code_rate:=6;
elsif poly_v="0100" then
    code_rate:=3;
elsif poly_v="0101" then
    code_rate:=2;
elsif poly_v="0110" then
    code_rate:=3;
elsif poly_v="0111" then
    code_rate:=4;
elsif poly_v="1000" then
    code_rate:=2;
elsif poly_v="1001" then
    code_rate:=3;
elsif poly_v="1010" then
    code_rate:=5;
elsif poly_v="1011" then
    code_rate:=4;
elsif poly_v="1100" then
    code_rate:=5;
elsif poly_v="1101" then
    code_rate:=3;
else
    code_rate:=0;
end if;

--setup the input to the adders
if code_rate = 4 then
    sb_to_combine_0:=0;
    sb_to_combine_1:=0;
```

```
    sb_to_combine_2:=0;
    sb_to_combine_3:=0;
    sb_to_combine_4:=conv_integer(signed(sb2));
    sb_to_combine_5:=conv_integer(signed(sb3));
elsif code_rate = 5 then
    sb_to_combine_0:=conv_integer(signed(sb0));
    sb_to_combine_1:=conv_integer(signed(sb1));
    sb_to_combine_2:=0;
    sb_to_combine_3:=0;
    sb_to_combine_4:=conv_integer(signed(sb3));
    sb_to_combine_5:=conv_integer(signed(sb4));
elsif code_rate = 6 then
    sb_to_combine_0:=conv_integer(signed(sb0));
    sb_to_combine_1:=conv_integer(signed(sb3));
    sb_to_combine_2:=conv_integer(signed(sb1));
    sb_to_combine_3:=conv_integer(signed(sb4));
    sb_to_combine_4:=conv_integer(signed(sb2));
    sb_to_combine_5:=conv_integer(signed(sb5));
else
    sb_to_combine_0:=0;
    sb_to_combine_1:=0;
    sb_to_combine_2:=0;
    sb_to_combine_3:=0;
    sb_to_combine_4:=0;
    sb_to_combine_5:=0;
end if;

-- combine paths
combined_sb0:=sb_to_combine_0+sb_to_combine_1;
combined_sb1:=sb_to_combine_2+sb_to_combine_3;
combined_sb2:=sb_to_combine_4+sb_to_combine_5;


--select output
if code_rate = 6 then
    sb0_eff<=conv_std_logic_vector(combined_sb0,6);
    sb1_eff<=conv_std_logic_vector(combined_sb1,6);
    sb2_eff<=conv_std_logic_vector(combined_sb2,6);
elsif code_rate = 5 then
    sb0_eff<=conv_std_logic_vector(combined_sb0,6);
    sb1_eff<=conv_std_logic_vector(conv_integer(signed(sb2)),6);
    sb2_eff<=conv_std_logic_vector(combined_sb2,6);
elsif code_rate = 4 then
    sb0_eff<=conv_std_logic_vector(conv_integer(signed(sb0)),6);
    sb1_eff<=conv_std_logic_vector(conv_integer(signed(sb1)),6);
    sb2_eff<=conv_std_logic_vector(combined_sb2,6);
elsif code_rate = 3 then
    sb0_eff<=conv_std_logic_vector(conv_integer(signed(sb0)),6);
    sb1_eff<=conv_std_logic_vector(conv_integer(signed(sb1)),6);
    sb2_eff<=conv_std_logic_vector(conv_integer(signed(sb2)),6);
elsif code_rate = 2 then
    sb0_eff<=conv_std_logic_vector(conv_integer(signed(sb0)),6);
    sb1_eff<=conv_std_logic_vector(conv_integer(signed(sb1)),6);
    sb2_eff<=conv_std_logic_vector(0,6);
else
    sb0_eff<=conv_std_logic_vector(0,6);
    sb1_eff<=conv_std_logic_vector(0,6);
    sb2_eff<=conv_std_logic_vector(0,6);
end if;



end process;
```

```
end behavioral;
```

A.1.8 Find new index

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--trace back unit. produce the decoded bit


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity find_new_index is
port(
    enable_find_new_index   : in    std_logic;
    poly_v          : in    std_logic_vector(3 downto 0);
    TB_data         : in    std_logic_vector(63 downto 0);
    current_index       : in    integer range 0 to 63;

    decoded_bit     : out   std_logic;
    new_index       : out   integer range 0 to 63

);
end find_new_index;


architecture behavioral of find_new_index is
begin

main : process (enable_find_new_index, poly_v, TB_data, current_index)

variable K5: std_logic;
variable RSC : std_logic;
variable TB_decision_bit : std_logic;

variable current_index_vector, new_index_vector : std_logic_vector(5 downto 0);
variable J : std_logic_vector(4 downto 0);

variable G0RSC, G3RSC, G4RSC, G6RSC : std_logic;
variable b0in : std_logic;

begin
--cjech for RSC
if poly_v(3)='1' or  poly_v(2 downto 0)="111" then
    RSC:='1';
else
    RSC:='0';
end if;
--check for K
if poly_v="0001" or poly_v="0010" or poly_v="0011"or poly_v="0111" or  poly_v="1000" or
poly_v="1001" or  poly_v="1010" then
    K5:='1';
else
    K5:='0';
end if;
--find decision bit
```

```
if enable_find_new_index ='1' then
    current_index_vector := conv_std_logic_vector(current_index, 6);
else
    current_index_vector := "000000";
end if;

TB_decision_bit:=TB_data(current_index);

--find the next index
if K5='1' then
    new_index_vector:="00"&TB_decision_bit&current_index_vector(3 downto 1);
else
    new_index_vector:=TB_decision_bit&current_index_vector(5 downto 1);
end if;


new_index<=conv_integer(unsigned(new_index_vector));

--find J
J:=current_index_vector(5 downto 1);

--feedbacks
G0RSC:= '0' xor J(2);
G3RSC:= '0' xor J(2) xor J(1) xor J(0);
G4RSC:= '0' xor J(4) xor J(2) xor J(1);
G6RSC:= '0' xor J(3) xor J(2) xor J(1) xor J(0);


--input to the register chain
if poly_v = "0111" or poly_v = "1001" or poly_v="1010" then
    b0in:=G3RSC xor '0';
elsif poly_v = "1000" then
    b0in:=G0RSC xor '0';
elsif poly_v = "1011" or poly_v="1100" then
    b0in:=G6RSC xor '0';
elsif poly_v="1101" then
    b0in:=G4RSC xor '0';
else
    b0in:='0';
end if;

--decoded bit
if RSC = '0' then
    decoded_bit<=current_index_vector(0);
elsif   b0in='0' then
    decoded_bit<=current_index_vector(0) xor TB_decision_bit;
else
    decoded_bit<= not (current_index_vector(0) xor TB_decision_bit);
end if;

end process;

end behavioral;
```

A.1.9 Path extension unit

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--structure of the path extension unit
```

```
library ieee;
library ti_custom;

use ieee.std_logic_1164.all;
use ti_custom.all;

entity PE_unit is
port(

    clk     : in std_logic;
    flush           : in std_logic;

    step    : in integer range 0 to 31;
    enable  : in std_logic;

    rotate_counter  : in integer range 0 to 7;
    k5      : in std_logic;
    k7      : in std_logic;
    threshold   : in integer range 0 to 511;
    last_PM_max : in integer range -512 to 511;

    sb0     : in    std_logic_vector(5 downto 0);
    sb1     : in    std_logic_vector(5 downto 0);
    sb2     : in    std_logic_vector(5 downto 0);

    poly_v  : in    std_logic_vector(3 downto 0);

    reset_PM_latch  : in    std_logic;

    TB00_latched        : out   std_logic;
    TB01_latched        : out   std_logic;
    TB02_latched        : out   std_logic;
    TB03_latched        : out   std_logic;
    TB04_latched        : out   std_logic;
    TB05_latched        : out   std_logic;
    TB06_latched        : out   std_logic;
    TB07_latched        : out   std_logic;
    TB08_latched        : out   std_logic;
    TB09_latched        : out   std_logic;
    TB10_latched        : out   std_logic;
    TB11_latched        : out   std_logic;
    TB12_latched        : out   std_logic;
    TB13_latched        : out   std_logic;
    TB14_latched        : out   std_logic;
    TB15_latched        : out   std_logic;
    TB16_latched        : out   std_logic;
    TB17_latched        : out   std_logic;
    TB18_latched        : out   std_logic;
    TB19_latched        : out   std_logic;
    TB20_latched        : out   std_logic;
    TB21_latched        : out   std_logic;
    TB22_latched        : out   std_logic;
    TB23_latched        : out   std_logic;
    TB24_latched        : out   std_logic;
    TB25_latched        : out   std_logic;
    TB26_latched        : out   std_logic;
    TB27_latched        : out   std_logic;
    TB28_latched        : out   std_logic;
    TB29_latched        : out   std_logic;
    TB30_latched        : out   std_logic;
    TB31_latched        : out   std_logic;
    TB32_latched        : out   std_logic;
```

```
    TB33_latched        : out   std_logic;
    TB34_latched        : out   std_logic;
    TB35_latched        : out   std_logic;
    TB36_latched        : out   std_logic;
    TB37_latched        : out   std_logic;
    TB38_latched        : out   std_logic;
    TB39_latched        : out   std_logic;
    TB40_latched        : out   std_logic;
    TB41_latched        : out   std_logic;
    TB42_latched        : out   std_logic;
    TB43_latched        : out   std_logic;
    TB44_latched        : out   std_logic;
    TB45_latched        : out   std_logic;
    TB46_latched        : out   std_logic;
    TB47_latched        : out   std_logic;
    TB48_latched        : out   std_logic;
    TB49_latched        : out   std_logic;
    TB50_latched        : out   std_logic;
    TB51_latched        : out   std_logic;
    TB52_latched        : out   std_logic;
    TB53_latched        : out   std_logic;
    TB54_latched        : out   std_logic;
    TB55_latched        : out   std_logic;
    TB56_latched        : out   std_logic;
    TB57_latched        : out   std_logic;
    TB58_latched        : out   std_logic;
    TB59_latched        : out   std_logic;
    TB60_latched        : out   std_logic;
    TB61_latched        : out   std_logic;
    TB62_latched        : out   std_logic;
    TB63_latched        : out   std_logic;

    PM_max_out          : out   integer range -512 to 511;
    index_max_out       : out   integer range 0 to 63

);
end PE_unit;


architecture structural of PE_unit is
--components

component bm_ram1
   port(
          AR   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
          AW   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
          D    :       in    STD_LOGIC_VECTOR(9 downto 0)
                         := (9 downto 0 => 'U');
          WCLK :       in    STD_LOGIC := 'U';
          WEN  :       in    STD_LOGIC := 'U';
          Q    :       out   STD_LOGIC_VECTOR(9 downto 0)
      );

end component;

component bm_ram2
   port(
          AR   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
          AW   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
```

```
          D    :       in    STD_LOGIC_VECTOR(9 downto 0)
                         := (9 downto 0 => 'U');
          WCLK :       in    STD_LOGIC := 'U';
          WEN  :       in    STD_LOGIC := 'U';
          Q    :       out   STD_LOGIC_VECTOR(9 downto 0)
      );

end component;

component read_PM
port(
    step                : in    integer range 0 to 31;
    rotate_counter      : in    integer range 0 to 7;
    k5                  : in std_logic;
    k7                  : in std_logic;

    Q_even              : in  std_logic_vector (9 downto 0);
    Q_odd               : in  std_logic_vector (9 downto 0);

    AR_even             : out std_logic_vector (4 downto 0);
    AR_odd              : out std_logic_vector (4 downto 0);

    PM_up               : out integer range -512 to 511;
    PM_down             : out integer range -512 to 511
);
end  component;

component Path_prune
port(

reset_PM_storage    : in std_logic;
enable              : in std_logic;
PM_up_in            : in integer range -512 to 511;
PM_down_in          : in integer range -512 to 511;
threshold           : in    integer range 0 to 511;
last_PM_max         : in    integer range -512 to 511;

PM_up_out           : out integer range -512 to 511;
PM_down_out         : out integer range -512 to 511;
PM_up_invalid       :out std_logic;
PM_down_invalid     :out std_logic
);
end  component;

component ACS_cc
port
(
    poly_v  : in    std_logic_vector(3 downto 0);
    J_in       : in    integer range 0 to 31;

    code_rate: out integer range 0 to 7;
    b0_shift_in : out std_logic;

    c0_eff: out std_logic;
    c1_eff: out std_logic;
    c2_eff: out std_logic

);
end component;

component pip1
port
(
```

```
    clk             : in std_logic;
    flush           : in std_logic;

    step_in        : in integer range 0 to 31;
    enable_in      : in std_logic;

    PM_up_in                : in integer range -512 to 511;
    PM_down_in              : in integer range -512 to 511;
    PM_up_invalid_in        : in std_logic;
    PM_down_invalid_in      : in std_logic;

    code_rate_in    : in integer range 0 to 7;
    b0_shift_in     : in std_logic;
    c0_eff_in       : in std_logic;
    c1_eff_in       : in std_logic;
    c2_eff_in       : in std_logic;


    step_out        : out integer range 0 to 31;
    enable_out      : out std_logic;

    PM_up_out               : out integer range -512 to 511;
    PM_down_out             : out integer range -512 to 511;
    PM_up_invalid_out       : out std_logic;
    PM_down_invalid_out     : out std_logic;

    code_rate_out    : out integer range 0 to 7;
    b0_shift_out     : out std_logic;
    c0_eff_out       : out std_logic;
    c1_eff_out       : out std_logic;
    c2_eff_out       : out std_logic
);
end component;

component ACS_bm
port
(
    sb0_eff_in     : in    std_logic_vector(5 downto 0);
    sb1_eff_in     : in    std_logic_vector(5 downto 0);
    sb2_eff_in     : in    std_logic_vector(5 downto 0);

    code_rate: in integer range 0 to 7;

    c0_eff: in std_logic;
    c1_eff: in std_logic;
    c2_eff: in std_logic;

    branch_metric_vector : out std_logic_vector(7 downto 0)
);
end component;

component ACS_pm
port
(
    PM_in_up            : in    integer range -512 to 511;
    PM_in_down          : in    integer range -512 to 511;
    PM_up_invalid       : in    std_logic;
    PM_down_invalid     : in    std_logic;

    poly_v      : in    std_logic_vector(3 downto 0);
    branch_metric_vector : in std_logic_vector(7 downto 0);
    b0_shift_in     : in std_logic;
```

```
    PM_out_up   : out   integer range -512 to 511;
    PM_out_down : out   integer range -512 to 511;

    TB_info_up  : out   std_logic;
    TB_info_down    : out   std_logic
);
end component;

component TB_update
port
(
    enable_TB_update    : in    std_logic;

    step                : in    integer range 0 to 31;

    k5                      : in std_logic;
    k7                      : in std_logic;

    clk             : in std_logic;

    TB00      : in    std_logic;
    TB01      : in    std_logic;

    TB00_latched        : out   std_logic;
    TB01_latched        : out   std_logic;
    TB02_latched        : out   std_logic;
    TB03_latched        : out   std_logic;
    TB04_latched        : out   std_logic;
    TB05_latched        : out   std_logic;
    TB06_latched        : out   std_logic;
    TB07_latched        : out   std_logic;
    TB08_latched        : out   std_logic;
    TB09_latched        : out   std_logic;
    TB10_latched        : out   std_logic;
    TB11_latched        : out   std_logic;
    TB12_latched        : out   std_logic;
    TB13_latched        : out   std_logic;
    TB14_latched        : out   std_logic;
    TB15_latched        : out   std_logic;
    TB16_latched        : out   std_logic;
    TB17_latched        : out   std_logic;
    TB18_latched        : out   std_logic;
    TB19_latched        : out   std_logic;
    TB20_latched        : out   std_logic;
    TB21_latched        : out   std_logic;
    TB22_latched        : out   std_logic;
    TB23_latched        : out   std_logic;
    TB24_latched        : out   std_logic;
    TB25_latched        : out   std_logic;
    TB26_latched        : out   std_logic;
    TB27_latched        : out   std_logic;
    TB28_latched        : out   std_logic;
    TB29_latched        : out   std_logic;
    TB30_latched        : out   std_logic;
    TB31_latched        : out   std_logic;
    TB32_latched        : out   std_logic;
    TB33_latched        : out   std_logic;
    TB34_latched        : out   std_logic;
    TB35_latched        : out   std_logic;
    TB36_latched        : out   std_logic;
    TB37_latched        : out   std_logic;
    TB38_latched        : out   std_logic;
    TB39_latched        : out   std_logic;
```

```
    TB40_latched        : out   std_logic;
    TB41_latched        : out   std_logic;
    TB42_latched        : out   std_logic;
    TB43_latched        : out   std_logic;
    TB44_latched        : out   std_logic;
    TB45_latched        : out   std_logic;
    TB46_latched        : out   std_logic;
    TB47_latched        : out   std_logic;
    TB48_latched        : out   std_logic;
    TB49_latched        : out   std_logic;
    TB50_latched        : out   std_logic;
    TB51_latched        : out   std_logic;
    TB52_latched        : out   std_logic;
    TB53_latched        : out   std_logic;
    TB54_latched        : out   std_logic;
    TB55_latched        : out   std_logic;
    TB56_latched        : out   std_logic;
    TB57_latched        : out   std_logic;
    TB58_latched        : out   std_logic;
    TB59_latched        : out   std_logic;
    TB60_latched        : out   std_logic;
    TB61_latched        : out   std_logic;
    TB62_latched        : out   std_logic;
    TB63_latched        : out   std_logic

);
end component;

component pip2
port
(
    clk              : in std_logic;
    flush            : in std_logic;

    step_in         : in integer range 0 to 31;
    enable_in       : in std_logic;

    TB00_in      : in    std_logic;
    TB01_in      : in    std_logic;

    pm_up_in         : in integer range -512 to 511;
    pm_down_in         : in integer range -512 to 511;

    step_out         : out integer range 0 to 31;
    enable_out       : out std_logic;

    TB00_out      : out    std_logic;
    TB01_out      : out    std_logic;

    pm_up_out         : out integer range -512 to 511;
    pm_down_out         : out integer range -512 to 511
);
end component;

component max
port(

    ACS00_PM_up     : in    integer range -512 to 511;
    ACS00_PM_down       : in    integer range -512 to 511;

    current_PM_max      :in     integer range -512 to 511;
    current_index_max   :in     integer range 0 to 63;
```

```
    step                : in    integer range 0 to 31;

    k5                      : in std_logic;
    k7                      : in std_logic;

    new_PM_max          : out   integer range -512 to 511;
    new_index_max       : out   integer range 0 to 63
);
end  component;

component write_PM
port(
    step                : in    integer range 0 to 31;
    rotate_counter      : in    integer range 0 to 7;
    k5                  : in std_logic;
    k7                  : in std_logic;

    PM_up               : in integer range -512 to 511;
    PM_down             : in integer range -512 to 511;

    D_even              : out std_logic_vector (9 downto 0);
    D_odd               : out std_logic_vector (9 downto 0);

    AW_even             : out std_logic_vector (4 downto 0);
    AW_odd              : out std_logic_vector (4 downto 0)


);
end component;

component pip3
port
(
    clk             : in std_logic;
    flush           : in std_logic;

    enable_in      : in std_logic;

    PM_max_in          : in   integer range -512 to 511;
    index_max_in       : in   integer range 0 to 63;

    PM_max_out          : out   integer range -512 to 511;
    index_max_out       : out   integer range 0 to 63
);
end  component;

--signal

signal      AR_even   :       STD_LOGIC_VECTOR(4 downto 0);
signal      AW_even   :       STD_LOGIC_VECTOR(4 downto 0);
signal      D_even    :       STD_LOGIC_VECTOR(9 downto 0);
signal      Q_even    :       STD_LOGIC_VECTOR(9 downto 0);

signal      AR_odd   :       STD_LOGIC_VECTOR(4 downto 0);
signal      AW_odd   :       STD_LOGIC_VECTOR(4 downto 0);
signal      D_odd    :       STD_LOGIC_VECTOR(9 downto 0);
signal      Q_odd    :       STD_LOGIC_VECTOR(9 downto 0);

signal      WCLK_TB_mem :       STD_LOGIC ;
signal      WEN_TB_mem  :       STD_LOGIC ;

signal      PM_up_read   :   integer range -512 to 511;
signal      PM_down_read :   integer range -512 to 511;
```

```
signal      PM_up_pruned   :   integer range -512 to 511;
signal      PM_down_pruned :   integer range -512 to 511;

signal      PM_up_invalid :     std_logic;
signal      PM_down_invalid :     std_logic;

signal    code_rate: integer range 0 to 7;
signal    b0_shift_in : std_logic;

signal    c0_eff: std_logic;
signal    c1_eff: std_logic;
signal    c2_eff: std_logic;

signal      PM_up_pruned_s2   :   integer range -512 to 511;
signal      PM_down_pruned_s2 :   integer range -512 to 511;

signal      PM_up_invalid_s2 :     std_logic;
signal      PM_down_invalid_s2 :     std_logic;

signal    code_rate_s2: integer range 0 to 7;
signal    b0_shift_in_s2 : std_logic;

signal    c0_eff_s2: std_logic;
signal    c1_eff_s2: std_logic;
signal    c2_eff_s2: std_logic;

signal  enable_s2: std_logic;
signal  step_s2: integer range 0 to 31;

signal  branch_metric_vector : std_logic_vector(7 downto 0);

signal      PM_up_merged   :   integer range -512 to 511;
signal      PM_down_merged :   integer range -512 to 511;

signal  TB_info_up_s2 :    std_logic;
signal  TB_info_down_s2 :    std_logic;

signal  TB_info_up_s3 :    std_logic;
signal  TB_info_down_s3 :    std_logic;

signal  enable_s3: std_logic;
signal  step_s3: integer range 0 to 31;

signal      PM_up_merged_s3   :   integer range -512 to 511;
signal      PM_down_merged_s3 :   integer range -512 to 511;

signal current_PM_max      :     integer range -512 to 511;
signal current_index_max   :     integer range 0 to 63;

signal new_PM_max      :     integer range -512 to 511;
signal new_index_max   :     integer range 0 to 63;

begin

even_mem: bm_ram1 port map (AR_even,AW_even,D_even,WCLK_TB_mem,WEN_TB_mem,Q_even);

odd_mem: bm_ram2 port map (AR_odd,AW_odd,D_odd,WCLK_TB_mem,WEN_TB_mem,Q_odd);

pm_fetch: read_PM port map(step,rotate_counter,k5,k7,Q_even,Q_odd,AR_even,AR_odd,
PM_up_read,PM_down_read);

t_unit: Path_prune port map(reset_PM_latch,enable,PM_up_read,PM_down_read,threshold,
```

```
last_PM_max,PM_up_pruned,PM_down_pruned,PM_up_invalid,PM_down_invalid);

encoder: ACS_cc port map(poly_v,step,code_rate,b0_shift_in,c0_eff,c1_eff,c2_eff);

pipeline1 : pip1 port map (clk,flush,step,enable,PM_up_pruned,PM_down_pruned,
PM_up_invalid,PM_down_invalid,code_rate, b0_shift_in, c0_eff, c1_eff, c2_eff,
step_s2,enable_s2,PM_up_pruned_s2,PM_down_pruned_s2,PM_up_invalid_s2,
PM_down_invalid_s2, code_rate_s2, b0_shift_in_s2, c0_eff_s2, c1_eff_s2, c2_eff_s2);

branch_metric: ACS_bm port map(sb0,sb1,sb2,code_rate_s2,c0_eff_s2,c1_eff_s2,c2_eff_s2,
branch_metric_vector);

path_merge: ACS_pm port map( PM_up_pruned_s2,PM_down_pruned_s2,PM_up_invalid_s2,
PM_down_invalid_s2,poly_v,branch_metric_vector,b0_shift_in_s2,PM_up_merged,
PM_down_merged,TB_info_up_s2,TB_info_down_s2);

tb_word_buffer: TB_update port map ( enable_s3,step_s3,k5,k7,clk,TB_info_up_s3,
TB_info_down_s3,TB00_latched,TB01_latched,TB02_latched,TB03_latched,TB04_latched,
TB05_latched,TB06_latched,TB07_latched, TB08_latched,TB09_latched,TB10_latched,
TB11_latched,TB12_latched,TB13_latched,TB14_latched,TB15_latched,TB16_latched,
TB17_latched,TB18_latched,TB19_latched,TB20_latched,TB21_latched,TB22_latched,
TB23_latched,TB24_latched,TB25_latched,TB26_latched,TB27_latched,TB28_latched,
TB29_latched,TB30_latched,TB31_latched,TB32_latched,TB33_latched,TB34_latched,
TB35_latched,TB36_latched,TB37_latched,TB38_latched,TB39_latched,TB40_latched,
TB41_latched,TB42_latched,TB43_latched,TB44_latched,TB45_latched,TB46_latched,
TB47_latched,TB48_latched,TB49_latched,TB50_latched,TB51_latched,TB52_latched,
TB53_latched,TB54_latched,TB55_latched,TB56_latched,TB57_latched,TB58_latched,
TB59_latched,TB60_latched,TB61_latched,TB62_latched,TB63_latched);

pipeline2 : pip2 port map (clk,flush,step_s2,enable_s2,TB_info_up_s2,
TB_info_down_s2,PM_up_merged,PM_down_merged, step_s3,enable_s3,TB_info_up_s3,
TB_info_down_s3,PM_up_merged_s3,PM_down_merged_s3);

find_max : max port map(PM_up_merged_s3,PM_down_merged_s3,current_PM_max,
current_index_max,step_s3,k5,k7,new_PM_max,new_index_max);

pm_store: write_PM port map(step_s3,rotate_counter,k5,k7, PM_up_merged_s3,
PM_down_merged_s3,D_even,D_odd,AW_even,AW_odd);

pipeline3: pip3 port map(clk,flush, enable_s3, new_PM_max,new_index_max,
current_PM_max,current_index_max);

WCLK_TB_mem<=enable_s3 and clk;
WEN_TB_mem<=enable_s3;

PM_max_out<=current_PM_max;
index_max_out<=current_index_max;


end structural;
```

A.1.10 Even memory

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--capsulated memory
-- path metric memory

library ieee;
```

```
library ti_custom;

use ieee.std_logic_1164.all;
use ti_custom.all;

entity bm_ram1 is
port(
          AR   :       in    STD_LOGIC_VECTOR(4 downto 0);
          AW   :       in    STD_LOGIC_VECTOR(4 downto 0);
          D    :       in    STD_LOGIC_VECTOR(9 downto 0);
          WCLK :       in    STD_LOGIC;
          WEN  :       in    STD_LOGIC;
          Q    :       out   STD_LOGIC_VECTOR(9 downto 0)
);
end bm_ram1;

architecture structural of bm_ram1 is
component MG00032010020
   port(
          AR   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
          AW   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
          D    :       in    STD_LOGIC_VECTOR(9 downto 0)
                         := (9 downto 0 => 'U');
          WCLK :       in    STD_LOGIC := 'U';
          WEN  :       in    STD_LOGIC := 'U';
          Q    :       out   STD_LOGIC_VECTOR(9 downto 0)
      );

end component;

begin
ram_core: MG00032010020 port map( AR,AW, D,WCLK,WEN,Q);


end structural;
```

A.1.11 Odd memory

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- capsulated memory
--path metric memory

library ieee;
library ti_custom;

use ieee.std_logic_1164.all;
use ti_custom.all;

entity bm_ram2 is
port(
          AR   :       in    STD_LOGIC_VECTOR(4 downto 0);
          AW   :       in    STD_LOGIC_VECTOR(4 downto 0);
          D    :       in    STD_LOGIC_VECTOR(9 downto 0);
          WCLK :       in    STD_LOGIC;
          WEN  :       in    STD_LOGIC;
          Q    :       out   STD_LOGIC_VECTOR(9 downto 0)
```

```
);
end bm_ram2;

architecture structural of bm_ram2 is
component MG00032010020
   port(
          AR   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
          AW   :       in    STD_LOGIC_VECTOR(4 downto 0)
                         := (4 downto 0 => 'U');
          D    :       in    STD_LOGIC_VECTOR(9 downto 0)
                         := (9 downto 0 => 'U');
          WCLK :       in    STD_LOGIC := 'U';
          WEN  :       in    STD_LOGIC := 'U';
          Q    :       out   STD_LOGIC_VECTOR(9 downto 0)
      );

end component;



begin

ram_core: MG00032010020 port map( AR,AW, D,WCLK,WEN,Q);


end structural;
```

A.1.12 Read PM

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- read address calculation for path metric storage

library ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity read_PM is
port(
    step                : in    integer range 0 to 31;
    rotate_counter      : in    integer range 0 to 7;
    k5                  : in std_logic;
    k7                  : in std_logic;

    Q_even              : in  std_logic_vector (9 downto 0);
    Q_odd               : in  std_logic_vector (9 downto 0);

    AR_even             : out std_logic_vector (4 downto 0);
    AR_odd              : out std_logic_vector (4 downto 0);

    PM_up               : out integer range -512 to 511;
    PM_down             : out integer range -512 to 511
);
end read_PM;

architecture behavioral of read_PM is
begin
```

```
main: process (step, rotate_counter, k5, k7, Q_even, Q_odd)

variable step_vector : std_logic_vector (4 downto 0);
variable J_odd_parity: std_logic;
variable index_up, index_down: integer range 0 to 63;
variable index_up_vector, index_down_vector: std_logic_vector (5 downto 0);


begin



step_vector:=conv_std_logic_vector(step,5);
--check parity for path J
J_odd_parity:=step_vector(0) xor step_vector(1) xor step_vector(2)
xor step_vector(3) xor step_vector(4);

if k5='1' and k7='0' and step<=7 then
--rotate index
index_up:=conv_integer(unsigned
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(step,4),'0')
ror rotate_counter) ));

index_down:= conv_integer(unsigned
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(8+step,4),'0')
ror rotate_counter) ));


elsif k5='0' and k7='1' then
--rotate index
index_up:= conv_integer(unsigned
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(step,6),'0')
ror rotate_counter) ));

index_down:= conv_integer(unsigned
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(32+step,6),'0')
ror rotate_counter) ));

else
  index_up:=0;
  index_down:=0;

end if;



index_up_vector:=conv_std_logic_vector(index_up, 6);
index_down_vector:=conv_std_logic_vector(index_down, 6);

-- output address and get data
if J_odd_parity='1' then

    AR_odd<=index_up_vector(5 downto 1);
    AR_even<=index_down_vector(5 downto 1);
    PM_up<=conv_integer(signed(Q_odd));
    PM_down<=conv_integer(signed(Q_even));
else
    AR_even<=index_up_vector(5 downto 1);
    AR_odd<=index_down_vector(5 downto 1);
    PM_down<=conv_integer(signed(Q_odd));
    PM_up<=conv_integer(signed(Q_even));
end if;
```

```
end process;
end behavioral;
```

A.1.13 Path prune

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- prune the paths with the threshold

library ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity Path_prune is
port(

reset_PM_storage : in std_logic;
enable  : in std_logic;
PM_up_in            : in integer range -512 to 511;
PM_down_in          : in integer range -512 to 511;
threshold        : in    integer range 0 to 511;
last_PM_max      : in    integer range -512 to 511;

PM_up_out           : out integer range -512 to 511;
PM_down_out          : out integer range -512 to 511;
PM_up_invalid       :out std_logic;
PM_down_invalid       :out std_logic
);
end Path_prune;

architecture behavioral of Path_prune is
begin

main: process (reset_PM_storage,PM_up_in,PM_down_in,threshold,last_PM_max, enable)

variable round_to : integer range -512 to 511 ;
variable rounded_threshold : integer range -512 to 511;

variable PM_up_in_rounded : integer range -512 to 511 ;
variable PM_down_in_rounded : integer range -512 to 511 ;

constant invalid_PM : integer:=-512;

begin
--normalized threshold
if enable='1' then
    round_to:=last_PM_max+256-threshold;
else
    round_to:=0;
end if;
--absolute threshold
rounded_threshold:=-256;



if reset_PM_storage='0' then
--normal function
```

```
if PM_up_in=invalid_PM then
--invalid input
   PM_up_out<=invalid_PM;
   PM_up_invalid<='1';
else
--normalize
   PM_up_in_rounded:=PM_up_in-round_to;

    if PM_up_in_rounded<rounded_threshold then
--pruned
        PM_up_out<=invalid_PM;
        PM_up_invalid<='1';
    else
--survive
        PM_up_out<=PM_up_in_rounded;
        PM_up_invalid<='0';
    end if;

end if;

if PM_down_in=invalid_PM then
--invalid input
   PM_down_out<=invalid_PM;
   PM_down_invalid<='1';
else
--normalize
   PM_down_in_rounded:=PM_down_in-round_to;

    if PM_down_in_rounded<rounded_threshold then
--prune
        PM_down_out<=invalid_PM;
        PM_down_invalid<='1';
    else
--survive
        PM_down_out<=PM_down_in_rounded;
        PM_down_invalid<='0';
    end if;

end if;

else
--pseudo reset
PM_up_out<=-256;
PM_down_out<=-256;
PM_up_invalid<='0';
PM_down_invalid<='0';


end if;
end process;
end behavioral;
```

A.1.14 ACS cc

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--local encoder

LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity ACS_cc is
port
(
    poly_v  : in    std_logic_vector(3 downto 0);
    J_in    : in    integer range 0 to 31;

    code_rate   : out integer range 0 to 7;
    b0_shift_in : out std_logic;

    c0_eff: out std_logic;
    c1_eff: out std_logic;
    c2_eff: out std_logic

);
end ACS_cc;


architecture behavioral of ACS_cc is

begin

main : process (poly_v,J_in)

variable J: std_logic_vector(4 downto 0);
variable RSC: std_logic;
variable G0RSC, G3RSC, G4RSC, G6RSC : std_logic;
variable b0in : std_logic;
variable b0G0, b0G1, b0G2, b0G3, b0G4, b0G5, b0G6, b0G7: std_logic;
variable b0O0: std_logic;
variable b0O1: std_logic;
variable b0O2: std_logic;
variable b0O3: std_logic;
variable b0O4: std_logic;
variable b0O5: std_logic;
variable code_rate_int: integer range 0 to 7;

begin


--find J
J:=conv_std_logic_vector(J_in, 5);

--check RSC
if poly_v(3)='1' or  poly_v(2 downto 0)="111" then
    RSC:='1';
else
    RSC:='0';
end if;

--compute all possible feedback value
G0RSC:= J(2);
G3RSC:= J(2) xor J(1) xor J(0);
G4RSC:= J(4) xor J(2) xor J(1);
G6RSC:= J(3) xor J(2) xor J(1) xor J(0);

--compute the shift-in bit of branch 0
if poly_v = "0111" or poly_v = "1001" or poly_v="1010" then
    b0in:=G3RSC;
elsif poly_v = "1000" then
```

```
    b0in:=G0RSC;
elsif poly_v = "1011" or poly_v="1100" then
    b0in:=G6RSC;
elsif poly_v="1101" then
    b0in:=G4RSC;
else
    b0in:='0';
end if;

b0_shift_in<=b0in;

--all the polynomials
b0G0:=b0in xor J(2);
b0G1:=b0in xor J(0) xor J(2);
b0G2:=b0in xor J(1);
b0G3:=b0in xor J(0) xor J(1) xor J(2);
b0G4:=b0in xor J(1) xor J(2) xor J(4);
b0G5:=b0in xor J(0) xor J(3);
b0G6:=b0in xor J(0) xor J(1) xor J(2) xor J(3);
b0G7:=b0in xor J(0) xor J(1) xor J(2);


-- find the real output codeword
if poly_v="0001" then
    code_rate_int:=2;
    b0O0:=b0G0;
    b0O1:=b0G1;
    b0O2:='0';
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="0010" then
    code_rate_int:=3;
    b0O0:=b0G1;
    b0O1:=b0G2;
    b0O2:=b0G3;
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="0011" then
    code_rate_int:=6;
    b0O0:=b0G1;
    b0O1:=b0G2;
    b0O2:=b0G3;
    b0O3:=b0G1;
    b0O4:=b0G2;
    b0O5:=b0G3;

elsif poly_v="0100" then
    code_rate_int:=3;
    b0O0:=b0G4;
    b0O1:=b0G5;
    b0O2:=b0G6;
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="0101" then
    code_rate_int:=2;
    b0O0:=b0G4;
    b0O1:=b0G6;
```

```
    b0O2:='0';
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="0110" then
    code_rate_int:=3;
    b0O0:=b0G4;
    b0O1:=b0G7;
    b0O2:=b0G5;
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="0111" then
    code_rate_int:=4;
    b0O0:=b0G1;
    b0O1:=b0G2;
    b0O2:='0';
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="1000" then
    code_rate_int:=2;
    b0O0:='0';
    b0O1:=b0G1;
    b0O2:='0';
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="1001" then
    code_rate_int:=3;
    b0O0:=b0G1;
    b0O1:=b0G2;
    b0O2:='0';
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="1010" then
    code_rate_int:=5;
    b0O0:=b0G1;
    b0O1:=b0G1;
    b0O2:=b0G2;
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="1011" then
    code_rate_int:=4;
    b0O0:=b0G4;
    b0O1:=b0G5;
    b0O2:='0';
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="1100" then
    code_rate_int:=5;
    b0O0:=b0G4;
    b0O1:=b0G4;
```

```
    b0O2:=b0G5;
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

elsif poly_v="1101" then
    code_rate_int:=3;
    b0O0:='0';
    b0O1:=b0G5;
    b0O2:=b0G6;
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

else
    code_rate_int:=0;
    b0O0:='0';
    b0O1:='0';
    b0O2:='0';
    b0O3:='0';
    b0O4:='0';
    b0O5:='0';

end if;

code_rate<=code_rate_int;

--combining the code.
if code_rate_int = 4 then
    c0_eff<=b0O0;
    c1_eff<=b0O1;
    c2_eff<=b0O2;
elsif code_rate_int = 5 then
    c0_eff<=b0O1;
    c1_eff<=b0O2;
    c2_eff<=b0O3;
elsif code_rate_int = 6 then
    c0_eff<=b0O0;
    c1_eff<=b0O1;
    c2_eff<=b0O2;
else
    c0_eff<=b0O0;
    c1_eff<=b0O1;
    c2_eff<=b0O2;
end if;



end process main;

end  behavioral;
```

A.1.15 Pipeline 1

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--pipeline register
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity pip1 is
port
(
    clk             : in std_logic;
    flush           : in std_logic;

    step_in        : in integer range 0 to 31;
    enable_in      : in std_logic;

    pm_up_in        : in integer range -512 to 511;
    pm_down_in        : in integer range -512 to 511;
    pm_up_invalid_in        : in std_logic;
    pm_down_invalid_in        : in std_logic;

    code_rate_in    : in integer range 0 to 7;
    b0_shift_in     : in std_logic;
    c0_eff_in       : in std_logic;
    c1_eff_in       : in std_logic;
    c2_eff_in       : in std_logic;


    step_out        : out integer range 0 to 31;
    enable_out      : out std_logic;

    pm_up_out        : out integer range -512 to 511;
    pm_down_out        : out integer range -512 to 511;
    pm_up_invalid_out        : out std_logic;
    pm_down_invalid_out        : out std_logic;

    code_rate_out    : out integer range 0 to 7;
    b0_shift_out     : out std_logic;
    c0_eff_out       : out std_logic;
    c1_eff_out       : out std_logic;
    c2_eff_out       : out std_logic
);
end pip1;

architecture behavioral of pip1 is
signal    code_rate_out_int    : integer range 0 to 7;
signal    pm_up_out_int       : integer range -512 to 511;
signal    pm_down_out_int     : integer range -512 to 511;
signal    step_out_int        : integer range 0 to 31;

begin

latch: process (clk,flush,enable_in,pm_up_invalid_in,pm_down_invalid_in)
variable invalid_path_counter: integer range 0 to 65536:=0;
variable double_invalid_path_counter: integer range 0 to 65536:=0;
begin

if flush='1' then
--reset
    step_out_int<=0;
    enable_out<='0';
    pm_up_out_int<=0;
    pm_down_out_int<=0;
    pm_up_invalid_out<='0';
    pm_down_invalid_out<='0';
    code_rate_out_int<=0;
    b0_shift_out<='0';
```

```
    c0_eff_out<='0';
    c1_eff_out<='0';
    c2_eff_out<='0';
elsif clk'event and clk='1' then
   enable_out<=enable_in;
   if enable_in = '1' then
        step_out_int<=step_in;
        pm_up_invalid_out<=pm_up_invalid_in;
        pm_down_invalid_out<=pm_down_invalid_in;
        code_rate_out_int<=code_rate_in;

    end if;
--isolate pm_out_up
    if enable_in = '1' and pm_up_invalid_in='0' then
        pm_up_out_int<=pm_up_in;
    end if;

--isolate PM_out_down
    if enable_in = '1' and pm_down_invalid_in='0' then
        pm_down_out_int<=pm_down_in;
    end if;

--ISOLATE ACS_bm
    if enable_in='1' and (pm_down_invalid_in='0' or pm_up_invalid_in='0')then
        c0_eff_out<=c0_eff_in;
        c1_eff_out<=c1_eff_in;
        c2_eff_out<=c2_eff_in;
        b0_shift_out<=b0_shift_in;
    end if;

--prune counterer
    if enable_in = '1' and pm_up_invalid_in='1' then
        invalid_path_counter:=invalid_path_counter+1;
    end if;

    if enable_in = '1' and pm_down_invalid_in='1' then
        invalid_path_counter:=invalid_path_counter+1;
    end if;

    if enable_in = '1' and pm_down_invalid_in='1' and pm_up_invalid_in='1' then
        double_invalid_path_counter:=double_invalid_path_counter+1;
    end if;

end if;

end process;

to_output: process(code_rate_out_int,pm_up_out_int,pm_down_out_int,step_out_int)
begin
--dummy process to reduce warning from DPFLI
code_rate_out<=code_rate_out_int;
pm_up_out<=pm_up_out_int;
pm_down_out<=pm_down_out_int;
step_out<=step_out_int;


end process;

end behavioral;
```

A.1.16 ACS BM

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--Branch metric calculation


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity ACS_bm is
port
(
    sb0_eff_in     : in    std_logic_vector(5 downto 0);
    sb1_eff_in     : in    std_logic_vector(5 downto 0);
    sb2_eff_in     : in    std_logic_vector(5 downto 0);

    code_rate: in integer range 0 to 7;

    c0_eff: in std_logic;
    c1_eff: in std_logic;
    c2_eff: in std_logic;

    branch_metric_vector : out std_logic_vector(7 downto 0)
);
end ACS_bm;


architecture behavioral of ACS_bm is


begin

main : process (sb0_eff_in, sb1_eff_in, sb2_eff_in,code_rate, c0_eff,c1_eff,c2_eff )

variable sb0_eff,sb1_eff,sb2_eff: integer range -32 to 31;
variable sb0_eff_credit, sb1_eff_credit, sb2_eff_credit: integer range -64 to 63;
variable sum_sb01: integer range -128 to 127;
variable sum_sb01_eff: integer range -128 to 127;
variable sum_sb012: integer range -128 to 127;
variable branch_metric : integer range -128 to 127;

begin

sb0_eff:=conv_integer(signed(sb0_eff_in));
sb1_eff:=conv_integer(signed(sb1_eff_in));
sb2_eff:=conv_integer(signed(sb2_eff_in));

--s0 credit calculation

if sb0_eff<0 and c0_eff='0' then
    sb0_eff_credit:=-sb0_eff;
elsif sb0_eff>=0 and c0_eff='1' then
    sb0_eff_credit:=sb0_eff;
elsif sb0_eff<0 and c0_eff='1' then
    sb0_eff_credit:=sb0_eff;
else
    sb0_eff_credit:=-sb0_eff;
```

```
end if;

--s1 credit calculation

if sb1_eff<0 and c1_eff='0' then
    sb1_eff_credit:=-sb1_eff;
elsif sb1_eff>=0 and c1_eff='1' then
    sb1_eff_credit:=sb1_eff;
elsif sb1_eff<0 and c1_eff='1' then
    sb1_eff_credit:=sb1_eff;
else
    sb1_eff_credit:=-sb1_eff;
end if;

--partial accumulation
sum_sb01:= sb0_eff_credit + sb1_eff_credit;

--operand isolation on adder
if code_rate>2 then
sum_sb01_eff:=sum_sb01;
else
sum_sb01_eff:=0;
end if;

--s2 credit calculation
if sb2_eff<0 and c2_eff='0' then
    sb2_eff_credit:=-sb2_eff;
elsif sb2_eff>=0 and c2_eff='1' then
    sb2_eff_credit:=sb2_eff;
elsif sb2_eff<0 and c2_eff='1' then
    sb2_eff_credit:=sb2_eff;
else
    sb2_eff_credit:=-sb2_eff;
end if;

-- accumulation over
sum_sb012:=sum_sb01_eff+sb2_eff_credit;

if code_rate=2 then
    branch_metric:=sum_sb01;
else
    branch_metric:=sum_sb012;
end if;

branch_metric_vector<=conv_std_logic_vector(branch_metric,8);

end process main;

end  behavioral;
```

A.1.17 ACS PM

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--path merge

LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
USE ieee.std_logic_arith.all;


entity ACS_pm is
port
(
    PM_in_up             : in    integer range -512 to 511;
    PM_in_down           : in    integer range -512 to 511;
    PM_up_invalid        : in    std_logic;
    PM_down_invalid      : in    std_logic;

    poly_v                 : in    std_logic_vector(3 downto 0);
    branch_metric_vector   : in std_logic_vector(7 downto 0);
    b0_shift_in            : in std_logic;

    PM_out_up   : out   integer range -512 to 511;
    PM_out_down : out   integer range -512 to 511;

    TB_info_up      : out   std_logic;
    TB_info_down    : out   std_logic
);
end ACS_pm;


architecture behavioral of ACS_pm is

begin

main : process (PM_in_up,PM_in_down,poly_v,branch_metric_vector,
b0_shift_in, PM_up_invalid,PM_down_invalid)

variable RSC: std_logic;
variable branch_metric : integer range -128 to 127;
variable PMb0, PMb1, PMb2, PMb3: integer range -512 to 511;
variable PM_out_1, PM_out_2: integer range -512 to 511;
variable TB_info_1, TB_info_2: std_logic;

begin

branch_metric:=conv_integer(signed(branch_metric_vector));

--check for RSC

if poly_v(3)='1' or  poly_v(2 downto 0)="111" then
    RSC:='1';
else
    RSC:='0';
end if;

PMb0:=PM_in_up+branch_metric;
PMb1:=PM_in_up-branch_metric;

PMb2:=PM_in_down-branch_metric;
PMb3:=PM_in_down+branch_metric;

if PM_up_invalid='0' and PM_down_invalid='0' then
--both valid

--merge
if PMb0 >PMb2 then
    PM_out_1:=PMb0;
    TB_info_1:='0';
else
```

```
    PM_out_1:=PMb2;
    TB_info_1:='1';
end if;

if PMb1 >PMb3 then
    PM_out_2:=PMb1;
    TB_info_2:='0';
else
    PM_out_2:=PMb3;
    TB_info_2:='1';
end if;

--one input invalud
elsif PM_up_invalid='0' and PM_down_invalid = '1' then
    PM_out_1:=PMb0;
    PM_out_2:=PMb1;
    TB_info_1:='0';
    TB_info_2:='0';
elsif PM_up_invalid='1' and PM_down_invalid = '0' then
    PM_out_1:=PMb2;
    PM_out_2:=PMb3;
    TB_info_1:='1';
    TB_info_2:='1';
else
    PM_out_1:=-512;
    PM_out_2:=-512;
    TB_info_1:='0';
    TB_info_2:='0';
end if;


-- switch the output for RSC code
if RSC='0' then
    PM_out_up<=PM_out_1;
    PM_out_down<=PM_out_2;
    TB_info_up<=TB_info_1;
    TB_info_down<=TB_info_2;
elsif b0_shift_in='0' then
    PM_out_up<=PM_out_1;
    PM_out_down<=PM_out_2;
    TB_info_up<=TB_info_1;
    TB_info_down<=TB_info_2;
else
    PM_out_up<=PM_out_2;
    PM_out_down<=PM_out_1;
    TB_info_up<=TB_info_2;
    TB_info_down<=TB_info_1;
end if;


end process main;

end  behavioral;
```

A.1.18 Pipeline 2

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03
```

```
--pipeline register

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity pip2 is
port
(
    clk             : in std_logic;
    flush           : in std_logic;

    step_in        : in integer range 0 to 31;
    enable_in      : in std_logic;

    TB00_in      : in    std_logic;
    TB01_in      : in    std_logic;

    pm_up_in        : in integer range -512 to 511;
    pm_down_in        : in integer range -512 to 511;

    step_out        : out integer range 0 to 31;
    enable_out      : out std_logic;

    TB00_out      : out    std_logic;
    TB01_out      : out    std_logic;

    pm_up_out        : out integer range -512 to 511;
    pm_down_out        : out integer range -512 to 511
);
end pip2;

architecture behavioral of pip2 is
signal     pm_up_out_int        : integer range -512 to 511;
signal     pm_down_out_int      : integer range -512 to 511;
signal     step_out_int         : integer range 0 to 31;
begin

latch: process (clk,flush,enable_in)


begin

if flush='1' then
--reset
    step_out_int<=0;
    enable_out<='0';
    pm_up_out_int<=0;
    pm_down_out_int<=0;
    TB00_out<='0';
    TB01_out<='0';
elsif clk'event and clk='1' then
   enable_out<=enable_in;
   if enable_in = '1' then
--update

        step_out_int<=step_in;
        pm_up_out_int<=pm_up_in;
        pm_down_out_int<=pm_down_in;
        TB00_out<=TB00_in;
        TB01_out<=TB01_in;
   end if;
end if;
```

```
end process;

to_output: process(pm_up_out_int,pm_down_out_int,step_out_int)
begin

--dummy signal for DPFLI
pm_up_out<=pm_up_out_int;
pm_down_out<=pm_down_out_int;
step_out<=step_out_int;

end process;
end behavioral;
```

A.1.19 TB update

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--temporary storage for 64-bit data

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity TB_update is
port
(
    enable_TB_update    : in    std_logic;

    step                : in    integer range 0 to 31;

    k5                      : in std_logic;--redundent
    k7                      : in std_logic;--redundent

    clk             : in std_logic;

    TB00      : in    std_logic;
    TB01      : in    std_logic;

    TB00_latched        : out   std_logic;
    TB01_latched        : out   std_logic;
    TB02_latched        : out   std_logic;
    TB03_latched        : out   std_logic;
    TB04_latched        : out   std_logic;
    TB05_latched        : out   std_logic;
    TB06_latched        : out   std_logic;
    TB07_latched        : out   std_logic;
    TB08_latched        : out   std_logic;
    TB09_latched        : out   std_logic;
    TB10_latched        : out   std_logic;
    TB11_latched        : out   std_logic;
    TB12_latched        : out   std_logic;
    TB13_latched        : out   std_logic;
    TB14_latched        : out   std_logic;
    TB15_latched        : out   std_logic;
    TB16_latched        : out   std_logic;
```

```
    TB17_latched        : out   std_logic;
    TB18_latched        : out   std_logic;
    TB19_latched        : out   std_logic;
    TB20_latched        : out   std_logic;
    TB21_latched        : out   std_logic;
    TB22_latched        : out   std_logic;
    TB23_latched        : out   std_logic;
    TB24_latched        : out   std_logic;
    TB25_latched        : out   std_logic;
    TB26_latched        : out   std_logic;
    TB27_latched        : out   std_logic;
    TB28_latched        : out   std_logic;
    TB29_latched        : out   std_logic;
    TB30_latched        : out   std_logic;
    TB31_latched        : out   std_logic;
    TB32_latched        : out   std_logic;
    TB33_latched        : out   std_logic;
    TB34_latched        : out   std_logic;
    TB35_latched        : out   std_logic;
    TB36_latched        : out   std_logic;
    TB37_latched        : out   std_logic;
    TB38_latched        : out   std_logic;
    TB39_latched        : out   std_logic;
    TB40_latched        : out   std_logic;
    TB41_latched        : out   std_logic;
    TB42_latched        : out   std_logic;
    TB43_latched        : out   std_logic;
    TB44_latched        : out   std_logic;
    TB45_latched        : out   std_logic;
    TB46_latched        : out   std_logic;
    TB47_latched        : out   std_logic;
    TB48_latched        : out   std_logic;
    TB49_latched        : out   std_logic;
    TB50_latched        : out   std_logic;
    TB51_latched        : out   std_logic;
    TB52_latched        : out   std_logic;
    TB53_latched        : out   std_logic;
    TB54_latched        : out   std_logic;
    TB55_latched        : out   std_logic;
    TB56_latched        : out   std_logic;
    TB57_latched        : out   std_logic;
    TB58_latched        : out   std_logic;
    TB59_latched        : out   std_logic;
    TB60_latched        : out   std_logic;
    TB61_latched        : out   std_logic;
    TB62_latched        : out   std_logic;
    TB63_latched        : out   std_logic

);
end TB_update;

architecture structural of TB_update is
component write_enable
port
(
        enable_TB_update    : in    std_logic;
        step              : in   integer range 0 to 31;
        clock_enable      : out  std_logic_vector(0 to 31)
);
end component;

component decision_bits
port
```

```
(
    clk                 : in  std_logic;
    clock_enable        : in  std_logic_vector(0 to 31);
    TB00       : in std_logic;
    TB01       : in std_logic;

    TB00_latched        : out   std_logic;
    TB01_latched        : out   std_logic;
    TB02_latched        : out   std_logic;
    TB03_latched        : out   std_logic;
    TB04_latched        : out   std_logic;
    TB05_latched        : out   std_logic;
    TB06_latched        : out   std_logic;
    TB07_latched        : out   std_logic;
    TB08_latched        : out   std_logic;
    TB09_latched        : out   std_logic;
    TB10_latched        : out   std_logic;
    TB11_latched        : out   std_logic;
    TB12_latched        : out   std_logic;
    TB13_latched        : out   std_logic;
    TB14_latched        : out   std_logic;
    TB15_latched        : out   std_logic;
    TB16_latched        : out   std_logic;
    TB17_latched        : out   std_logic;
    TB18_latched        : out   std_logic;
    TB19_latched        : out   std_logic;
    TB20_latched        : out   std_logic;
    TB21_latched        : out   std_logic;
    TB22_latched        : out   std_logic;
    TB23_latched        : out   std_logic;
    TB24_latched        : out   std_logic;
    TB25_latched        : out   std_logic;
    TB26_latched        : out   std_logic;
    TB27_latched        : out   std_logic;
    TB28_latched        : out   std_logic;
    TB29_latched        : out   std_logic;
    TB30_latched        : out   std_logic;
    TB31_latched        : out   std_logic;
    TB32_latched        : out   std_logic;
    TB33_latched        : out   std_logic;
    TB34_latched        : out   std_logic;
    TB35_latched        : out   std_logic;
    TB36_latched        : out   std_logic;
    TB37_latched        : out   std_logic;
    TB38_latched        : out   std_logic;
    TB39_latched        : out   std_logic;
    TB40_latched        : out   std_logic;
    TB41_latched        : out   std_logic;
    TB42_latched        : out   std_logic;
    TB43_latched        : out   std_logic;
    TB44_latched        : out   std_logic;
    TB45_latched        : out   std_logic;
    TB46_latched        : out   std_logic;
    TB47_latched        : out   std_logic;
    TB48_latched        : out   std_logic;
    TB49_latched        : out   std_logic;
    TB50_latched        : out   std_logic;
    TB51_latched        : out   std_logic;
    TB52_latched        : out   std_logic;
    TB53_latched        : out   std_logic;
    TB54_latched        : out   std_logic;
    TB55_latched        : out   std_logic;
    TB56_latched        : out   std_logic;
```

```
    TB57_latched        : out   std_logic;
    TB58_latched        : out   std_logic;
    TB59_latched        : out   std_logic;
    TB60_latched        : out   std_logic;
    TB61_latched        : out   std_logic;
    TB62_latched        : out   std_logic;
    TB63_latched        : out   std_logic
);
end component;

signal clock_enable     : std_logic_vector(0 to 31);

begin

clock_gates: write_enable port map(enable_TB_update,step,clock_enable);

word_latch: decision_bits port map(clk,clock_enable,TB00,TB01,
TB00_latched,TB01_latched,TB02_latched,TB03_latched,TB04_latched,
TB05_latched,TB06_latched,TB07_latched,TB08_latched,TB09_latched,
TB10_latched,TB11_latched,TB12_latched,TB13_latched,TB14_latched,
TB15_latched,TB16_latched,TB17_latched,TB18_latched,TB19_latched,
TB20_latched,TB21_latched,TB22_latched,TB23_latched,TB24_latched,
TB25_latched,TB26_latched,TB27_latched,TB28_latched,TB29_latched,
TB30_latched,TB31_latched,TB32_latched,TB33_latched,TB34_latched,
TB35_latched,TB36_latched,TB37_latched,TB38_latched,TB39_latched,
TB40_latched,TB41_latched,TB42_latched,TB43_latched,TB44_latched,
TB45_latched,TB46_latched,TB47_latched,TB48_latched,TB49_latched,
TB50_latched,TB51_latched,TB52_latched,TB53_latched,TB54_latched,
TB55_latched,TB56_latched,TB57_latched,TB58_latched,TB59_latched,
TB60_latched,TB61_latched,TB62_latched,TB63_latched);

end structural;



--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- 64-bit storage of the decision bits. part of the TB_update

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity decision_bits is
port
(
    clk                 : in  std_logic;
    clock_enable        : in  std_logic_vector(0 to 31);
    TB00       : in std_logic;
    TB01       : in std_logic;

    TB00_latched        : out   std_logic;
    TB01_latched        : out   std_logic;
    TB02_latched        : out   std_logic;
    TB03_latched        : out   std_logic;
    TB04_latched        : out   std_logic;
    TB05_latched        : out   std_logic;
    TB06_latched        : out   std_logic;
    TB07_latched        : out   std_logic;
    TB08_latched        : out   std_logic;
```

```
    TB09_latched        : out   std_logic;
    TB10_latched        : out   std_logic;
    TB11_latched        : out   std_logic;
    TB12_latched        : out   std_logic;
    TB13_latched        : out   std_logic;
    TB14_latched        : out   std_logic;
    TB15_latched        : out   std_logic;
    TB16_latched        : out   std_logic;
    TB17_latched        : out   std_logic;
    TB18_latched        : out   std_logic;
    TB19_latched        : out   std_logic;
    TB20_latched        : out   std_logic;
    TB21_latched        : out   std_logic;
    TB22_latched        : out   std_logic;
    TB23_latched        : out   std_logic;
    TB24_latched        : out   std_logic;
    TB25_latched        : out   std_logic;
    TB26_latched        : out   std_logic;
    TB27_latched        : out   std_logic;
    TB28_latched        : out   std_logic;
    TB29_latched        : out   std_logic;
    TB30_latched        : out   std_logic;
    TB31_latched        : out   std_logic;
    TB32_latched        : out   std_logic;
    TB33_latched        : out   std_logic;
    TB34_latched        : out   std_logic;
    TB35_latched        : out   std_logic;
    TB36_latched        : out   std_logic;
    TB37_latched        : out   std_logic;
    TB38_latched        : out   std_logic;
    TB39_latched        : out   std_logic;
    TB40_latched        : out   std_logic;
    TB41_latched        : out   std_logic;
    TB42_latched        : out   std_logic;
    TB43_latched        : out   std_logic;
    TB44_latched        : out   std_logic;
    TB45_latched        : out   std_logic;
    TB46_latched        : out   std_logic;
    TB47_latched        : out   std_logic;
    TB48_latched        : out   std_logic;
    TB49_latched        : out   std_logic;
    TB50_latched        : out   std_logic;
    TB51_latched        : out   std_logic;
    TB52_latched        : out   std_logic;
    TB53_latched        : out   std_logic;
    TB54_latched        : out   std_logic;
    TB55_latched        : out   std_logic;
    TB56_latched        : out   std_logic;
    TB57_latched        : out   std_logic;
    TB58_latched        : out   std_logic;
    TB59_latched        : out   std_logic;
    TB60_latched        : out   std_logic;
    TB61_latched        : out   std_logic;
    TB62_latched        : out   std_logic;
    TB63_latched        : out   std_logic


);
end decision_bits;

architecture behavioral of decision_bits is
    signal gated_clock  : std_logic_vector(0 to 63);
    signal gclk00       : std_logic;
```

```
signal gclk01        : std_logic;
signal gclk02        : std_logic;
signal gclk03        : std_logic;
signal gclk04        : std_logic;
signal gclk05        : std_logic;
signal gclk06        : std_logic;
signal gclk07        : std_logic;
signal gclk08        : std_logic;
signal gclk09        : std_logic;

signal gclk10        : std_logic;
signal gclk11        : std_logic;
signal gclk12        : std_logic;
signal gclk13        : std_logic;
signal gclk14        : std_logic;
signal gclk15        : std_logic;
signal gclk16        : std_logic;
signal gclk17        : std_logic;
signal gclk18        : std_logic;
signal gclk19        : std_logic;

signal gclk20        : std_logic;
signal gclk21        : std_logic;
signal gclk22        : std_logic;
signal gclk23        : std_logic;
signal gclk24        : std_logic;
signal gclk25        : std_logic;
signal gclk26        : std_logic;
signal gclk27        : std_logic;
signal gclk28        : std_logic;
signal gclk29        : std_logic;

signal gclk30        : std_logic;
signal gclk31        : std_logic;
signal gclk32        : std_logic;
signal gclk33        : std_logic;
signal gclk34        : std_logic;
signal gclk35        : std_logic;
signal gclk36        : std_logic;
signal gclk37        : std_logic;
signal gclk38        : std_logic;
signal gclk39        : std_logic;

signal gclk40        : std_logic;
signal gclk41        : std_logic;
signal gclk42        : std_logic;
signal gclk43        : std_logic;
signal gclk44        : std_logic;
signal gclk45        : std_logic;
signal gclk46        : std_logic;
signal gclk47        : std_logic;
signal gclk48        : std_logic;
signal gclk49        : std_logic;

signal gclk50        : std_logic;
signal gclk51        : std_logic;
signal gclk52        : std_logic;
signal gclk53        : std_logic;
signal gclk54        : std_logic;
signal gclk55        : std_logic;
signal gclk56        : std_logic;
signal gclk57        : std_logic;
signal gclk58        : std_logic;
```

```
    signal gclk59       : std_logic;

    signal gclk60       : std_logic;
    signal gclk61       : std_logic;
    signal gclk62       : std_logic;
    signal gclk63       : std_logic;
    signal decision_bit_vector  : std_logic_vector(0 to 63);
begin

clock_gate: process (clock_enable, clk)
variable i :integer range 0 to 31;
variable double_i :integer range 0 to 63;
variable double_i_inc :integer range 0 to 63;
begin

--generate clock gate

for i in 0 to 31 loop
    double_i:=i*2;
    double_i_inc:=i*2+1;
    gated_clock(double_i)<=not clk and clock_enable(i);
    gated_clock(double_i_inc)<=not clk and clock_enable(i);
end loop;

end process;

clk_conv: process (gated_clock)
begin
--dummy signal
    gclk00<=gated_clock(00);
    gclk01<=gated_clock(01);
    gclk02<=gated_clock(02);
    gclk03<=gated_clock(03);
    gclk04<=gated_clock(04);
    gclk05<=gated_clock(05);
    gclk06<=gated_clock(06);
    gclk07<=gated_clock(07);
    gclk08<=gated_clock(08);
    gclk09<=gated_clock(09);

    gclk10<=gated_clock(10);
    gclk11<=gated_clock(11);
    gclk12<=gated_clock(12);
    gclk13<=gated_clock(13);
    gclk14<=gated_clock(14);
    gclk15<=gated_clock(15);
    gclk16<=gated_clock(16);
    gclk17<=gated_clock(17);
    gclk18<=gated_clock(18);
    gclk19<=gated_clock(19);

    gclk20<=gated_clock(20);
    gclk21<=gated_clock(21);
    gclk22<=gated_clock(22);
    gclk23<=gated_clock(23);
    gclk24<=gated_clock(24);
    gclk25<=gated_clock(25);
    gclk26<=gated_clock(26);
    gclk27<=gated_clock(27);
    gclk28<=gated_clock(28);
    gclk29<=gated_clock(29);

    gclk30<=gated_clock(30);
```

```
    gclk31<=gated_clock(31);
    gclk32<=gated_clock(32);
    gclk33<=gated_clock(33);
    gclk34<=gated_clock(34);
    gclk35<=gated_clock(35);
    gclk36<=gated_clock(36);
    gclk37<=gated_clock(37);
    gclk38<=gated_clock(38);
    gclk39<=gated_clock(39);

    gclk40<=gated_clock(40);
    gclk41<=gated_clock(41);
    gclk42<=gated_clock(42);
    gclk43<=gated_clock(43);
    gclk44<=gated_clock(44);
    gclk45<=gated_clock(45);
    gclk46<=gated_clock(46);
    gclk47<=gated_clock(47);
    gclk48<=gated_clock(48);
    gclk49<=gated_clock(49);

    gclk50<=gated_clock(50);
    gclk51<=gated_clock(51);
    gclk52<=gated_clock(52);
    gclk53<=gated_clock(53);
    gclk54<=gated_clock(54);
    gclk55<=gated_clock(55);
    gclk56<=gated_clock(56);
    gclk57<=gated_clock(57);
    gclk58<=gated_clock(58);
    gclk59<=gated_clock(59);

    gclk60<=gated_clock(60);
    gclk61<=gated_clock(61);
    gclk62<=gated_clock(62);
    gclk63<=gated_clock(63);

end process;

--latched signals
    latch00 : process(gclk00)
    begin
    if gclk00'event and gclk00 ='1' then
        decision_bit_vector(0)<=TB00;
    end if;
    end process;

    latch01 : process(gclk01)
    begin
    if gclk01'event and gclk01 ='1' then
        decision_bit_vector(1)<=TB01;
    end if;
    end process;

    latch02 : process(gclk02)
    begin
    if gclk02'event and gclk02 ='1' then
        decision_bit_vector(2)<=TB00;
    end if;
    end process;

    latch03 : process(gclk03)
    begin
```

```
if gclk03'event and gclk03 ='1' then
    decision_bit_vector(3)<=TB01;
end if;
end process;

latch04 : process(gclk04)
begin
if gclk04'event and gclk04 ='1' then
    decision_bit_vector(4)<=TB00;
end if;
end process;

latch05 : process(gclk05)
begin
if gclk05'event and gclk05='1' then
    decision_bit_vector(5)<=TB01;
end if;
end process;

latch06 : process(gclk06)
begin
if gclk06'event and gclk06 ='1' then
    decision_bit_vector(6)<=TB00;
end if;
end process;

latch07 : process(gclk07)
begin
if gclk07'event and gclk07 ='1' then
    decision_bit_vector(7)<=TB01;
end if;
end process;

latch08 : process(gclk08)
begin
if gclk08'event and gclk08 ='1' then
    decision_bit_vector(8)<=TB00;
end if;
end process;

latch09 : process(gclk09)
begin
if gclk09'event and gclk09 ='1' then
    decision_bit_vector(9)<=TB01;
end if;
end process;

latch10 : process(gclk10)
begin
if gclk10'event and gclk10 ='1' then
    decision_bit_vector(10)<=TB00;
end if;
end process;

latch11 : process(gclk11)
begin
if gclk11'event and gclk11 ='1' then
    decision_bit_vector(11)<=TB01;
end if;
end process;

latch12 : process(gclk12)
begin
```

```
if gclk12'event and gclk12 ='1' then
    decision_bit_vector(12)<=TB00;
end if;
end process;

latch13 : process(gclk13)
begin
if gclk13'event and gclk13 ='1' then
    decision_bit_vector(13)<=TB01;
end if;
end process;

latch14 : process(gclk14)
begin
if gclk14'event and gclk14='1' then
    decision_bit_vector(14)<=TB00;
end if;
end process;

latch15 : process(gclk15)
begin
if gclk15'event and gclk15 ='1' then
    decision_bit_vector(15)<=TB01;
end if;
end process;

latch16 : process(gclk16)
begin
if gclk16'event and gclk16 ='1' then
    decision_bit_vector(16)<=TB00;
end if;
end process;

latch17 : process(gclk17)
begin
if gclk17'event and gclk17 ='1' then
    decision_bit_vector(17)<=TB01;
end if;
end process;

latch18 : process(gclk18)
begin
if gclk18'event and gclk18 ='1' then
    decision_bit_vector(18)<=TB00;
end if;
end process;

latch19 : process(gclk19)
begin
if gclk19'event and gclk19 ='1' then
    decision_bit_vector(19)<=TB01;
end if;
end process;

latch20 : process(gclk20)
begin
if gclk20'event and gclk20 ='1' then
    decision_bit_vector(20)<=TB00;
end if;
end process;

latch21 : process(gclk21)
begin
```

```
if gclk21'event and gclk21 ='1' then
    decision_bit_vector(21)<=TB01;
end if;
end process;

latch22 : process(gclk22)
begin
if gclk22'event and gclk22='1' then
    decision_bit_vector(22)<=TB00;
end if;
end process;

latch23 : process(gclk23)
begin
if gclk23'event and gclk23 ='1' then
    decision_bit_vector(23)<=TB01;
end if;
end process;

latch24 : process(gclk24)
begin
if gclk24'event and gclk24 ='1' then
    decision_bit_vector(24)<=TB00;
end if;
end process;

latch25 : process(gclk25)
begin
if gclk25'event and gclk25 ='1' then
    decision_bit_vector(25)<=TB01;
end if;
end process;

latch26 : process(gclk26)
begin
if gclk26'event and gclk26='1' then
    decision_bit_vector(26)<=TB00;
end if;
end process;

latch27 : process(gclk27)
begin
if gclk27'event and gclk27 ='1' then
    decision_bit_vector(27)<=TB01;
end if;
end process;

latch28 : process(gclk28)
begin
if gclk28'event and gclk28 ='1' then
    decision_bit_vector(28)<=TB00;
end if;
end process;

latch29 : process(gclk29)
begin
if gclk29'event and gclk29 ='1' then
    decision_bit_vector(29)<=TB01;
end if;
end process;

latch30 : process(gclk30)
begin
```

```
if gclk30'event and gclk30 ='1' then
    decision_bit_vector(30)<=TB00;
end if;
end process;

latch31 : process(gclk31)
begin
if gclk31'event and gclk31 ='1' then
    decision_bit_vector(31)<=TB01;
end if;
end process;


latch32 : process(gclk32)
begin
if gclk32'event and gclk32 ='1' then
    decision_bit_vector(32)<=TB00;
end if;
end process;

latch33 : process(gclk33)
begin
if gclk33'event and gclk33 ='1' then
    decision_bit_vector(33)<=TB01;
end if;
end process;

latch34 : process(gclk34)
begin
if gclk34'event and gclk34 ='1' then
    decision_bit_vector(34)<=TB00;
end if;
end process;

latch35 : process(gclk35)
begin
if gclk35'event and gclk35 ='1' then
    decision_bit_vector(35)<=TB01;
end if;
end process;

latch36 : process(gclk36)
begin
if gclk36'event and gclk36 ='1' then
    decision_bit_vector(36)<=TB00;
end if;
end process;

latch37 : process(gclk37)
begin
if gclk37'event and gclk37='1' then
    decision_bit_vector(37)<=TB01;
end if;
end process;

latch38 : process(gclk38)
begin
if gclk38'event and gclk38 ='1' then
    decision_bit_vector(38)<=TB00;
end if;
end process;

latch39 : process(gclk39)
```

```
begin
if gclk39'event and gclk39 ='1' then
    decision_bit_vector(39)<=TB01;
end if;
end process;

latch40 : process(gclk40)
begin
if gclk40'event and gclk40 ='1' then
    decision_bit_vector(40)<=TB00;
end if;
end process;

latch41 : process(gclk41)
begin
if gclk41'event and gclk41 ='1' then
    decision_bit_vector(41)<=TB01;
end if;
end process;

latch42 : process(gclk42)
begin
if gclk42'event and gclk42 ='1' then
    decision_bit_vector(42)<=TB00;
end if;
end process;

latch43 : process(gclk43)
begin
if gclk43'event and gclk43 ='1' then
    decision_bit_vector(43)<=TB01;
end if;
end process;

latch44 : process(gclk44)
begin
if gclk44'event and gclk44 ='1' then
    decision_bit_vector(44)<=TB00;
end if;
end process;

latch45 : process(gclk45)
begin
if gclk45'event and gclk45 ='1' then
    decision_bit_vector(45)<=TB01;
end if;
end process;

latch46 : process(gclk46)
begin
if gclk46'event and gclk46='1' then
    decision_bit_vector(46)<=TB00;
end if;
end process;

latch47 : process(gclk47)
begin
if gclk47'event and gclk47 ='1' then
    decision_bit_vector(47)<=TB01;
end if;
end process;

latch48 : process(gclk48)
```

```
begin
if gclk48'event and gclk48 ='1' then
    decision_bit_vector(48)<=TB00;
end if;
end process;

latch49 : process(gclk49)
begin
if gclk49'event and gclk49 ='1' then
    decision_bit_vector(49)<=TB01;
end if;
end process;

latch50 : process(gclk50)
begin
if gclk50'event and gclk50 ='1' then
    decision_bit_vector(50)<=TB00;
end if;
end process;

latch51 : process(gclk51)
begin
if gclk51'event and gclk51 ='1' then
    decision_bit_vector(51)<=TB01;
end if;
end process;

latch52 : process(gclk52)
begin
if gclk52'event and gclk52 ='1' then
    decision_bit_vector(52)<=TB00;
end if;
end process;

latch53 : process(gclk53)
begin
if gclk53'event and gclk53 ='1' then
    decision_bit_vector(53)<=TB01;
end if;
end process;

latch54 : process(gclk54)
begin
if gclk54'event and gclk54='1' then
    decision_bit_vector(54)<=TB00;
end if;
end process;

latch55 : process(gclk55)
begin
if gclk55'event and gclk55 ='1' then
    decision_bit_vector(55)<=TB01;
end if;
end process;

latch56 : process(gclk56)
begin
if gclk56'event and gclk56 ='1' then
    decision_bit_vector(56)<=TB00;
end if;
end process;

latch57 : process(gclk57)
```

```
    begin
    if gclk57'event and gclk57 ='1' then
        decision_bit_vector(57)<=TB01;
    end if;
    end process;

    latch58 : process(gclk58)
    begin
    if gclk58'event and gclk58='1' then
        decision_bit_vector(58)<=TB00;
    end if;
    end process;

    latch59 : process(gclk59)
    begin
    if gclk59'event and gclk59 ='1' then
        decision_bit_vector(59)<=TB01;
    end if;
    end process;

    latch60 : process(gclk60)
    begin
    if gclk60'event and gclk60 ='1' then
        decision_bit_vector(60)<=TB00;
    end if;
    end process;

    latch61 : process(gclk61)
    begin
    if gclk61'event and gclk61 ='1' then
        decision_bit_vector(61)<=TB01;
    end if;
    end process;

    latch62 : process(gclk62)
    begin
    if gclk62'event and gclk62 ='1' then
        decision_bit_vector(62)<=TB00;
    end if;
    end process;

    latch63 : process(gclk63)
    begin
    if gclk63'event and gclk63 ='1' then
        decision_bit_vector(63)<=TB01;
    end if;
    end process;

output_stage: process(decision_bit_vector)
begin
-- dummy
    TB00_latched<=decision_bit_vector(0);
    TB01_latched<=decision_bit_vector(1);
    TB02_latched<=decision_bit_vector(2);
    TB03_latched<=decision_bit_vector(3);
    TB04_latched<=decision_bit_vector(4);
    TB05_latched<=decision_bit_vector(5);
    TB06_latched<=decision_bit_vector(6);
    TB07_latched<=decision_bit_vector(7);
    TB08_latched<=decision_bit_vector(8);
    TB09_latched<=decision_bit_vector(9);

    TB10_latched<=decision_bit_vector(10);
```

```
    TB11_latched<=decision_bit_vector(11);
    TB12_latched<=decision_bit_vector(12);
    TB13_latched<=decision_bit_vector(13);
    TB14_latched<=decision_bit_vector(14);
    TB15_latched<=decision_bit_vector(15);
    TB16_latched<=decision_bit_vector(16);
    TB17_latched<=decision_bit_vector(17);
    TB18_latched<=decision_bit_vector(18);
    TB19_latched<=decision_bit_vector(19);

    TB20_latched<=decision_bit_vector(20);
    TB21_latched<=decision_bit_vector(21);
    TB22_latched<=decision_bit_vector(22);
    TB23_latched<=decision_bit_vector(23);
    TB24_latched<=decision_bit_vector(24);
    TB25_latched<=decision_bit_vector(25);
    TB26_latched<=decision_bit_vector(26);
    TB27_latched<=decision_bit_vector(27);
    TB28_latched<=decision_bit_vector(28);
    TB29_latched<=decision_bit_vector(29);

    TB30_latched<=decision_bit_vector(30);
    TB31_latched<=decision_bit_vector(31);
    TB32_latched<=decision_bit_vector(32);
    TB33_latched<=decision_bit_vector(33);
    TB34_latched<=decision_bit_vector(34);
    TB35_latched<=decision_bit_vector(35);
    TB36_latched<=decision_bit_vector(36);
    TB37_latched<=decision_bit_vector(37);
    TB38_latched<=decision_bit_vector(38);
    TB39_latched<=decision_bit_vector(39);

    TB40_latched<=decision_bit_vector(40);
    TB41_latched<=decision_bit_vector(41);
    TB42_latched<=decision_bit_vector(42);
    TB43_latched<=decision_bit_vector(43);
    TB44_latched<=decision_bit_vector(44);
    TB45_latched<=decision_bit_vector(45);
    TB46_latched<=decision_bit_vector(46);
    TB47_latched<=decision_bit_vector(47);
    TB48_latched<=decision_bit_vector(48);
    TB49_latched<=decision_bit_vector(49);

    TB50_latched<=decision_bit_vector(50);
    TB51_latched<=decision_bit_vector(51);
    TB52_latched<=decision_bit_vector(52);
    TB53_latched<=decision_bit_vector(53);
    TB54_latched<=decision_bit_vector(54);
    TB55_latched<=decision_bit_vector(55);
    TB56_latched<=decision_bit_vector(56);
    TB57_latched<=decision_bit_vector(57);
    TB58_latched<=decision_bit_vector(58);
    TB59_latched<=decision_bit_vector(59);

    TB60_latched<=decision_bit_vector(60);
    TB61_latched<=decision_bit_vector(61);
    TB62_latched<=decision_bit_vector(62);
    TB63_latched<=decision_bit_vector(63);

end process;

end behavioral;
```

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- generate clock gate of the TB_update unit

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity write_enable is
port
(
        enable_TB_update    : in    std_logic;
        step              : in   integer range 0 to 31;
        clock_enable      : out  std_logic_vector(0 to 31)
);
end write_enable;

architecture behavioral of write_enable is
begin

main: process (step,enable_TB_update)
variable i : integer range 0 to 31;
begin

-- clock gates for TB update
for i in 0 to 31 loop
    if enable_TB_update='1' then
        if i= step then
            clock_enable(i) <= '1' ;
        else
            clock_enable(i) <= '0' ;
        end if;
    else
        clock_enable<="00000000000000000000000000000000";
    end if;


end loop;

end process;

end behavioral;
```

A.1.20 MAX

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- find the temporary best path and its index
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity max is
```

```
port(

    ACS00_PM_up     : in    integer range -512 to 511;
    ACS00_PM_down       : in    integer range -512 to 511;

    current_PM_max      :in     integer range -512 to 511;
    current_index_max   :in     integer range 0 to 63;

    step                : in    integer range 0 to 31;

    k5                      : in std_logic;
    k7                      : in std_logic;

    new_PM_max          : out   integer range -512 to 511;
    new_index_max       : out   integer range 0 to 63
);
end max;


architecture behavioral of  max is

begin

main : process(ACS00_PM_up,ACS00_PM_down,current_PM_max,current_index_max,step, k5,k7)

variable temp_max: integer range -512 to 511;
variable temp_index: integer range 0 to 1;
variable temp_index_true: integer range 0 to 63;
begin

--larger one of the new imputs

if ACS00_PM_up>ACS00_PM_down then
    temp_max:=ACS00_PM_up;
    temp_index:=0;
else
    temp_max:=ACS00_PM_down;
    temp_index:=1;
end if;

--true index of the temp max
if temp_index = 0 then
    temp_index_true:= step*2;
elsif k5='1' and k7 = '0' then
    temp_index_true:= 2*step+1;
elsif k5='0' and k7 = '1' then
    temp_index_true:= 2*step+1;
else
    temp_index_true:=0;
end if;

-- compare to previous best path
if temp_max>current_PM_max then
    new_PM_max<=temp_max;
    new_index_max<=temp_index_true;
else
    new_PM_max<=current_PM_max;
    new_index_max<=current_index_max;
end if;

end process;

end behavioral;
```

A.1.21 Write PM

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

-- index calculation for the path metric storage

library ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity write_PM is
port(
    step                : in    integer range 0 to 31;
    rotate_counter      : in    integer range 0 to 7;
    k5                  : in std_logic;
    k7                  : in std_logic;

    PM_up               : in integer range -512 to 511;
    PM_down             : in integer range -512 to 511;

    D_even              : out std_logic_vector (9 downto 0);
    D_odd               : out std_logic_vector (9 downto 0);

    AW_even             : out std_logic_vector (4 downto 0);
    AW_odd              : out std_logic_vector (4 downto 0)


);
end write_PM;

architecture behavioral of write_PM is
begin

main: process (step, rotate_counter, k5, k7, PM_up, PM_down)

variable step_vector : std_logic_vector (4 downto 0);
variable J_odd_parity: std_logic;
variable index_up, index_down: integer range 0 to 63;
variable index_up_vector, index_down_vector: std_logic_vector (5 downto 0);

begin

step_vector:=conv_std_logic_vector(step,5);

-- check parity if J
J_odd_parity:=step_vector(0) xor step_vector(1) xor step_vector(2)
xor step_vector(3) xor step_vector(4);

if k5='1' and k7='0' and step<=7 then
--rotate index

index_up:=conv_integer(unsigned
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(step,4),'0')
ror rotate_counter) ));

index_down:= conv_integer(unsigned
```

```
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(8+step,4),'0')
ror rotate_counter) ));

elsif k5='0' and k7='1' then
--rotate index

index_up:= conv_integer(unsigned
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(step,6),'0')
ror rotate_counter) ));

index_down:= conv_integer(unsigned
(To_stdlogicvector(To_bitvector(conv_std_logic_vector(32+step,6),'0')
ror rotate_counter) ));

else
  index_up:=0;
  index_down:=0;

end if;


index_up_vector:=conv_std_logic_vector(index_up, 6);
index_down_vector:=conv_std_logic_vector(index_down, 6);

-- output address and data
if J_odd_parity='1' then
    AW_odd<=index_up_vector(5 downto 1);
    AW_even<=index_down_vector(5 downto 1);
    D_odd<=conv_std_logic_vector(PM_up,10);
    D_even<=conv_std_logic_vector(PM_down,10);
else
    AW_even<=index_up_vector(5 downto 1);
    AW_odd<=index_down_vector(5 downto 1);
    D_even<=conv_std_logic_vector(PM_up,10);
    D_odd<=conv_std_logic_vector(PM_down,10);
end if;



end process;
end behavioral;
```

A.1.22 Pipeline 3

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--pipeline register

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity pip3 is
port
(
    clk             : in std_logic;
```

```
    flush           : in std_logic;


    enable_in      : in std_logic;

    pm_max_in          : in   integer range -512 to 511;
    index_max_in       : in   integer range 0 to 63;



    pm_max_out          : out   integer range -512 to 511;
    index_max_out       : out   integer range 0 to 63
);
end pip3;

architecture behavioral of pip3 is
signal      pm_max_out_int          : integer range -512 to 511;
signal      index_max_out_int       : integer range 0 to 63;

begin

latch: process (enable_in, flush, clk)


begin

if flush='1' then
--reset
        pm_max_out_int<=-512;
        index_max_out_int<=0;
elsif clk'event and clk='1' then
   if enable_in = '1' then
--enabled
        pm_max_out_int<=pm_max_in;
        index_max_out_int<=index_max_in;
   end if;
end if;
end process;

to_output:process(pm_max_out_int,index_max_out_int)
begin
--dummy signal for DPFLI
pm_max_out<=pm_max_out_int;
index_max_out<=index_max_out_int;
end process;

end behavioral;
```

A.1.23 Cyclic decoder

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--structure of the cyclic decoder

library ieee;
use ieee.std_logic_1164.all;
```

```
entity cyclic_decoder is
port(
    reset       : in std_logic;
    clk     : in std_logic;

    run_c       : in std_logic;
    data_in_c   : in std_logic_vector(31 downto 0);
    block_size  : in integer range 0 to 612;
    poly_c      : in std_logic_vector(3 downto 0);

    read_c      : out std_logic;
    write_c     : out std_logic;
    data_out_c  : out std_logic_vector(31 downto 0);
    address_c   : out std_logic_vector(9 downto 0);

    state_c     : out std_logic
);
end cyclic_decoder;

architecture structural of cyclic_decoder is

component CRC_control_unit
port(
    reset       : in std_logic;
    clk         : in std_logic;

    run_c       : in std_logic;
    data_in_c   : in std_logic_vector(31 downto 0);
    poly_c      : in std_logic_vector(3 downto 0);
    block_size  : in integer range 0 to 612;


    out_00      : in    std_logic;
    out_01      : in    std_logic;
    out_02      : in    std_logic;
    out_03      : in    std_logic;
    out_04      : in    std_logic;
    out_05      : in    std_logic;
    out_06      : in    std_logic;
    out_07      : in    std_logic;
    out_08      : in    std_logic;
    out_09      : in    std_logic;

    out_10      : in    std_logic;
    out_11      : in    std_logic;
    out_12      : in    std_logic;
    out_13      : in    std_logic;
    out_14      : in    std_logic;
    out_15      : in    std_logic;
    out_16      : in    std_logic;
    out_17      : in    std_logic;
    out_18      : in    std_logic;
    out_19      : in    std_logic;

    out_20      : in    std_logic;
    out_21      : in    std_logic;
    out_22      : in    std_logic;
    out_23      : in    std_logic;
    out_24      : in    std_logic;
    out_25      : in    std_logic;
    out_26      : in    std_logic;
    out_27      : in    std_logic;
```

```
    out_28      : in    std_logic;
    out_29      : in    std_logic;

    out_30      : in    std_logic;
    out_31      : in    std_logic;
    out_32      : in    std_logic;
    out_33      : in    std_logic;
    out_34      : in    std_logic;
    out_35      : in    std_logic;
    out_36      : in    std_logic;
    out_37      : in    std_logic;
    out_38      : in    std_logic;
    out_39      : in    std_logic;

    read_c      : out   std_logic;
    write_c     : out   std_logic;
    address_c   : out   std_logic_vector(9 downto 0);
    data_out_c  : out   std_logic_vector(31 downto 0);
    state_c     : out   std_logic;

    shift       : out   std_logic;
    in_bit      : out   std_logic;
    clear       : out   std_logic
);

end component;

component shift_chain
port(
    clk     : in    std_logic;
    clear       : in    std_logic;
    in_bit      : in    std_logic;
    shift       : in    std_logic;
    poly_c      : in    std_logic_vector (3 downto 0);

    out_00      : out   std_logic;
    out_01      : out   std_logic;
    out_02      : out   std_logic;
    out_03      : out   std_logic;
    out_04      : out   std_logic;
    out_05      : out   std_logic;
    out_06      : out   std_logic;
    out_07      : out   std_logic;
    out_08      : out   std_logic;
    out_09      : out   std_logic;

    out_10      : out   std_logic;
    out_11      : out   std_logic;
    out_12      : out   std_logic;
    out_13      : out   std_logic;
    out_14      : out   std_logic;
    out_15      : out   std_logic;
    out_16      : out   std_logic;
    out_17      : out   std_logic;
    out_18      : out   std_logic;
    out_19      : out   std_logic;

    out_20      : out   std_logic;
    out_21      : out   std_logic;
    out_22      : out   std_logic;
    out_23      : out   std_logic;
    out_24      : out   std_logic;
    out_25      : out   std_logic;
```

```
    out_26      : out   std_logic;
    out_27      : out   std_logic;
    out_28      : out   std_logic;
    out_29      : out   std_logic;

    out_30      : out   std_logic;
    out_31      : out   std_logic;
    out_32      : out   std_logic;
    out_33      : out   std_logic;
    out_34      : out   std_logic;
    out_35      : out   std_logic;
    out_36      : out   std_logic;
    out_37      : out   std_logic;
    out_38      : out   std_logic;
    out_39      : out   std_logic
);
end component;

signal out_00,out_01,out_02,out_03,out_04,out_05,out_06,out_07,out_08,out_09,
out_10,out_11,out_12,out_13,out_14,out_15,out_16,out_17,out_18,out_19,out_20,
out_21,out_22,out_23,out_24,out_25,out_26,out_27,out_28,out_29,out_30,out_31,
out_32,out_33,out_34,out_35,out_36,out_37,out_38,out_39: std_logic;

signal shift, in_bit, clear: std_logic;

begin

controller: CRC_control_unit port map (reset, clk, run_c, data_in_c, poly_c,
block_size,out_00,out_01,out_02,out_03,out_04,out_05,out_06,out_07,out_08,
out_09,out_10,out_11,out_12,out_13,out_14,out_15,out_16,out_17,out_18,out_19,
out_20,out_21,out_22,out_23,out_24,out_25,out_26,out_27,out_28,out_29,out_30,
out_31,out_32,out_33,out_34,out_35,out_36,out_37,out_38,out_39,read_c,write_c,
address_c,data_out_c,state_c, shift, in_bit, clear );

crc_chain: shift_chain  port map (clk, clear, in_bit, shift, poly_c, out_00,
out_01,out_02,out_03,out_04,out_05,out_06,out_07,out_08,out_09,out_10,out_11,
out_12,out_13,out_14,out_15,out_16,out_17,out_18,out_19,out_20,out_21,out_22,
out_23,out_24,out_25,out_26,out_27,out_28,out_29,out_30,out_31,out_32,out_33,
out_34,out_35,out_36,out_37,out_38,out_39);

end structural;
```

A.1.24 Cyclic decoder control

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--controller of the CRC unit

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity CRC_control_unit is
port(
    reset       : in std_logic;
    clk         : in std_logic;
```

```
    run_c       : in std_logic;
    data_in_c   : in std_logic_vector(31 downto 0);
    poly_c      : in std_logic_vector(3 downto 0);
    block_size  : in integer range 0 to 612;


    out_00      : in    std_logic;
    out_01      : in    std_logic;
    out_02      : in    std_logic;
    out_03      : in    std_logic;
    out_04      : in    std_logic;
    out_05      : in    std_logic;
    out_06      : in    std_logic;
    out_07      : in    std_logic;
    out_08      : in    std_logic;
    out_09      : in    std_logic;

    out_10      : in    std_logic;
    out_11      : in    std_logic;
    out_12      : in    std_logic;
    out_13      : in    std_logic;
    out_14      : in    std_logic;
    out_15      : in    std_logic;
    out_16      : in    std_logic;
    out_17      : in    std_logic;
    out_18      : in    std_logic;
    out_19      : in    std_logic;

    out_20      : in    std_logic;
    out_21      : in    std_logic;
    out_22      : in    std_logic;
    out_23      : in    std_logic;
    out_24      : in    std_logic;
    out_25      : in    std_logic;
    out_26      : in    std_logic;
    out_27      : in    std_logic;
    out_28      : in    std_logic;
    out_29      : in    std_logic;

    out_30      : in    std_logic;
    out_31      : in    std_logic;
    out_32      : in    std_logic;
    out_33      : in    std_logic;
    out_34      : in    std_logic;
    out_35      : in    std_logic;
    out_36      : in    std_logic;
    out_37      : in    std_logic;
    out_38      : in    std_logic;
    out_39      : in    std_logic;

    read_c      : out   std_logic;
    write_c     : out   std_logic;
    address_c   : out   std_logic_vector(9 downto 0);
    data_out_c  : out   std_logic_vector(31 downto 0);
    state_c     : out   std_logic;

    shift       : out   std_logic;
    in_bit      : out   std_logic;
    clear       : out   std_logic
);

end CRC_control_unit;
```

```
architecture behavioral of CRC_control_unit is
type state_type is(idle,read_first, read_shift, write_back_1, write_back_2);

signal current_state, next_state : state_type;

signal current_w_counter : integer range 0 to 31;
signal next_w_counter : integer range 0 to 31;

signal current_bit_index : integer range 0 to 31;
signal next_bit_index : integer range 0 to 31;

signal current_shift_counter : integer range 0 to 612;
signal next_shift_counter : integer range 0 to 612;


begin

control_logic : process(run_c,  data_in_c,poly_c,block_size,out_00,
out_01,out_02,out_03,out_04,out_05,out_06,out_07,out_08,out_09,out_10,
out_11,out_12,out_13,out_14,out_15,out_16,out_17,out_18,out_19,out_20,
out_21,out_22,out_23,out_24,out_25,out_26,out_27,out_28,out_29,out_30,
out_31,out_32,out_33,out_34,out_35,out_36,out_37,out_38,out_39,
current_w_counter, current_shift_counter, current_bit_index, current_state)

variable w_counter_incr: integer range 0 to 31;

begin
w_counter_incr:=current_w_counter+1;

case current_state is
    when idle =>

--do nothing

        next_shift_counter<=0;
        write_c<='0';
        address_c<=conv_std_logic_vector(current_w_counter,10);
        state_c<='0';
        data_out_c<="00000000000000000000000000000000";
        clear<='1';
        in_bit<='0';
        shift<='1';
        next_w_counter<=0;
        next_bit_index<=0;
        read_c<='0';

        if run_c='0' then
            next_state<=idle;
        else
            next_state<=read_first;
        end if;

    when read_first =>
--read 32 bits from the memory
        next_shift_counter<=0;
        write_c<='0';
        address_c<="0000000000";
        state_c<='1';
        data_out_c<="00000000000000000000000000000000";
        clear<='0';
        in_bit<='0';
        shift<='0';
        next_w_counter<=0;
```

```
        next_bit_index<=0;
        next_state<=read_shift;
        read_c<='1';

    when read_shift =>
-- read data and shift data into decoder
        if current_shift_counter<block_size-1 then
            next_state<=read_shift;
            write_c<='0';
            state_c<='1';
            next_shift_counter<=current_shift_counter+1;
            data_out_c<="00000000000000000000000000000000";
            clear<='0';
            in_bit<=data_in_c(current_bit_index);
            shift<='1';
            if current_bit_index=31 then
                next_bit_index<=0;
                address_c<=conv_std_logic_vector(w_counter_incr,10);
                next_w_counter<=w_counter_incr;
                read_c<='1';
            else
                next_bit_index<=current_bit_index+1;
                address_c<=conv_std_logic_vector(current_w_counter,10);
                next_w_counter<=current_w_counter;
                read_c<='0';
            end if;
        else
            next_state<=write_back_1;
            write_c<='0';
            state_c<='1';
            next_shift_counter<=current_shift_counter+1;
            data_out_c<="00000000000000000000000000000000";
            clear<='0';
            in_bit<=data_in_c(current_bit_index);
            shift<='1';
            next_bit_index<=0;
            address_c<=conv_std_logic_vector(current_w_counter,10);
            next_w_counter<=current_w_counter;
            read_c<='0';
        end if;

    when write_back_1 =>

--stores the syndrome into the memory

        write_c<='1';
        state_C<='1';
        next_shift_counter<=0;

        if poly_c="0001" then
                          data_out_c(0)<=out_32;
                          data_out_c(1)<=out_33;
                          data_out_c(2)<=out_34;
                          data_out_c(3)<=out_35;
                          data_out_c(4)<=out_36;
                          data_out_c(5)<=out_37;
                          data_out_c(6)<=out_38;
                          data_out_c(7)<=out_39;
                          data_out_c(8)<='0';
                          data_out_c(9)<='0';

                          data_out_c(10)<='0';
                          data_out_c(11)<='0';
```

```
                  data_out_c(12)<='0';
                  data_out_c(13)<='0';
                  data_out_c(14)<='0';
                  data_out_c(15)<='0';
                  data_out_c(16)<='0';
                  data_out_c(17)<='0';
                  data_out_c(18)<='0';
                  data_out_c(19)<='0';

                  data_out_c(20)<='0';
                  data_out_c(21)<='0';
                  data_out_c(22)<='0';
                  data_out_c(23)<='0';
                  data_out_c(24)<='0';
                  data_out_c(25)<='0';
                  data_out_c(26)<='0';
                  data_out_c(27)<='0';
                  data_out_c(28)<='0';
                  data_out_c(29)<='0';

                  data_out_c(30)<='0';
                  data_out_c(31)<='0';

elsif poly_c="0010" then
                  data_out_c(0)<=out_37;
                  data_out_c(1)<=out_38;
                  data_out_c(2)<=out_39;
                  data_out_c(3)<='0';
                  data_out_c(4)<='0';
                  data_out_c(5)<='0';
                  data_out_c(6)<='0';
                  data_out_c(7)<='0';
                  data_out_c(8)<='0';
                  data_out_c(9)<='0';

                  data_out_c(10)<='0';
                  data_out_c(11)<='0';
                  data_out_c(12)<='0';
                  data_out_c(13)<='0';
                  data_out_c(14)<='0';
                  data_out_c(15)<='0';
                  data_out_c(16)<='0';
                  data_out_c(17)<='0';
                  data_out_c(18)<='0';
                  data_out_c(19)<='0';

                  data_out_c(20)<='0';
                  data_out_c(21)<='0';
                  data_out_c(22)<='0';
                  data_out_c(23)<='0';
                  data_out_c(24)<='0';
                  data_out_c(25)<='0';
                  data_out_c(26)<='0';
                  data_out_c(27)<='0';
                  data_out_c(28)<='0';
                  data_out_c(29)<='0';

                  data_out_c(30)<='0';
                  data_out_c(31)<='0';


elsif poly_c="0011" then
                  data_out_c(0)<=out_26;
```

```
                  data_out_c(1)<=out_27;
                  data_out_c(2)<=out_28;
                  data_out_c(3)<=out_29;
                  data_out_c(4)<=out_30;
                  data_out_c(5)<=out_31;
                  data_out_c(6)<=out_32;
                  data_out_c(7)<=out_33;
                  data_out_c(8)<=out_34;
                  data_out_c(9)<=out_35;

                  data_out_c(10)<=out_36;
                  data_out_c(11)<=out_37;
                  data_out_c(12)<=out_38;
                  data_out_c(13)<=out_39;
                  data_out_c(14)<='0';
                  data_out_c(15)<='0';
                  data_out_c(16)<='0';
                  data_out_c(17)<='0';
                  data_out_c(18)<='0';
                  data_out_c(19)<='0';

                  data_out_c(20)<='0';
                  data_out_c(21)<='0';
                  data_out_c(22)<='0';
                  data_out_c(23)<='0';
                  data_out_c(24)<='0';
                  data_out_c(25)<='0';
                  data_out_c(26)<='0';
                  data_out_c(27)<='0';
                  data_out_c(28)<='0';
                  data_out_c(29)<='0';

                  data_out_c(30)<='0';
                  data_out_c(31)<='0';


      elsif poly_c="0100" then
                  data_out_c(0)<=out_34;
                  data_out_c(1)<=out_35;
                  data_out_c(2)<=out_36;
                  data_out_c(3)<=out_37;
                  data_out_c(4)<=out_38;
                  data_out_c(5)<=out_39;
                  data_out_c(6)<='0';
                  data_out_c(7)<='0';
                  data_out_c(8)<='0';
                  data_out_c(9)<='0';

                  data_out_c(10)<='0';
                  data_out_c(11)<='0';
                  data_out_c(12)<='0';
                  data_out_c(13)<='0';
                  data_out_c(14)<='0';
                  data_out_c(15)<='0';
                  data_out_c(16)<='0';
                  data_out_c(17)<='0';
                  data_out_c(18)<='0';
                  data_out_c(19)<='0';

                  data_out_c(20)<='0';
                  data_out_c(21)<='0';
                  data_out_c(22)<='0';
                  data_out_c(23)<='0';
```

```
                        data_out_c(24)<='0';
                        data_out_c(25)<='0';
                        data_out_c(26)<='0';
                        data_out_c(27)<='0';
                        data_out_c(28)<='0';
                        data_out_c(29)<='0';

                        data_out_c(30)<='0';
                        data_out_c(31)<='0';

          elsif poly_c="0101" then
                        data_out_c(0)<=out_00;
                        data_out_c(1)<=out_01;
                        data_out_c(2)<=out_02;
                        data_out_c(3)<=out_03;
                        data_out_c(4)<=out_04;
                        data_out_c(5)<=out_05;
                        data_out_c(6)<=out_06;
                        data_out_c(7)<=out_07;
                        data_out_c(8)<=out_08;
                        data_out_c(9)<=out_09;

                        data_out_c(10)<=out_10;
                        data_out_c(11)<=out_11;
                        data_out_c(12)<=out_12;
                        data_out_c(13)<=out_13;
                        data_out_c(14)<=out_14;
                        data_out_c(15)<=out_15;
                        data_out_c(16)<=out_16;
                        data_out_c(17)<=out_17;
                        data_out_c(18)<=out_18;
                        data_out_c(19)<=out_19;

                        data_out_c(20)<=out_20;
                        data_out_c(21)<=out_21;
                        data_out_c(22)<=out_22;
                        data_out_c(23)<=out_23;
                        data_out_c(24)<=out_24;
                        data_out_c(25)<=out_25;
                        data_out_c(26)<=out_26;
                        data_out_c(27)<=out_27;
                        data_out_c(28)<=out_28;
                        data_out_c(29)<=out_29;

                        data_out_c(30)<=out_30;
                        data_out_c(31)<=out_31;


          elsif poly_c="0110" then
                        data_out_c(0)<=out_30;
                        data_out_c(1)<=out_31;
                        data_out_c(2)<=out_32;
                        data_out_c(3)<=out_33;
                        data_out_c(4)<=out_34;
                        data_out_c(5)<=out_35;
                        data_out_c(6)<=out_36;
                        data_out_c(7)<=out_37;
                        data_out_c(8)<=out_38;
                        data_out_c(9)<=out_39;

                        data_out_c(10)<='0';
                        data_out_c(11)<='0';
                        data_out_c(12)<='0';
```

```
                        data_out_c(13)<='0';
                        data_out_c(14)<='0';
                        data_out_c(15)<='0';
                        data_out_c(16)<='0';
                        data_out_c(17)<='0';
                        data_out_c(18)<='0';
                        data_out_c(19)<='0';

                        data_out_c(20)<='0';
                        data_out_c(21)<='0';
                        data_out_c(22)<='0';
                        data_out_c(23)<='0';
                        data_out_c(24)<='0';
                        data_out_c(25)<='0';
                        data_out_c(26)<='0';
                        data_out_c(27)<='0';
                        data_out_c(28)<='0';
                        data_out_c(29)<='0';

                        data_out_c(30)<='0';
                        data_out_c(31)<='0';


        elsif poly_c="0111" then
                        data_out_c(0)<=out_24;
                        data_out_c(1)<=out_25;
                        data_out_c(2)<=out_26;
                        data_out_c(3)<=out_27;
                        data_out_c(4)<=out_28;
                        data_out_c(5)<=out_29;
                        data_out_c(6)<=out_30;
                        data_out_c(7)<=out_31;
                        data_out_c(8)<=out_32;
                        data_out_c(9)<=out_33;

                        data_out_c(10)<=out_34;
                        data_out_c(11)<=out_35;
                        data_out_c(12)<=out_36;
                        data_out_c(13)<=out_37;
                        data_out_c(14)<=out_38;
                        data_out_c(15)<=out_39;
                        data_out_c(16)<='0';
                        data_out_c(17)<='0';
                        data_out_c(18)<='0';
                        data_out_c(19)<='0';

                        data_out_c(20)<='0';
                        data_out_c(21)<='0';
                        data_out_c(22)<='0';
                        data_out_c(23)<='0';
                        data_out_c(24)<='0';
                        data_out_c(25)<='0';
                        data_out_c(26)<='0';
                        data_out_c(27)<='0';
                        data_out_c(28)<='0';
                        data_out_c(29)<='0';

                        data_out_c(30)<='0';
                        data_out_c(31)<='0';


        elsif poly_c="1000" then
                        data_out_c(0)<=out_32;
```

```
                  data_out_c(1)<=out_33;
                  data_out_c(2)<=out_34;
                  data_out_c(3)<=out_35;
                  data_out_c(4)<=out_36;
                  data_out_c(5)<=out_37;
                  data_out_c(6)<=out_38;
                  data_out_c(7)<=out_39;
                  data_out_c(8)<='0';
                  data_out_c(9)<='0';

                  data_out_c(10)<='0';
                  data_out_c(11)<='0';
                  data_out_c(12)<='0';
                  data_out_c(13)<='0';
                  data_out_c(14)<='0';
                  data_out_c(15)<='0';
                  data_out_c(16)<='0';
                  data_out_c(17)<='0';
                  data_out_c(18)<='0';
                  data_out_c(19)<='0';

                  data_out_c(20)<='0';
                  data_out_c(21)<='0';
                  data_out_c(22)<='0';
                  data_out_c(23)<='0';
                  data_out_c(24)<='0';
                  data_out_c(25)<='0';
                  data_out_c(26)<='0';
                  data_out_c(27)<='0';
                  data_out_c(28)<='0';
                  data_out_c(29)<='0';

                  data_out_c(30)<='0';
                  data_out_c(31)<='0';



      elsif poly_c="1001" then
                  data_out_c(0)<=out_28;
                  data_out_c(1)<=out_29;
                  data_out_c(2)<=out_30;
                  data_out_c(3)<=out_31;
                  data_out_c(4)<=out_32;
                  data_out_c(5)<=out_33;
                  data_out_c(6)<=out_34;
                  data_out_c(7)<=out_35;
                  data_out_c(8)<=out_36;
                  data_out_c(9)<=out_37;

                  data_out_c(10)<=out_38;
                  data_out_c(11)<=out_39;
                  data_out_c(12)<='0';
                  data_out_c(13)<='0';
                  data_out_c(14)<='0';
                  data_out_c(15)<='0';
                  data_out_c(16)<='0';
                  data_out_c(17)<='0';
                  data_out_c(18)<='0';
                  data_out_c(19)<='0';

                  data_out_c(20)<='0';
                  data_out_c(21)<='0';
                  data_out_c(22)<='0';
```

```
                        data_out_c(23)<='0';
                        data_out_c(24)<='0';
                        data_out_c(25)<='0';
                        data_out_c(26)<='0';
                        data_out_c(27)<='0';
                        data_out_c(28)<='0';
                        data_out_c(29)<='0';

                        data_out_c(30)<='0';
                        data_out_c(31)<='0';

        else
                        data_out_c<="00000000000000000000000000000000";
        end if;


        in_bit<='0';
        next_bit_index<=0;
        address_c<=conv_std_logic_vector(w_counter_incr,10);
        next_w_counter<=w_counter_incr;
        read_c<='0';
        if poly_c="0101" then
            next_state<=write_back_2;
            clear<='0';
            shift<='0';
        else
            next_state<=idle;
            clear<='0';
            shift<='0';
        end if;



    when write_back_2 =>
-- store the rest of the syndrome in the memory
        next_state<=idle;
        write_c<='1';
        state_c<='1';
        next_shift_counter<=0;

                data_out_c(0)<=out_32;
                data_out_c(1)<=out_33;
                data_out_c(2)<=out_34;
                data_out_c(3)<=out_35;
                data_out_c(4)<=out_36;
                data_out_c(5)<=out_37;
                data_out_c(6)<=out_38;
                data_out_c(7)<=out_39;
        data_out_c(31 downto 8)<="000000000000000000000000";

        clear<='0';
        in_bit<='0';
        shift<='0';

        next_bit_index<=0;
        address_c<=conv_std_logic_vector(w_counter_incr,10);
        next_w_counter<=w_counter_incr;
        read_c<='0';



end case;
end process;
```

```
state_transition: process (clk, reset)
begin

if reset ='1' then
    current_state<=idle;
    current_bit_index<=0;
    current_w_counter<=0;
    current_shift_counter<=0;

elsif clk'event and clk='1' then
    current_state<=next_state;-- after 1 ns;
    current_bit_index<=next_bit_index;-- after 1 ns;
    current_w_counter<=next_w_counter;-- after 1 ns;
    current_shift_counter<=next_shift_counter;-- after 1 ns;
end if;
end process;

end behavioral;
```

A.1.25 Shift chain

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--CRC decoder core
--configurable register chain

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


entity shift_chain is
port(
    clk     : in    std_logic;
    clear       : in    std_logic;
    in_bit      : in    std_logic;
    shift       : in    std_logic;
    poly_c      : in    std_logic_vector (3 downto 0);

    out_00      : out   std_logic;
    out_01      : out   std_logic;
    out_02      : out   std_logic;
    out_03      : out   std_logic;
    out_04      : out   std_logic;
    out_05      : out   std_logic;
    out_06      : out   std_logic;
    out_07      : out   std_logic;
    out_08      : out   std_logic;
    out_09      : out   std_logic;

    out_10      : out   std_logic;
    out_11      : out   std_logic;
    out_12      : out   std_logic;
    out_13      : out   std_logic;
    out_14      : out   std_logic;
    out_15      : out   std_logic;
    out_16      : out   std_logic;
```

```
    out_17      : out   std_logic;
    out_18      : out   std_logic;
    out_19      : out   std_logic;

    out_20      : out   std_logic;
    out_21      : out   std_logic;
    out_22      : out   std_logic;
    out_23      : out   std_logic;
    out_24      : out   std_logic;
    out_25      : out   std_logic;
    out_26      : out   std_logic;
    out_27      : out   std_logic;
    out_28      : out   std_logic;
    out_29      : out   std_logic;

    out_30      : out   std_logic;
    out_31      : out   std_logic;
    out_32      : out   std_logic;
    out_33      : out   std_logic;
    out_34      : out   std_logic;
    out_35      : out   std_logic;
    out_36      : out   std_logic;
    out_37      : out   std_logic;
    out_38      : out   std_logic;
    out_39      : out   std_logic
);
end shift_chain;

architecture behavioral of shift_chain is
signal feedback: std_logic;
signal stage00_out,stage01_out,stage02_out,stage03_out,stage04_out,stage05_out,
stage06_out,stage07_out,stage08_out,stage09_out,stage10_out,stage11_out,
stage12_out,stage13_out,stage14_out,stage15_out,stage16_out,stage17_out,
stage18_out,stage19_out,stage20_out,stage21_out,stage22_out,stage23_out,
stage24_out,stage25_out,stage26_out,stage27_out,stage28_out,stage29_out,
stage30_out,stage31_out,stage32_out,stage33_out,stage34_out,stage35_out,
stage36_out,stage37_out,stage38_out,stage39_out: std_logic;
begin

--sorry! I forgot to remove redundant signals in the sensitive list
stage00: process(poly_c, feedback, in_bit, clear, shift,clk, stage00_out)
variable up,left, reg00_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0101" then
    left:=in_bit;
else
    left:='0';
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg00_in:='0';
            stage00_out<=reg00_in;
        else
```

```
            reg00_in:=up xor left;
            stage00_out<=reg00_in;
        end if;
    end if;
end if;

out_00<=stage00_out;
end process;



stage01: process(poly_c, feedback, in_bit, clear, shift,clk, stage00_out,stage01_out)
variable up,left, reg01_in: std_logic;
begin



if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg01_in:='0';
            stage01_out<=reg01_in;
        else
            reg01_in:=stage00_out;
            stage01_out<=reg01_in;
        end if;
    end if;
end if;

out_01<=stage01_out;
end process;



stage02: process(poly_c, feedback, in_bit, clear, shift,clk, stage01_out,stage02_out)
variable up,left, reg02_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg02_in:='0';
            stage02_out<=reg02_in;
        else
            reg02_in:=stage01_out;
            stage02_out<=reg02_in;
        end if;
    end if;
end if;

out_02<=stage02_out;
end process;



stage03: process(poly_c, feedback, in_bit, clear, shift,clk, stage03_out,stage02_out)
variable up,left, reg03_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
```

```
end if;

left:=stage02_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg03_in:='0';
            stage03_out<=reg03_in;
        else
            reg03_in:=up xor left;
            stage03_out<=reg03_in;
        end if;
    end if;
end if;

out_03<=stage03_out;
end process;


stage04: process(poly_c, feedback, in_bit, clear, shift,clk, stage03_out,stage04_out)
variable up,left, reg04_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg04_in:='0';
            stage04_out<=reg04_in;
        else
            reg04_in:=stage03_out;
            stage04_out<=reg04_in;
        end if;
    end if;
end if;

out_04<=stage04_out;
end process;




stage05: process(poly_c, feedback, in_bit, clear, shift,clk, stage05_out,stage04_out)
variable up,left, reg05_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg05_in:='0';
            stage05_out<=reg05_in;
        else
            reg05_in:=stage04_out;
            stage05_out<=reg05_in;
        end if;
    end if;
end if;

out_05<=stage05_out;
end process;
```

```
stage06: process(poly_c, feedback, in_bit, clear, shift,clk, stage06_out,stage05_out)
variable up,left, reg06_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg06_in:='0';
            stage06_out<=reg06_in;
        else
            reg06_in:=stage05_out;
            stage06_out<=reg06_in;
        end if;
    end if;
end if;

out_06<=stage06_out;
end process;


stage07: process(poly_c, feedback, in_bit, clear, shift,clk, stage07_out,stage06_out)
variable up,left, reg07_in: std_logic;
begin


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg07_in:='0';
            stage07_out<=reg07_in;
        else
            reg07_in:=stage06_out;
            stage07_out<=reg07_in;
        end if;
    end if;
end if;

out_07<=stage07_out;
end process;




stage08: process(poly_c, feedback, in_bit, clear, shift,clk, stage08_out,stage07_out)
variable up,left, reg08_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg08_in:='0';
            stage08_out<=reg08_in;
        else
            reg08_in:=stage07_out;
            stage08_out<=reg08_in;
        end if;
    end if;
end if;

out_08<=stage08_out;
end process;
```

```
stage09: process(poly_c, feedback, in_bit, clear, shift,clk, stage09_out,stage08_out)
variable up,left, reg09_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg09_in:='0';
            stage09_out<=reg09_in;
        else
            reg09_in:=stage08_out;
            stage09_out<=reg09_in;
        end if;
    end if;
end if;

out_09<=stage09_out;
end process;

stage10: process(poly_c, feedback, in_bit, clear, shift,clk, stage10_out,stage09_out)
variable up,left, reg10_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg10_in:='0';
            stage10_out<=reg10_in;
        else
            reg10_in:=stage09_out;
            stage10_out<=reg10_in;
        end if;
    end if;
end if;

out_10<=stage10_out;
end process;


stage11: process(poly_c, feedback, in_bit, clear, shift,clk, stage11_out,stage10_out)
variable up,left, reg11_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg11_in:='0';
            stage11_out<=reg11_in;
        else
            reg11_in:=stage10_out;
            stage11_out<=reg11_in;
        end if;
    end if;
end if;

out_11<=stage11_out;
end process;


stage12: process(poly_c, feedback, in_bit, clear, shift,clk, stage12_out,stage11_out)
variable up,left, reg12_in: std_logic;
begin
```

```
if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg12_in:='0';
            stage12_out<=reg12_in;
        else
            reg12_in:=stage11_out;
            stage12_out<=reg12_in;
        end if;
    end if;
end if;

out_12<=stage12_out;
end process;


stage13: process(poly_c, feedback, in_bit, clear, shift,clk, stage13_out,stage12_out)
variable up,left, reg13_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg13_in:='0';
            stage13_out<=reg13_in;
        else
            reg13_in:=stage12_out;
            stage13_out<=reg13_in;
        end if;
    end if;
end if;

out_13<=stage13_out;
end process;

stage14: process(poly_c, feedback, in_bit, clear, shift,clk, stage14_out,stage13_out)
variable up,left, reg14_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg14_in:='0';
            stage14_out<=reg14_in;
        else
            reg14_in:=stage13_out;
            stage14_out<=reg14_in;
        end if;
    end if;
end if;

out_14<=stage14_out;
end process;


stage15: process(poly_c, feedback, in_bit, clear, shift,clk, stage15_out,stage14_out)
variable up,left, reg15_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg15_in:='0';
```

```
            stage15_out<=reg15_in;
        else
            reg15_in:=stage14_out;
            stage15_out<=reg15_in;
        end if;
    end if;
end if;

out_15<=stage15_out;
end process;


stage16: process(poly_c, feedback, in_bit, clear, shift,clk, stage16_out,stage15_out)
variable up,left, reg16_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg16_in:='0';
            stage16_out<=reg16_in;
        else
            reg16_in:=stage15_out;
            stage16_out<=reg16_in;
        end if;
    end if;
end if;

out_16<=stage16_out;
end process;


stage17: process(poly_c, feedback, in_bit, clear, shift,clk, stage17_out,stage16_out)
variable up,left, reg17_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;


left:=stage16_out;

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg17_in:='0';
            stage17_out<=reg17_in;
        else
            reg17_in:=up xor left;
            stage17_out<=reg17_in;
        end if;
    end if;
end if;

out_17<=stage17_out;
end process;



stage18: process(poly_c, feedback, in_bit, clear, shift,clk, stage18_out,stage17_out)
```

```
variable up,left, reg18_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg18_in:='0';
            stage18_out<=reg18_in;
        else
            reg18_in:=stage17_out;
            stage18_out<=reg18_in;
        end if;
    end if;
end if;

out_18<=stage18_out;
end process;


stage19: process(poly_c, feedback, in_bit, clear, shift,clk, stage19_out,stage18_out)
variable up,left, reg19_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg19_in:='0';
            stage19_out<=reg19_in;
        else
            reg19_in:=stage18_out;
            stage19_out<=reg19_in;
        end if;
    end if;
end if;

out_19<=stage19_out;
end process;


stage20: process(poly_c, feedback, in_bit, clear, shift,clk, stage20_out,stage19_out)
variable up,left, reg20_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg20_in:='0';
            stage20_out<=reg20_in;
        else
            reg20_in:=stage19_out;
            stage20_out<=reg20_in;
        end if;
    end if;
end if;

out_20<=stage20_out;
end process;


stage21: process(poly_c, feedback, in_bit, clear, shift,clk, stage21_out,stage20_out)
variable up,left, reg21_in: std_logic;
begin
```

```
if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg21_in:='0';
            stage21_out<=reg21_in;
        else
            reg21_in:=stage20_out;
            stage21_out<=reg21_in;
        end if;
    end if;
end if;

out_21<=stage21_out;
end process;


stage22: process(poly_c, feedback, in_bit, clear, shift,clk, stage22_out,stage21_out)
variable up,left, reg22_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg22_in:='0';
            stage22_out<=reg22_in;
        else
            reg22_in:=stage21_out;
            stage22_out<=reg22_in;
        end if;
    end if;
end if;

out_22<=stage22_out;
end process;



stage23: process(poly_c, feedback, in_bit, clear, shift,clk, stage23_out,stage22_out)
variable up,left, reg23_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;


left:=stage22_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg23_in:='0';
            stage23_out<=reg23_in;
        else
            reg23_in:=up xor left;
            stage23_out<=reg23_in;
        end if;
    end if;
end if;
```

```
out_23<=stage23_out;
end process;



stage24: process(poly_c, feedback, in_bit, clear, shift,clk, stage24_out,stage23_out)
variable up,left, reg24_in: std_logic;
begin

if poly_c="0111" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0111" then
    left:=in_bit;
else
    left:=stage23_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg24_in:='0';
            stage24_out<=reg24_in;
        else
            reg24_in:=up xor left;
            stage24_out<=reg24_in;
        end if;
    end if;
end if;

out_24<=stage24_out;
end process;



stage25: process(poly_c, feedback, in_bit, clear, shift,clk, stage25_out,stage24_out)
variable up,left, reg25_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg25_in:='0';
            stage25_out<=reg25_in;
        else
            reg25_in:=stage24_out;
            stage25_out<=reg25_in;
        end if;
    end if;
end if;

out_25<=stage25_out;
end process;


stage26: process(poly_c, feedback, in_bit, clear, shift,clk, stage26_out,stage25_out)
variable up,left, reg26_in: std_logic;
begin
```

```
if poly_c="0011"or poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0011" then
    left:=in_bit;
else
    left:=stage25_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg26_in:='0';
            stage26_out<=reg26_in;
        else
            reg26_in:=up xor left;
            stage26_out<=reg26_in;
        end if;
    end if;
end if;

out_26<=stage26_out;
end process;


stage27: process(poly_c, feedback, in_bit, clear, shift,clk, stage27_out,stage26_out)
variable up,left, reg27_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg27_in:='0';
            stage27_out<=reg27_in;
        else
            reg27_in:=stage26_out;
            stage27_out<=reg27_in;
        end if;
    end if;
end if;

out_27<=stage27_out;
end process;

stage28: process(poly_c, feedback, in_bit, clear, shift,clk, stage28_out,stage27_out)
variable up,left, reg28_in: std_logic;
begin

if poly_c="0011" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "1001" then
    left:=in_bit;
else
    left:=stage27_out;
```

```
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg28_in:='0';
            stage28_out<=reg28_in;
        else
            reg28_in:=up xor left;
            stage28_out<=reg28_in;
        end if;
    end if;
end if;

out_28<=stage28_out;
end process;


stage29: process(poly_c, feedback, in_bit, clear, shift,clk, stage29_out,stage28_out)
variable up,left, reg29_in: std_logic;
begin

if poly_c="0011"or poly_c="0111" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage28_out;



if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg29_in:='0';
            stage29_out<=reg29_in;
        else
            reg29_in:=up xor left;
            stage29_out<=reg29_in;
        end if;
    end if;
end if;

out_29<=stage29_out;
end process;

stage30: process(poly_c, feedback, in_bit, clear, shift,clk, stage30_out,stage29_out)
variable up,left, reg30_in: std_logic;
begin

if poly_c="0110" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0110" then
    left:=in_bit;
else
    left:=stage29_out;
end if;
```

```
if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg30_in:='0';
            stage30_out<=reg30_in;
        else
            reg30_in:=up xor left;
            stage30_out<=reg30_in;
        end if;
    end if;
end if;

out_30<=stage30_out;
end process;


stage31: process(poly_c, feedback, in_bit, clear, shift,clk, stage31_out,stage30_out)
variable up,left, reg31_in: std_logic;
begin

if poly_c="0011" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage30_out;



if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg31_in:='0';
            stage31_out<=reg31_in;
        else
            reg31_in:=up xor left;
            stage31_out<=reg31_in;
        end if;
    end if;
end if;

out_31<=stage31_out;
end process;

stage32: process(poly_c, feedback, in_bit, clear, shift,clk, stage32_out,stage31_out)
variable up,left, reg32_in: std_logic;
begin

if poly_c="0001"or poly_c="0110" or poly_c="1000" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0001" or poly_c = "1000" then
    left:=in_bit;
else
    left:=stage31_out;
end if;
```

```
if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg32_in:='0';
            stage32_out<=reg32_in;
        else
            reg32_in:=up xor left;
            stage32_out<=reg32_in;
        end if;
    end if;
end if;

out_32<=stage32_out;
end process;


stage33: process(poly_c, feedback, in_bit, clear, shift,clk, stage33_out,stage32_out)
variable up,left, reg33_in: std_logic;
begin

if poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;


left:=stage32_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg33_in:='0';
            stage33_out<=reg33_in;
        else
            reg33_in:=up xor left;
            stage33_out<=reg33_in;
        end if;
    end if;
end if;

out_33<=stage33_out;
end process;


stage34: process(poly_c, feedback, in_bit, clear, shift,clk, stage34_out,stage33_out)
variable up,left, reg34_in: std_logic;
begin

if poly_c="0001" or poly_c="0100" or poly_c="0110" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0100" then
    left:=in_bit;
else
    left:=stage33_out;
end if;
```

```
if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg34_in:='0';
            stage34_out<=reg34_in;
        else
            reg34_in:=up xor left;
            stage34_out<=reg34_in;
        end if;
    end if;
end if;

out_34<=stage34_out;
end process;


stage35: process(poly_c, feedback, in_bit, clear, shift,clk, stage35_out,stage34_out)
variable up,left, reg35_in: std_logic;
begin

if poly_c="0001" or poly_c="0100" or poly_c="0110" or poly_c="1000" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage34_out;

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg35_in:='0';
            stage35_out<=reg35_in;
        else
            reg35_in:=up xor left;
            stage35_out<=reg35_in;
        end if;
    end if;
end if;

out_35<=stage35_out;
end process;

stage36: process(poly_c, feedback, in_bit, clear, shift,clk, stage36_out,stage35_out)
variable up,left, reg36_in: std_logic;
begin

if poly_c="0001" or poly_c="0100" or poly_c="0110" or poly_c="0111" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage35_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg36_in:='0';
            stage36_out<=reg36_in;
        else
```

```
            reg36_in:=up xor left;
            stage36_out<=reg36_in;
        end if;
    end if;
end if;

out_36<=stage36_out;
end process;

stage37: process(poly_c, feedback, in_bit, clear, shift,clk, stage37_out,stage36_out)
variable up,left, reg37_in: std_logic;
begin

if poly_c="0010"or poly_c="0100" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0010" then
    left:=in_bit;
else
    left:=stage36_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg37_in:='0';
            stage37_out<=reg37_in;
        else
            reg37_in:=up xor left;
            stage37_out<=reg37_in;
        end if;
    end if;
end if;

out_37<=stage37_out;
end process;

stage38: process(poly_c, feedback, in_bit, clear, shift,clk, stage38_out,stage37_out)
variable up,left, reg38_in: std_logic;
begin

if poly_c="0010" or poly_c="0110" or poly_c="1000" or poly_c="1001"  then
    up:=feedback;
else
    up:='0';
end if;


left:=stage37_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg38_in:='0';
            stage38_out<=reg38_in;
        else
            reg38_in:=up xor left;
            stage38_out<=reg38_in;
```

```
        end if;
    end if;
end if;

out_38<=stage38_out;
end process;


stage39: process(poly_c, feedback, in_bit, clear, shift,clk, stage39_out,stage38_out)
variable up,left, reg39_in: std_logic;
begin

if poly_c="0011" or poly_c="0100" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage38_out;



if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg39_in:='0';
            stage39_out<=reg39_in;
        else
            reg39_in:=up xor left;
            stage39_out<=reg39_in;
        end if;
    end if;
end if;

out_39<=stage39_out;
end process;



stage40: process(stage39_out)

begin
feedback<=stage39_out;
end process;


end behavioral;
```

A.2 Test bench

A.2.1 Decoder stimuli

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--Stimuli of the decoder. Configurable encoding unit
```

```
--if the decoding result is correct
--the signal "success" will go high
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


entity decoder_stim is
port(
    reset       : in std_logic;
    clk     : in std_logic;

    slave       : out std_logic;
    read        : out std_logic;
    write       : out std_logic;
    address : out std_logic_vector(9 downto 0);
    success : out std_logic;

    intf        : in std_logic;

    data        : inout std_logic_vector(31 downto 0)

);
end decoder_stim;

architecture behavioral of decoder_stim is

signal clear    : std_logic;
signal in_bit: std_logic;
signal shift    : std_logic;
signal poly_c: std_logic_vector (3 downto 0);

signal out_00,out_01,out_02,out_03,out_04,out_05,out_06,out_07,out_08,out_09:std_logic;
signal out_10,out_11,out_12,out_13,out_14,out_15,out_16,out_17,out_18,out_19:std_logic;
signal out_20,out_21,out_22,out_23,out_24,out_25,out_26,out_27,out_28,out_29:std_logic;
signal out_30,out_31,out_32,out_33,out_34,out_35,out_36,out_37,out_38,out_39:std_logic;

signal feedback: std_logic;
signal stage00_out,stage01_out,stage02_out,stage03_out,stage04_out,stage05_out,
    stage06_out,stage07_out: std_logic;
signal stage08_out,stage09_out,stage10_out,stage11_out,stage12_out,stage13_out,
    stage14_out,stage15_out: std_logic;
signal stage16_out,stage17_out,stage18_out,stage19_out,stage20_out,stage21_out,
    stage22_out,stage23_out: std_logic;
signal stage24_out,stage25_out,stage26_out,stage27_out,stage28_out,stage29_out,
    stage30_out,stage31_out: std_logic;
signal stage32_out,stage33_out,stage34_out,stage35_out,stage36_out,stage37_out,
    stage38_out,stage39_out: std_logic;

begin


main:process
constant Tclk : time := 10 ns;
variable data_to_encode : std_logic_vector(0 to 1023);
variable crc_coded_data : std_logic_vector(0 to 1023);
variable decoded_data : std_logic_vector(1023 downto 0);


variable data_size : integer range 0 to 1023;
variable op_mode: std_logic_vector(1 downto 0);
variable poly_v: std_logic_vector(3 downto 0);
```

```
variable threshold: std_logic_vector(8 downto 0);
variable soft_bit_no: std_logic;

variable flip_counter: integer range 0 to 32;
variable crc_counter: integer range 0 to 1023;
variable write_counter: integer range 0 to 1023;
variable read_counter: integer range 0 to 1023;
variable compare_counter: integer range 0 to 1023;


variable in_conv, rec_bit, s1_bit, s2_bit, s3_bit, s4_bit,s5_bit, s6_bit: std_logic;
variable o0, o1, o2, o3, o4,o5 : std_logic;

variable different_flag: std_logic;


begin

------------------------------------------------
-------------- coding information---------------
-- The only data need to be modified          --
op_mode:="11";
poly_c<="1001";
poly_v:="1100";
soft_bit_no:='1';
threshold:=conv_std_logic_vector(400,9);
data_to_encode(0 to 752):="1101001110011100110011011010010111011001110010101001010101
0101100011111111111100111010101111110011000001111111110000000000001111111000001010101011
0010101010101000000111111010101011000000000000000000000011001111111110000000101010101010
0000000111110000011111111011001100111001100110110100101110110011100101010010101010101
1000111111111111100111010101111110011000001111111110000000000001111111000001010101100010
1010101010000000111111010101011000000000000000000000011001111111110000000101010101010000
0001111100000111111111011001100111001100110110100101110110011100101010010101010101011000
1111111111110011101010111111001100000111111111000000000000111111100000101010110010101 0
1010100000001111110101010110000000000000000000000001100111111111000000010101010101000000001
111100000111111";
data_size:=600; --important note!!!
--notice the data size does not include CRC code. MUST BE LESS THAN 612-CRC CODE LENGTH
-------------------------------------------------

slave<='0';
read<='0';
write<='0';


-- clear shift chain
clear<='1';
in_bit<='0';
shift<='1';

wait for 31 ns;

crc_counter:=0;

if op_mode="10" or op_mode="11" then
--CRC encoding
    clear<='0';
    shift<='1';
    while crc_counter<data_size loop
--send data into encoder
        in_bit<=data_to_encode(crc_counter);
        wait for 10 ns;
```

```
        crc_counter:=crc_counter+1;
    end loop;
    shift<='0';
    crc_coded_data:=data_to_encode;

--attach redundant code to the data block

    if poly_c="0001" then
                  crc_coded_data(data_size+0):= out_39;
                          crc_coded_data(data_size+1):= out_38;
                          crc_coded_data(data_size+2):= out_37;
                          crc_coded_data(data_size+3):= out_36;
                          crc_coded_data(data_size+4):= out_35;
                          crc_coded_data(data_size+5):= out_34;
                          crc_coded_data(data_size+6):= out_33;
                          crc_coded_data(data_size+7):= out_32;
                  data_size:=data_size+8;

    elsif poly_c="0010" then
                  crc_coded_data(data_size+0):= not out_39;
                          crc_coded_data(data_size+1):= not out_38;
                          crc_coded_data(data_size+2):= not out_37;
                  data_size:=data_size+3;

    elsif poly_c="0011" then

                          crc_coded_data(data_size+0):= not out_39;
                          crc_coded_data(data_size+1):= not out_38;
                          crc_coded_data(data_size+2):= not out_37;
                          crc_coded_data(data_size+3):= not out_36;
                          crc_coded_data(data_size+4):= not out_35;
                          crc_coded_data(data_size+5):= not out_34;
                          crc_coded_data(data_size+6):= not out_33;
                          crc_coded_data(data_size+7):= not out_32;
                          crc_coded_data(data_size+8):= not out_31;
                          crc_coded_data(data_size+9):= not out_30;

                          crc_coded_data(data_size+10):= not out_29;
                          crc_coded_data(data_size+11):= not out_28;
                          crc_coded_data(data_size+12):= not out_27;
                          crc_coded_data(data_size+13):= not out_26;
                  data_size:=data_size+14;

    elsif poly_c="0100" then

                          crc_coded_data(data_size+0):= not out_39;
                          crc_coded_data(data_size+1):= not out_38;
                          crc_coded_data(data_size+2):= not out_37;
                          crc_coded_data(data_size+3):= not out_36;
                          crc_coded_data(data_size+4):= not out_35;
                          crc_coded_data(data_size+5):= not out_34;
                  data_size:=data_size+6;
    elsif poly_c="0101" then

                          crc_coded_data(data_size+0):= not out_39;
                          crc_coded_data(data_size+1):= not out_38;
                          crc_coded_data(data_size+2):= not out_37;
                          crc_coded_data(data_size+3):= not out_36;
                          crc_coded_data(data_size+4):= not out_35;
                          crc_coded_data(data_size+5):= not out_34;
                          crc_coded_data(data_size+6):= not out_33;
                          crc_coded_data(data_size+7):= not out_32;
                          crc_coded_data(data_size+8):= not out_31;
```

```
                        crc_coded_data(data_size+9):= not out_30;

                        crc_coded_data(data_size+10):= not out_29;
                        crc_coded_data(data_size+11):= not out_28;
                        crc_coded_data(data_size+12):= not out_27;
                        crc_coded_data(data_size+13):= not out_26;
                        crc_coded_data(data_size+14):= not out_25;
                        crc_coded_data(data_size+15):= not out_24;
                        crc_coded_data(data_size+16):= not out_23;
                        crc_coded_data(data_size+17):= not out_22;
                        crc_coded_data(data_size+18):= not out_21;
                        crc_coded_data(data_size+19):= not out_20;

                        crc_coded_data(data_size+20):= not out_19;
                        crc_coded_data(data_size+21):= not out_18;
                        crc_coded_data(data_size+22):= not out_17;
                        crc_coded_data(data_size+23):= not out_16;
                        crc_coded_data(data_size+24):= not out_15;
                        crc_coded_data(data_size+25):= not out_14;
                        crc_coded_data(data_size+26):= not out_13;
                        crc_coded_data(data_size+27):= not out_12;
                        crc_coded_data(data_size+28):= not out_11;
                        crc_coded_data(data_size+29):= not out_10;

                        crc_coded_data(data_size+30):= not out_09;
                        crc_coded_data(data_size+31):= not out_08;
                        crc_coded_data(data_size+32):= not out_07;
                        crc_coded_data(data_size+33):= not out_06;
                        crc_coded_data(data_size+34):= not out_05;
                        crc_coded_data(data_size+35):= not out_04;
                        crc_coded_data(data_size+36):= not out_03;
                        crc_coded_data(data_size+37):= not out_02;
                        crc_coded_data(data_size+38):= not out_01;
                        crc_coded_data(data_size+39):= not out_00;
                data_size:=data_size+40;

elsif poly_c="0110" then

                crc_coded_data(data_size+0):= not out_39;
                        crc_coded_data(data_size+1):= not out_38;
                        crc_coded_data(data_size+2):= not out_37;
                        crc_coded_data(data_size+3):= not out_36;
                        crc_coded_data(data_size+4):= not out_35;
                        crc_coded_data(data_size+5):= not out_34;
                        crc_coded_data(data_size+6):= not out_33;
                        crc_coded_data(data_size+7):= not out_32;
                        crc_coded_data(data_size+8):= not out_31;
                        crc_coded_data(data_size+9):= not out_30;
                        data_size:=data_size+10;
elsif poly_c="0111" then

                        crc_coded_data(data_size+0):= not out_39;
                        crc_coded_data(data_size+1):= not out_38;
                        crc_coded_data(data_size+2):= not out_37;
                        crc_coded_data(data_size+3):= not out_36;
                        crc_coded_data(data_size+4):= not out_35;
                        crc_coded_data(data_size+5):= not out_34;
                        crc_coded_data(data_size+6):= not out_33;
                        crc_coded_data(data_size+7):= not out_32;
                        crc_coded_data(data_size+8):= not out_31;
                        crc_coded_data(data_size+9):= not out_30;

                        crc_coded_data(data_size+10):= not out_29;
```

```
                        crc_coded_data(data_size+11):= not out_28;
                        crc_coded_data(data_size+12):= not out_27;
                        crc_coded_data(data_size+13):= not out_26;
                        crc_coded_data(data_size+14):= not out_25;
                        crc_coded_data(data_size+15):= not out_24;
                data_size:=data_size+16;

    elsif poly_c="1000" then

                        crc_coded_data(data_size+0):= not out_39;
                        crc_coded_data(data_size+1):= not out_38;
                        crc_coded_data(data_size+2):= not out_37;
                        crc_coded_data(data_size+3):= not out_36;
                        crc_coded_data(data_size+4):= not out_35;
                        crc_coded_data(data_size+5):= not out_34;
                        crc_coded_data(data_size+6):= not out_33;
                        crc_coded_data(data_size+7):= not out_32;
                data_size:=data_size+8;

    elsif poly_c="1001" then

                        crc_coded_data(data_size+0):= not out_39;
                        crc_coded_data(data_size+1):= not out_38;
                        crc_coded_data(data_size+2):= not out_37;
                        crc_coded_data(data_size+3):= not out_36;
                        crc_coded_data(data_size+4):= not out_35;
                        crc_coded_data(data_size+5):= not out_34;
                        crc_coded_data(data_size+6):= not out_33;
                        crc_coded_data(data_size+7):= not out_32;
                        crc_coded_data(data_size+8):= not out_31;
                        crc_coded_data(data_size+9):= not out_30;

                        crc_coded_data(data_size+10):= not out_29;
                        crc_coded_data(data_size+11):= not out_28;
                data_size:=data_size+12;

    end if;
else
--no CRC needed.
    crc_coded_data:=data_to_encode;

end if;


slave<='1';
read<='0';
write<='1';

wait for 10 ns;

write_counter:=0;
in_conv:='0';
rec_bit:='0';
s1_bit:='0';
s2_bit:='0';
s3_bit:='0';
s4_bit:='0';
s5_bit:='0';
s6_bit:='0';



if op_mode="10" then
```

```
--CRC only
    while write_counter<data_size loop
--flip data order for easier indexing (dummy)

        flip_counter:=0;
        while flip_counter<32 loop
            data(flip_counter)<=crc_coded_data(write_counter+flip_counter);
            flip_counter:=flip_counter+1;
        end loop;


        address<=conv_std_logic_vector((write_counter/32), 10);
        wait for 10 ns;
        write_counter:=write_counter+32;
    end loop;
else
--convolutional coding
    while write_counter<data_size loop
        if poly_v="0001" then
            in_conv:=crc_coded_data(write_counter);
            o0:=in_conv xor s3_bit xor s4_bit;--(10011)
            o1:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
        elsif poly_v="0010" then
            in_conv:=crc_coded_data(write_counter);
            o0:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
            o1:=in_conv xor s2_bit xor s4_bit;--(10101)
            o2:=in_conv xor s1_bit xor s2_bit xor s3_bit xor s4_bit;--(11111)
        elsif poly_v="0011" then
            in_conv:=crc_coded_data(write_counter);
            o0:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
            o1:=in_conv xor s2_bit xor s4_bit;--(10101)
            o2:=in_conv xor s1_bit xor s2_bit xor s3_bit xor s4_bit;--(11111)
            o3:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
            o4:=in_conv xor s2_bit xor s4_bit;--(10101)
            o5:=in_conv xor s1_bit xor s2_bit xor s3_bit xor s4_bit;--(11111)
        elsif poly_v="0100" then
            in_conv:=crc_coded_data(write_counter);
            o0:=in_conv xor s2_bit xor s3_bit xor s5_bit xor s6_bit;--(1011011)
            o1:=in_conv xor s1_bit xor s4_bit xor s6_bit;--(1100101)
            o2:=in_conv xor s1_bit xor s2_bit xor s3_bit xor s4_bit xor s6_bit;--(1111101)
        elsif poly_v="0101" then
            in_conv:=crc_coded_data(write_counter);
            o0:=in_conv xor s2_bit xor s3_bit xor s5_bit xor s6_bit;--(1011011)
            o1:=in_conv xor s1_bit xor s2_bit xor s3_bit xor s4_bit xor s6_bit;--(1111101)
        elsif poly_v="0110" then
            in_conv:=crc_coded_data(write_counter);
            o0:=in_conv xor s2_bit xor s3_bit xor s5_bit xor s6_bit;--(1011011)
            o1:=in_conv xor s1_bit xor s2_bit xor s3_bit xor s6_bit;--(1111001)
            o2:=in_conv xor s1_bit xor s4_bit xor s6_bit;--(1100101)
        elsif poly_v="0111" then
            rec_bit:=s1_bit xor s2_bit xor s3_bit xor s4_bit;--(X1111)
            in_conv:=crc_coded_data(write_counter) xor rec_bit;
            o0:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
            o1:=in_conv xor s2_bit xor s4_bit;--(10101)
            o2:=crc_coded_data(write_counter);
            o3:=crc_coded_data(write_counter);
        elsif poly_v="1000" then
            rec_bit:=s3_bit xor s4_bit;--(X0011)
            in_conv:=crc_coded_data(write_counter) xor rec_bit;
            o0:=crc_coded_data(write_counter);
            o1:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
        elsif poly_v="1001" then
            rec_bit:=s1_bit xor s2_bit xor s3_bit xor s4_bit;--(X1111)
```

```
            in_conv:=crc_coded_data(write_counter) xor rec_bit;
            o0:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
            o1:=in_conv xor s2_bit xor s4_bit;--(10101)
            o2:=crc_coded_data(write_counter);
        elsif poly_v="1010" then
            rec_bit:=s1_bit xor s2_bit xor s3_bit xor s4_bit;--(X1111)
            in_conv:=crc_coded_data(write_counter) xor rec_bit;
            o0:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
            o1:=in_conv xor s1_bit xor s3_bit xor s4_bit;--(11011)
            o2:=in_conv xor s2_bit xor s4_bit;--(10101)
            o3:=crc_coded_data(write_counter);
            o4:=crc_coded_data(write_counter);
        elsif poly_v="1011" then
            rec_bit:=s1_bit xor s2_bit xor s3_bit xor s4_bit xor s6_bit; --(X111101)
            in_conv:=crc_coded_data(write_counter) xor rec_bit;
            o0:=in_conv xor s2_bit xor s3_bit xor s5_bit xor s6_bit;--(1011011)
            o1:=in_conv xor s1_bit xor s4_bit xor s6_bit;--(1100101)
            o2:=crc_coded_data(write_counter);
            o3:=crc_coded_data(write_counter);
        elsif poly_v="1100" then
            rec_bit:=s1_bit xor s2_bit xor s3_bit xor s4_bit xor s6_bit; --(X111101)
            in_conv:=crc_coded_data(write_counter) xor rec_bit;
            o0:=in_conv xor s2_bit xor s3_bit xor s5_bit xor s6_bit;--(1011011)
            o1:=in_conv xor s2_bit xor s3_bit xor s5_bit xor s6_bit;--(1011011)
            o2:=in_conv xor s1_bit xor s4_bit xor s6_bit;--(1100101)
            o3:=crc_coded_data(write_counter);
            o4:=crc_coded_data(write_counter);
        elsif poly_v="1101" then
            rec_bit:=s2_bit xor s3_bit xor s5_bit xor s6_bit; --(X111101)
            in_conv:=crc_coded_data(write_counter) xor rec_bit;
            o0:=crc_coded_data(write_counter);
            o1:=in_conv xor s1_bit xor s4_bit xor s6_bit;--(1100101)
            o2:=in_conv xor s1_bit xor s2_bit xor s3_bit xor s4_bit xor s6_bit; --(1111101)
        end if;


-- binray to soft decision
        if soft_bit_no='0' then
            if o0='0' then
                data(3 downto 0)<="1000";
            else
                data(3 downto 0)<="0111";
            end if;

            if o1='0' then
                data(7 downto 4)<="1000";
            else
                data(7 downto 4)<="0111";
            end if;

            if o2='0' then
                data(11 downto 8)<="1000";
            else
                data(11 downto 8)<="0111";
            end if;

            if o3='0' then
                data(15 downto 12)<="1000";
            else
                data(15 downto 12)<="0111";
            end if;

            if o4='0' then
```

```
                data(19 downto 16)<="1000";
            else
                data(19 downto 16)<="0111";
            end if;

            if o5='0' then
                data(23 downto 20)<="1000";
            else
                data(23 downto 20)<="0111";
            end if;

            data(31 downto 24) <="00000000";
        elsif soft_bit_no='1' then
            if o0='0' then
                data(4 downto 0)<="10000";
            else
                data(4 downto 0)<="01111";
            end if;

            if o1='0' then
                data(9 downto 5)<="10000";
            else
                data(9 downto 5)<="01111";
            end if;

            if o2='0' then
                data(14 downto 10)<="10000";
            else
                data(14 downto 10)<="01111";
            end if;

            if o3='0' then
                data(19 downto 15)<="10000";
            else
                data(19 downto 15)<="01111";
            end if;

            if o4='0' then
                data(24 downto 20)<="10000";
            else
                data(24 downto 20)<="01111";
            end if;

            if o5='0' then
                data(29 downto 25)<="10000";
            else
                data(29 downto 25)<="01111";
            end if;
            data(31 downto 30)<="00";
        end if;

--sent to decoder input buffer
        address<=conv_std_logic_vector(write_counter, 10);

        wait for 10 ns;

--shift register update
        write_counter:=write_counter+1;
        s6_bit:=s5_bit;
        s5_bit:=s4_bit;
        s4_bit:=s3_bit;
        s3_bit:=s2_bit;
        s2_bit:=s1_bit;
```

```
        s1_bit:=in_conv;

    end loop;

end if;

--decoding parameter
    address<="1001100101" ;
    data(1 downto 0)<=op_mode;
    data(5 downto 2)<=poly_v;
    data(6)<=soft_bit_no;
    data(15 downto 7)<=threshold;
    data(19 downto 16)<=poly_c;
    data(29 downto 20)<=conv_std_logic_vector(data_size,10);
    wait for 10 ns;

slave<='0';
read<='0';
write<='0';
data<="00000000000000000000000000000000";

wait until intf'event and intf='1';
--decode finish
data<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
wait for 1 ns;
read_counter:=0;
wait for 10 ns;
slave<='1';
read<='1';

while read_counter<data_size loop
--read data from decoder
    address<=conv_std_logic_vector((read_counter/32), 10);

    wait for 10 ns;

    decoded_data(read_counter+31 downto read_counter):=data(31 downto 0);

    read_counter:=read_counter+32;

end loop;

if op_mode="10" or op_mode="11" then
--CRC code
    address<=conv_std_logic_vector((read_counter/32), 10);
    wait for 10 ns;
    read_counter:=read_counter+32;
    address<=conv_std_logic_vector((read_counter/32), 10);
end if;

compare_counter:=0;
different_flag:='0';
while compare_counter<data_size loop

--compare decoded data with encoded data
    different_flag:=different_flag or (crc_coded_data(compare_counter)
                    xor decoded_data(compare_counter));
    compare_counter:=compare_counter+1;
    wait for 1 ns;
end loop;

success<=not different_flag;
```

```
wait for 100 us;
end process;



---------------------------------------------------------------------------
---------------                                              ---------------
---------------                     shift   chain            ---------------
---------------                                              ---------------
---------------------------------------------------------------------------


stage00: process(poly_c, feedback, in_bit, clear, shift,clk, stage00_out)
variable up,left, reg00_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0101" then
    left:='0';
else
    left:='0';
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg00_in:='0';
            stage00_out<=reg00_in;
        else
            reg00_in:=up xor left;
            stage00_out<=reg00_in;
        end if;
    end if;
end if;

out_00<=stage00_out;
end process;



stage01: process(poly_c, feedback, in_bit, clear, shift,clk, stage00_out,stage01_out)
variable up,left, reg01_in: std_logic;
begin



if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg01_in:='0';
            stage01_out<=reg01_in;
        else
            reg01_in:=stage00_out;
            stage01_out<=reg01_in;
        end if;
    end if;
end if;
```

```
out_01<=stage01_out;
end process;



stage02: process(poly_c, feedback, in_bit, clear, shift,clk, stage01_out,stage02_out)
variable up,left, reg02_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg02_in:='0';
            stage02_out<=reg02_in;
        else
            reg02_in:=stage01_out;
            stage02_out<=reg02_in;
        end if;
    end if;
end if;

out_02<=stage02_out;
end process;



stage03: process(poly_c, feedback, in_bit, clear, shift,clk, stage03_out,stage02_out)
variable up,left, reg03_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage02_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg03_in:='0';
            stage03_out<=reg03_in;
        else
            reg03_in:=up xor left;
            stage03_out<=reg03_in;
        end if;
    end if;
end if;

out_03<=stage03_out;
end process;


stage04: process(poly_c, feedback, in_bit, clear, shift,clk, stage03_out,stage04_out)
variable up,left, reg04_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
```

```
            reg04_in:='0';
            stage04_out<=reg04_in;
        else
            reg04_in:=stage03_out;
            stage04_out<=reg04_in;
        end if;
    end if;
end if;

out_04<=stage04_out;
end process;



stage05: process(poly_c, feedback, in_bit, clear, shift,clk, stage05_out,stage04_out)
variable up,left, reg05_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg05_in:='0';
            stage05_out<=reg05_in;
        else
            reg05_in:=stage04_out;
            stage05_out<=reg05_in;
        end if;
    end if;
end if;

out_05<=stage05_out;
end process;


stage06: process(poly_c, feedback, in_bit, clear, shift,clk, stage06_out,stage05_out)
variable up,left, reg06_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg06_in:='0';
            stage06_out<=reg06_in;
        else
            reg06_in:=stage05_out;
            stage06_out<=reg06_in;
        end if;
    end if;
end if;

out_06<=stage06_out;
end process;


stage07: process(poly_c, feedback, in_bit, clear, shift,clk, stage07_out,stage06_out)
variable up,left, reg07_in: std_logic;
begin


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
```

```
            reg07_in:='0';
            stage07_out<=reg07_in;
        else
            reg07_in:=stage06_out;
            stage07_out<=reg07_in;
        end if;
    end if;
end if;

out_07<=stage07_out;
end process;



stage08: process(poly_c, feedback, in_bit, clear, shift,clk, stage08_out,stage07_out)
variable up,left, reg08_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg08_in:='0';
            stage08_out<=reg08_in;
        else
            reg08_in:=stage07_out;
            stage08_out<=reg08_in;
        end if;
    end if;
end if;

out_08<=stage08_out;
end process;


stage09: process(poly_c, feedback, in_bit, clear, shift,clk, stage09_out,stage08_out)
variable up,left, reg09_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg09_in:='0';
            stage09_out<=reg09_in;
        else
            reg09_in:=stage08_out;
            stage09_out<=reg09_in;
        end if;
    end if;
end if;

out_09<=stage09_out;
end process;

stage10: process(poly_c, feedback, in_bit, clear, shift,clk, stage10_out,stage09_out)
variable up,left, reg10_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg10_in:='0';
            stage10_out<=reg10_in;
```

```
        else
            reg10_in:=stage09_out;
            stage10_out<=reg10_in;
        end if;
    end if;
end if;

out_10<=stage10_out;
end process;


stage11: process(poly_c, feedback, in_bit, clear, shift,clk, stage11_out,stage10_out)
variable up,left, reg11_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg11_in:='0';
            stage11_out<=reg11_in;
        else
            reg11_in:=stage10_out;
            stage11_out<=reg11_in;
        end if;
    end if;
end if;

out_11<=stage11_out;
end process;


stage12: process(poly_c, feedback, in_bit, clear, shift,clk, stage12_out,stage11_out)
variable up,left, reg12_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg12_in:='0';
            stage12_out<=reg12_in;
        else
            reg12_in:=stage11_out;
            stage12_out<=reg12_in;
        end if;
    end if;
end if;

out_12<=stage12_out;
end process;


stage13: process(poly_c, feedback, in_bit, clear, shift,clk, stage13_out,stage12_out)
variable up,left, reg13_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg13_in:='0';
            stage13_out<=reg13_in;
        else
            reg13_in:=stage12_out;
            stage13_out<=reg13_in;
```

```
        end if;
    end if;
end if;

out_13<=stage13_out;
end process;

stage14: process(poly_c, feedback, in_bit, clear, shift,clk, stage14_out,stage13_out)
variable up,left, reg14_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg14_in:='0';
            stage14_out<=reg14_in;
        else
            reg14_in:=stage13_out;
            stage14_out<=reg14_in;
        end if;
    end if;
end if;

out_14<=stage14_out;
end process;


stage15: process(poly_c, feedback, in_bit, clear, shift,clk, stage15_out,stage14_out)
variable up,left, reg15_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg15_in:='0';
            stage15_out<=reg15_in;
        else
            reg15_in:=stage14_out;
            stage15_out<=reg15_in;
        end if;
    end if;
end if;

out_15<=stage15_out;
end process;


stage16: process(poly_c, feedback, in_bit, clear, shift,clk, stage16_out,stage15_out)
variable up,left, reg16_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg16_in:='0';
            stage16_out<=reg16_in;
        else
            reg16_in:=stage15_out;
            stage16_out<=reg16_in;
        end if;
    end if;
end if;
```

```
out_16<=stage16_out;
end process;


stage17: process(poly_c, feedback, in_bit, clear, shift,clk, stage17_out,stage16_out)
variable up,left, reg17_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;


left:=stage16_out;

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg17_in:='0';
            stage17_out<=reg17_in;
        else
            reg17_in:=up xor left;
            stage17_out<=reg17_in;
        end if;
    end if;
end if;

out_17<=stage17_out;
end process;



stage18: process(poly_c, feedback, in_bit, clear, shift,clk, stage18_out,stage17_out)
variable up,left, reg18_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg18_in:='0';
            stage18_out<=reg18_in;
        else
            reg18_in:=stage17_out;
            stage18_out<=reg18_in;
        end if;
    end if;
end if;

out_18<=stage18_out;
end process;


stage19: process(poly_c, feedback, in_bit, clear, shift,clk, stage19_out,stage18_out)
variable up,left, reg19_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg19_in:='0';
            stage19_out<=reg19_in;
```

```
        else
            reg19_in:=stage18_out;
            stage19_out<=reg19_in;
        end if;
    end if;
end if;

out_19<=stage19_out;
end process;


stage20: process(poly_c, feedback, in_bit, clear, shift,clk, stage20_out,stage19_out)
variable up,left, reg20_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg20_in:='0';
            stage20_out<=reg20_in;
        else
            reg20_in:=stage19_out;
            stage20_out<=reg20_in;
        end if;
    end if;
end if;

out_20<=stage20_out;
end process;


stage21: process(poly_c, feedback, in_bit, clear, shift,clk, stage21_out,stage20_out)
variable up,left, reg21_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg21_in:='0';
            stage21_out<=reg21_in;
        else
            reg21_in:=stage20_out;
            stage21_out<=reg21_in;
        end if;
    end if;
end if;

out_21<=stage21_out;
end process;


stage22: process(poly_c, feedback, in_bit, clear, shift,clk, stage22_out,stage21_out)
variable up,left, reg22_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg22_in:='0';
            stage22_out<=reg22_in;
        else
            reg22_in:=stage21_out;
            stage22_out<=reg22_in;
```

```
        end if;
    end if;
end if;

out_22<=stage22_out;
end process;



stage23: process(poly_c, feedback, in_bit, clear, shift,clk, stage23_out,stage22_out)
variable up,left, reg23_in: std_logic;
begin

if poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;


left:=stage22_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg23_in:='0';
            stage23_out<=reg23_in;
        else
            reg23_in:=up xor left;
            stage23_out<=reg23_in;
        end if;
    end if;
end if;

out_23<=stage23_out;
end process;




stage24: process(poly_c, feedback, in_bit, clear, shift,clk, stage24_out,stage23_out)
variable up,left, reg24_in: std_logic;
begin

if poly_c="0111" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0111" then
    left:='0';
else
    left:=stage23_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg24_in:='0';
            stage24_out<=reg24_in;
```

```
        else
            reg24_in:=up xor left;
            stage24_out<=reg24_in;
        end if;
    end if;
end if;

out_24<=stage24_out;
end process;



stage25: process(poly_c, feedback, in_bit, clear, shift,clk, stage25_out,stage24_out)
variable up,left, reg25_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg25_in:='0';
            stage25_out<=reg25_in;
        else
            reg25_in:=stage24_out;
            stage25_out<=reg25_in;
        end if;
    end if;
end if;

out_25<=stage25_out;
end process;


stage26: process(poly_c, feedback, in_bit, clear, shift,clk, stage26_out,stage25_out)
variable up,left, reg26_in: std_logic;
begin

if poly_c="0011"or poly_c="0101" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0011" then
    left:='0';
else
    left:=stage25_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg26_in:='0';
            stage26_out<=reg26_in;
        else
            reg26_in:=up xor left;
            stage26_out<=reg26_in;
        end if;
    end if;
end if;

out_26<=stage26_out;
end process;
```

```
stage27: process(poly_c, feedback, in_bit, clear, shift,clk, stage27_out,stage26_out)
variable up,left, reg27_in: std_logic;
begin

if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg27_in:='0';
            stage27_out<=reg27_in;
        else
            reg27_in:=stage26_out;
            stage27_out<=reg27_in;
        end if;
    end if;
end if;

out_27<=stage27_out;
end process;

stage28: process(poly_c, feedback, in_bit, clear, shift,clk, stage28_out,stage27_out)
variable up,left, reg28_in: std_logic;
begin

if poly_c="0011" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "1001" then
    left:='0';
else
    left:=stage27_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg28_in:='0';
            stage28_out<=reg28_in;
        else
            reg28_in:=up xor left;
            stage28_out<=reg28_in;
        end if;
    end if;
end if;

out_28<=stage28_out;
end process;


stage29: process(poly_c, feedback, in_bit, clear, shift,clk, stage29_out,stage28_out)
variable up,left, reg29_in: std_logic;
begin

if poly_c="0011"or poly_c="0111" then
    up:=feedback;
else
    up:='0';
end if;
```

```
left:=stage28_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg29_in:='0';
            stage29_out<=reg29_in;
        else
            reg29_in:=up xor left;
            stage29_out<=reg29_in;
        end if;
    end if;
end if;

out_29<=stage29_out;
end process;

stage30: process(poly_c, feedback, in_bit, clear, shift,clk, stage30_out,stage29_out)
variable up,left, reg30_in: std_logic;
begin

if poly_c="0110" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0110" then
    left:='0';
else
    left:=stage29_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg30_in:='0';
            stage30_out<=reg30_in;
        else
            reg30_in:=up xor left;
            stage30_out<=reg30_in;
        end if;
    end if;
end if;

out_30<=stage30_out;
end process;


stage31: process(poly_c, feedback, in_bit, clear, shift,clk, stage31_out,stage30_out)
variable up,left, reg31_in: std_logic;
begin

if poly_c="0011" then
    up:=feedback;
else
    up:='0';
end if;
```

```
left:=stage30_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg31_in:='0';
            stage31_out<=reg31_in;
        else
            reg31_in:=up xor left;
            stage31_out<=reg31_in;
        end if;
    end if;
end if;

out_31<=stage31_out;
end process;

stage32: process(poly_c, feedback, in_bit, clear, shift,clk, stage32_out,stage31_out)
variable up,left, reg32_in: std_logic;
begin

if poly_c="0001"or poly_c="0110" or poly_c="1000" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0001" or poly_c = "1000" then
    left:='0';
else
    left:=stage31_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg32_in:='0';
            stage32_out<=reg32_in;
        else
            reg32_in:=up xor left;
            stage32_out<=reg32_in;
        end if;
    end if;
end if;

out_32<=stage32_out;
end process;


stage33: process(poly_c, feedback, in_bit, clear, shift,clk, stage33_out,stage32_out)
variable up,left, reg33_in: std_logic;
begin

if poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;
```

```
left:=stage32_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg33_in:='0';
            stage33_out<=reg33_in;
        else
            reg33_in:=up xor left;
            stage33_out<=reg33_in;
        end if;
    end if;
end if;

out_33<=stage33_out;
end process;


stage34: process(poly_c, feedback, in_bit, clear, shift,clk, stage34_out,stage33_out)
variable up,left, reg34_in: std_logic;
begin

if poly_c="0001" or poly_c="0100" or poly_c="0110" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0100" then
    left:='0';
else
    left:=stage33_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg34_in:='0';
            stage34_out<=reg34_in;
        else
            reg34_in:=up xor left;
            stage34_out<=reg34_in;
        end if;
    end if;
end if;

out_34<=stage34_out;
end process;


stage35: process(poly_c, feedback, in_bit, clear, shift,clk, stage35_out,stage34_out)
variable up,left, reg35_in: std_logic;
begin

if poly_c="0001" or poly_c="0100" or poly_c="0110" or poly_c="1000" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage34_out;
```

```
if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg35_in:='0';
            stage35_out<=reg35_in;
        else
            reg35_in:=up xor left;
            stage35_out<=reg35_in;
        end if;
    end if;
end if;

out_35<=stage35_out;
end process;

stage36: process(poly_c, feedback, in_bit, clear, shift,clk, stage36_out,stage35_out)
variable up,left, reg36_in: std_logic;
begin

if poly_c="0001" or poly_c="0100" or poly_c="0110" or poly_c="0111" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage35_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg36_in:='0';
            stage36_out<=reg36_in;
        else
            reg36_in:=up xor left;
            stage36_out<=reg36_in;
        end if;
    end if;
end if;

out_36<=stage36_out;
end process;

stage37: process(poly_c, feedback, in_bit, clear, shift,clk, stage37_out,stage36_out)
variable up,left, reg37_in: std_logic;
begin

if poly_c="0010"or poly_c="0100" then
    up:=feedback;
else
    up:='0';
end if;

if poly_c = "0010" then
    left:='0';
else
    left:=stage36_out;
end if;


if shift='1' then
    if clk'event and clk='1' then
```

```
        if clear = '1' then
            reg37_in:='0';
            stage37_out<=reg37_in;
        else
            reg37_in:=up xor left;
            stage37_out<=reg37_in;
        end if;
    end if;
end if;

out_37<=stage37_out;
end process;

stage38: process(poly_c, feedback, in_bit, clear, shift,clk, stage38_out,stage37_out)
variable up,left, reg38_in: std_logic;
begin

if poly_c="0010" or poly_c="0110" or poly_c="1000" or poly_c="1001"  then
    up:=feedback;
else
    up:='0';
end if;


left:=stage37_out;


if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg38_in:='0';
            stage38_out<=reg38_in;
        else
            reg38_in:=up xor left;
            stage38_out<=reg38_in;
        end if;
    end if;
end if;

out_38<=stage38_out;
end process;


stage39: process(poly_c, feedback, in_bit, clear, shift,clk, stage39_out,stage38_out)
variable up,left, reg39_in: std_logic;
begin

if poly_c="0011" or poly_c="0100" or poly_c="1001" then
    up:=feedback;
else
    up:='0';
end if;

left:=stage38_out;



if shift='1' then
    if clk'event and clk='1' then
        if clear = '1' then
            reg39_in:='0';
            stage39_out<=reg39_in;
        else
```

```
            reg39_in:=up xor left;
            stage39_out<=reg39_in;
        end if;
    end if;
end if;

out_39<=stage39_out;
end process;



stage40: process(stage39_out, in_bit)

begin
feedback<=stage39_out xor in_bit;
end process;
end behavioral;
```

A.2.2 Test bench entity

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--test bench entity
library ieee;
library ti_custom;

use ieee.std_logic_1164.all;
use ti_custom.all;



entity decoder_bench is
end decoder_bench;

architecture structural of decoder_bench is

component decoder_stim
port(
    reset       : in std_logic;
    clk     : in std_logic;

    slave       : out std_logic;
    read        : out std_logic;
    write       : out std_logic;
    address     : out std_logic_vector(9 downto 0);
    success : out std_logic;

    intf        : in std_logic;

    data        : inout std_logic_vector(31 downto 0)
);
end component;

component decoder
port(
    reset       : in std_logic;
    clk     : in std_logic;
```

```
    slave       : in std_logic;
    read        : in std_logic;
    write       : in std_logic;
    address     : in std_logic_vector(9 downto 0);

    intf        : out std_logic;

    data        : inout std_logic_vector(31 downto 0)


);
end component;

component clock_gen
port(
clk : out std_logic;
reset : out std_logic
);
end component;

signal  reset       : std_logic;
signal  clk     : std_logic;

signal  slave       : std_logic;
signal  read        : std_logic;
signal  write       : std_logic;
signal  address     : std_logic_vector(9 downto 0);

signal  intf        : std_logic;

signal  data        : std_logic_vector(31 downto 0);
signal  success : std_logic;
begin

tester:  decoder_stim port map (reset,clk,slave,read,write,address,success,intf,data);

dut: decoder port map (reset,clk,slave,read,write,address,intf,data);

clk_reset: clock_gen port map (clk, reset);

end structural;
```

A.2.3 Clock and reset generator

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

--clock and reset signal generator

library ieee;
use ieee.std_logic_1164.all;

entity clock_gen is
port(
clk : out std_logic;
reset : out std_logic
);
```

```
end clock_gen;

architecture behavioral of clock_gen is
begin

reset<='1', '0' after 30 ns;

clock_driver: process
constant Tclk : time := 10 ns;
begin
    clk<='0';
    wait for Tclk;
    loop
        clk<='1', '0' after Tclk/2;
        wait for Tclk;
        end loop;
end process;
end behavioral;
```

A.2.4 Test configuration file

```
--configuration file for test


configuration decoder_test_bench_config of decoder_bench is
    for structural
        for tester: decoder_stim
            use entity work.decoder_stim(behavioral);
        end for;
        for dut : decoder
            use entity work.decoder(structural);
            for structural
                for controller: decoder_control_unit
                    use entity work.decoder_control_unit(behavioral);
                end for;
                for trellis: viterbi_unit
                    use entity work.viterbi_unit(structural);
                    for structural
                        for controller: viterbi_control_unit
                            use entity work.viterbi_control_unit(behavioral);
                        end for;
                        for TB_memory : tb_ram
                            use entity work.tb_ram (structural);
                        end for;
                        for SB_decoder: soft_bit_decoder
                            use entity work.soft_bit_decoder(behavioral);
                        end for;
                        for update_index: find_new_index
                            use entity work.find_new_index(behavioral);
                        end for;
                        for PE_pipelined: PE_unit
                            use entity work.PE_unit(structural);
                            for structural
                                for even_mem: bm_ram1
                                    use entity work.bm_ram1(structural);
                                end for;
                                for odd_mem: bm_ram2
                                    use entity work.bm_ram2(structural);
                                end for;
                                for PM_fetch: read_PM
                                    use entity work.read_PM(behavioral);
```

```
                                end for;
                                for T_unit: Path_prune
                                    use entity work.Path_prune(behavioral);
                                end for;
                                for Encoder: ACS_cc
                                    use entity work.ACS_cc(behavioral);
                                end for;
                                for pipeline1 : pip1
                                    use entity work.pip1(behavioral);
                                end for;
                                for branch_metric: ACS_bm
                                    use entity work.ACS_bm(behavioral);
                                end for;
                                for Path_merge: ACS_pm
                                    use entity work.ACS_pm(behavioral);
                                end for;
                                for TB_word_buffer: TB_update
                                    use entity work.TB_update(structural);
                                    for structural
                                        for clock_gates: write_enable
                                            use entity work.write_enable(behavioral);
                                        end for;
                                        for word_latch: decision_bits
                                            use entity work.decision_bits(behavioral);
                                        end for;
                                    end for;
                                end for;
                                for pipeline2 : pip2
                                    use entity work.pip2(behavioral);
                                end for;
                                for find_max : max
                                    use entity work.max(behavioral);
                                end for;
                                for PM_store: write_PM
                                    use entity work.write_PM(behavioral);
                                end for;
                                for pipeline3: pip3
                                    use entity work.pip3(behavioral);
                                end for;
                            end for;
                        end for;
                    end for;
                end for;
                for CRC_check: cyclic_decoder
                    use entity work.cyclic_decoder(structural);
                    for structural
                        for controller: CRC_control_unit
                            use entity work.CRC_control_unit(behavioral);
                        end for;
                        for CRC_chain: shift_chain
                            use entity work.shift_chain(behavioral);
                        end for;
                    end for;
                end for;
                for input_buffer: buffer_ram
                    use entity work.buffer_ram (structural);
                end for;
            end for;
        end for;
        for clk_reset: clock_gen use entity work.clock_gen(behavioral);
        end for;
    end for;
end decoder_test_bench_config;
```

A.3 Gray code test

A.3.1 Entity declaration

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- structure declaration of the 2*64-bit storage
entity invalid_bit_backup is
port
(
        clk                 : in  std_logic;
        step_write              : in    integer range 0 to 31;
        step_read              : in    integer range 0 to 31;
        selection               :in  std_logic;

        PM_up_invalid       : in std_logic;
        PM_down_invalid     : in std_logic;

        PM_invalid_up_out   : out std_logic;
        PM_invalid_down_out   : out std_logic



);
end invalid_bit_backup;

architecture structural of invalid_bit_backup is

component write_gray_code
port
(
        step                : in    integer range 0 to 31;
        step_gray           : out   integer range 0 to 31
);
end component;

component write_enable
port
(
        step_gray           : in   integer range 0 to 31;
        clock_enable        : out  std_logic_vector(0 to 31)

);
end component;


component write_demux
port
(
```

```
        selection               :in  std_logic;
        clock_enable_in        : in  std_logic_vector(0 to 31);
        clock_enable_out_1        : out  std_logic_vector(0 to 31);
        clock_enable_out_2        : out  std_logic_vector(0 to 31)

);
end component;

component invalid_flag_1
port
(
    clk                 : in  std_logic;
    clock_enable        : in  std_logic_vector(0 to 31);
    PM_up_invalid       : in std_logic;
    PM_down_invalid     : in std_logic;
    invalid_bit_vector  : out std_logic_vector(0 to 63)
);
end component;

component invalid_flag_2
port
(
    clk                 : in  std_logic;
    clock_enable        : in  std_logic_vector(0 to 31);
    PM_up_invalid       : in std_logic;
    PM_down_invalid     : in std_logic;
    invalid_bit_vector  : out std_logic_vector(0 to 63)
);
end component;

component read_mux
port
(
        selection               :in  std_logic;
        invalid_bit_vector_in_1  : in std_logic_vector(0 to 63);
        invalid_bit_vector_in_2  : in std_logic_vector(0 to 63);
        invalid_bit_vector_out  : out std_logic_vector(0 to 63)
);
end component;

component read_gray_code
port
(
        step                : in    integer range 0 to 31;
        step_gray           : out   integer range 0 to 31
);
end component;

component read_bits
port
(
        invalid_bit_vector  : in std_logic_vector(0 to 63);
        step_gray           : in integer range 0 to 31;

        PM_invalid_up_out   : out std_logic;
        PM_invalid_down_out   : out std_logic
);
end component;

signal step_gray_w : integer range 0 to 31;
signal step_gray_r : integer range 0 to 31;
signal clock_enable: std_logic_vector(0 to 31);
signal clock_enable_out_1: std_logic_vector(0 to 31);
```

```
signal clock_enable_out_2: std_logic_vector(0 to 31);
signal invalid_bit_vector_1: std_logic_vector(0 to 63);
signal invalid_bit_vector_2: std_logic_vector(0 to 63);
signal invalid_bit_vector_out: std_logic_vector(0 to 63);

begin

gray_code_write: write_gray_code port map (step_write,step_gray_w);
enable_write: write_enable port map (step_gray_w, clock_enable);
demux: write_demux port map (selection, clock_enable,clock_enable_out_1,
    clock_enable_out_2);
storage_1: invalid_flag_1 port map (clk,clock_enable_out_1,PM_up_invalid,
    PM_down_invalid, invalid_bit_vector_1);
storage_2: invalid_flag_2 port map (clk,clock_enable_out_2,PM_up_invalid,
    PM_down_invalid, invalid_bit_vector_2);
mux: read_mux port map(selection,invalid_bit_vector_1,invalid_bit_vector_2,
    invalid_bit_vector_out);
gray_code_read: read_gray_code port map (step_read,step_gray_r);
invalid_bit_output: read_bits port map (invalid_bit_vector_out, step_gray_r,
    PM_invalid_up_out,PM_invalid_down_out);


end structural;
```

A.3.2 Gray code read

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;


--converts the binary code to gray code

entity  read_gray_code is
port
(
        step                : in    integer range 0 to 31;
        step_gray           : out   integer range 0 to 31
);
end read_gray_code;

architecture behavioral of read_gray_code is
begin

main: process (step)
variable step_v : std_logic_vector (4 downto 0);
variable step_gray_v : std_logic_vector (4 downto 0);
begin

step_v:=conv_std_logic_vector(step, 5);

-- conversion algorithm
step_gray_v(4):=step_v(4);
step_gray_v(3):=step_v(4) xor step_v(3);
step_gray_v(2):=step_v(3) xor step_v(2);
step_gray_v(1):=step_v(2) xor step_v(1);
```

```
step_gray_v(0):=step_v(1) xor step_v(0);

step_gray<=conv_integer(unsigned(step_gray_v));

end process;

end behavioral;
```

A.3.3 Gray code write

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- binary to Gray conversion

entity  write_gray_code is
port
(
        step                : in    integer range 0 to 31;
        step_gray           : out   integer range 0 to 31
);
end write_gray_code;

architecture behavioral of write_gray_code is
begin

main: process (step)
variable step_v : std_logic_vector (4 downto 0);
variable step_gray_v : std_logic_vector (4 downto 0);
begin

step_v:=conv_std_logic_vector(step, 5);

-- coding algorithm

step_gray_v(4):=step_v(4);
step_gray_v(3):=step_v(4) xor step_v(3);
step_gray_v(2):=step_v(3) xor step_v(2);
step_gray_v(1):=step_v(2) xor step_v(1);
step_gray_v(0):=step_v(1) xor step_v(0);

step_gray<=conv_integer(unsigned(step_gray_v));

end process;

end behavioral;
```

A.3.4 Write enable

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- 1-to-32 demux circuit that generates the clock gate
entity write_enable is
port
(
        step_gray           : in   integer range 0 to 31;
        clock_enable        : out  std_logic_vector(0 to 31)

);
end write_enable;

architecture behavioral of write_enable is
begin

main: process (step_gray)
variable i : integer range 0 to 31;
begin

for i in 0 to 31 loop
    if i= step_gray then
        clock_enable(i) <= '1' ;
    else
        clock_enable(i) <= '0' ;
    end if;
end loop;

end process;

end behavioral;
```

A.3.5 Write DEMUX

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- 32 1-to-2 demux
entity write_demux is
port
(
        selection               :in  std_logic;
        clock_enable_in        : in  std_logic_vector(0 to 31);
        clock_enable_out_1        : out  std_logic_vector(0 to 31);
        clock_enable_out_2        : out  std_logic_vector(0 to 31)
```

```
);
end write_demux;

architecture behavioral of write_demux is
begin

main: process (selection, clock_enable_in)
begin

-- demultiplexing
    if selection='1' then
        clock_enable_out_1<=clock_enable_in;
        clock_enable_out_2<="00000000000000000000000000000000";
    else
        clock_enable_out_2<=clock_enable_in;
        clock_enable_out_1<="00000000000000000000000000000000";
    end if;

end process;

end behavioral;
```

A.3.6 Storage 1

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- one of the 64-bit storages
entity invalid_flag_1 is
port
(
    clk                 : in  std_logic;
    clock_enable        : in  std_logic_vector(0 to 31);
    PM_up_invalid       : in std_logic;
    PM_down_invalid     : in std_logic;
    invalid_bit_vector  : out std_logic_vector(0 to 63)
);
end invalid_flag_1;

architecture behavioral of invalid_flag_1 is
    signal gated_clock  : std_logic_vector(0 to 63);
    signal gclk00       : std_logic;
    signal gclk01       : std_logic;
    signal gclk02       : std_logic;
    signal gclk03       : std_logic;
    signal gclk04       : std_logic;
    signal gclk05       : std_logic;
    signal gclk06       : std_logic;
    signal gclk07       : std_logic;
    signal gclk08       : std_logic;
    signal gclk09       : std_logic;
```

```
    signal gclk10       : std_logic;
    signal gclk11       : std_logic;
    signal gclk12       : std_logic;
    signal gclk13       : std_logic;
    signal gclk14       : std_logic;
    signal gclk15       : std_logic;
    signal gclk16       : std_logic;
    signal gclk17       : std_logic;
    signal gclk18       : std_logic;
    signal gclk19       : std_logic;

    signal gclk20       : std_logic;
    signal gclk21       : std_logic;
    signal gclk22       : std_logic;
    signal gclk23       : std_logic;
    signal gclk24       : std_logic;
    signal gclk25       : std_logic;
    signal gclk26       : std_logic;
    signal gclk27       : std_logic;
    signal gclk28       : std_logic;
    signal gclk29       : std_logic;

    signal gclk30       : std_logic;
    signal gclk31       : std_logic;
    signal gclk32       : std_logic;
    signal gclk33       : std_logic;
    signal gclk34       : std_logic;
    signal gclk35       : std_logic;
    signal gclk36       : std_logic;
    signal gclk37       : std_logic;
    signal gclk38       : std_logic;
    signal gclk39       : std_logic;

    signal gclk40       : std_logic;
    signal gclk41       : std_logic;
    signal gclk42       : std_logic;
    signal gclk43       : std_logic;
    signal gclk44       : std_logic;
    signal gclk45       : std_logic;
    signal gclk46       : std_logic;
    signal gclk47       : std_logic;
    signal gclk48       : std_logic;
    signal gclk49       : std_logic;

    signal gclk50       : std_logic;
    signal gclk51       : std_logic;
    signal gclk52       : std_logic;
    signal gclk53       : std_logic;
    signal gclk54       : std_logic;
    signal gclk55       : std_logic;
    signal gclk56       : std_logic;
    signal gclk57       : std_logic;
    signal gclk58       : std_logic;
    signal gclk59       : std_logic;

    signal gclk60       : std_logic;
    signal gclk61       : std_logic;
    signal gclk62       : std_logic;
    signal gclk63       : std_logic;

begin
```

```
clock_gate: process (clock_enable, clk)
variable i :integer range 0 to 31;
variable i_v :std_logic_vector(4 downto 0);
variable i_tail :std_logic;
variable double_i_G :integer range 0 to 63;
variable double_i_G_inc :integer range 0 to 63;
begin

-- setup connection of the clock gate signal
-- ONLY WIRES. NO EFFICIENCY ISSUE

for i in 0 to 31 loop
    i_v:=conv_std_logic_vector(i,5);
    i_tail:=i_v(0) xor i_v(1) xor i_v(2) xor i_v(3) xor i_v(4);
    double_i_G:=conv_integer(unsigned(i_v & i_tail));
    double_i_G_inc:=conv_integer(unsigned(i_v & not i_tail));
    gated_clock(double_i_G)<=clk and clock_enable(i);
    gated_clock(double_i_G_inc)<=clk and clock_enable(i);
end loop;

--stored_invalid_bit<=gated_clock;

end process;

clk_conv: process (gated_clock)
begin

--Dummy connections for DPFLI
--not sure if work

    gclk00<=gated_clock(00);
    gclk01<=gated_clock(01);
    gclk02<=gated_clock(02);
    gclk03<=gated_clock(03);
    gclk04<=gated_clock(04);
    gclk05<=gated_clock(05);
    gclk06<=gated_clock(06);
    gclk07<=gated_clock(07);
    gclk08<=gated_clock(08);
    gclk09<=gated_clock(09);

    gclk10<=gated_clock(10);
    gclk11<=gated_clock(11);
    gclk12<=gated_clock(12);
    gclk13<=gated_clock(13);
    gclk14<=gated_clock(14);
    gclk15<=gated_clock(15);
    gclk16<=gated_clock(16);
    gclk17<=gated_clock(17);
    gclk18<=gated_clock(18);
    gclk19<=gated_clock(19);

    gclk20<=gated_clock(20);
    gclk21<=gated_clock(21);
    gclk22<=gated_clock(22);
    gclk23<=gated_clock(23);
    gclk24<=gated_clock(24);
    gclk25<=gated_clock(25);
    gclk26<=gated_clock(26);
    gclk27<=gated_clock(27);
    gclk28<=gated_clock(28);
    gclk29<=gated_clock(29);
```

```
    gclk30<=gated_clock(30);
    gclk31<=gated_clock(31);
    gclk32<=gated_clock(32);
    gclk33<=gated_clock(33);
    gclk34<=gated_clock(34);
    gclk35<=gated_clock(35);
    gclk36<=gated_clock(36);
    gclk37<=gated_clock(37);
    gclk38<=gated_clock(38);
    gclk39<=gated_clock(39);

    gclk40<=gated_clock(40);
    gclk41<=gated_clock(41);
    gclk42<=gated_clock(42);
    gclk43<=gated_clock(43);
    gclk44<=gated_clock(44);
    gclk45<=gated_clock(45);
    gclk46<=gated_clock(46);
    gclk47<=gated_clock(47);
    gclk48<=gated_clock(48);
    gclk49<=gated_clock(49);

    gclk50<=gated_clock(50);
    gclk51<=gated_clock(51);
    gclk52<=gated_clock(52);
    gclk53<=gated_clock(53);
    gclk54<=gated_clock(54);
    gclk55<=gated_clock(55);
    gclk56<=gated_clock(56);
    gclk57<=gated_clock(57);
    gclk58<=gated_clock(58);
    gclk59<=gated_clock(59);

    gclk60<=gated_clock(60);
    gclk61<=gated_clock(61);
    gclk62<=gated_clock(62);
    gclk63<=gated_clock(63);

end process;

--ALL the latches and clock gates
    latch00 : process(gclk00)
    begin
    if gclk00'event and gclk00 ='1' then
        invalid_bit_vector(0)<=PM_up_invalid;
    end if;
    end process;

    latch01 : process(gclk01)
    begin
    if gclk01'event and gclk01 ='1' then
        invalid_bit_vector(1)<=PM_down_invalid;
    end if;
    end process;

    latch02 : process(gclk02)
    begin
    if gclk02'event and gclk02 ='1' then
        invalid_bit_vector(2)<=PM_down_invalid;
    end if;
    end process;

    latch03 : process(gclk03)
```

```
begin
if gclk03'event and gclk03 ='1' then
    invalid_bit_vector(3)<=PM_up_invalid;
end if;
end process;

latch04 : process(gclk04)
begin
if gclk04'event and gclk04 ='1' then
    invalid_bit_vector(4)<=PM_down_invalid;
end if;
end process;

latch05 : process(gclk05)
begin
if gclk05'event and gclk05='1' then
    invalid_bit_vector(5)<=PM_up_invalid;
end if;
end process;

latch06 : process(gclk06)
begin
if gclk06'event and gclk06 ='1' then
    invalid_bit_vector(6)<=PM_up_invalid;
end if;
end process;

latch07 : process(gclk07)
begin
if gclk07'event and gclk07 ='1' then
    invalid_bit_vector(7)<=PM_down_invalid;
end if;
end process;

latch08 : process(gclk08)
begin
if gclk08'event and gclk08 ='1' then
    invalid_bit_vector(8)<=PM_down_invalid;
end if;
end process;

latch09 : process(gclk09)
begin
if gclk09'event and gclk09 ='1' then
    invalid_bit_vector(9)<=PM_up_invalid;
end if;
end process;

latch10 : process(gclk10)
begin
if gclk10'event and gclk10 ='1' then
    invalid_bit_vector(10)<=PM_up_invalid;
end if;
end process;

latch11 : process(gclk11)
begin
if gclk11'event and gclk11 ='1' then
    invalid_bit_vector(11)<=PM_down_invalid;
end if;
end process;

latch12 : process(gclk12)
```

```
begin
if gclk12'event and gclk12 ='1' then
    invalid_bit_vector(12)<=PM_up_invalid;
end if;
end process;

latch13 : process(gclk13)
begin
if gclk13'event and gclk13 ='1' then
    invalid_bit_vector(13)<=PM_down_invalid;
end if;
end process;

latch14 : process(gclk14)
begin
if gclk14'event and gclk14='1' then
    invalid_bit_vector(14)<=PM_down_invalid;
end if;
end process;

latch15 : process(gclk15)
begin
if gclk15'event and gclk15 ='1' then
    invalid_bit_vector(15)<=PM_up_invalid;
end if;
end process;

latch16 : process(gclk16)
begin
if gclk16'event and gclk16 ='1' then
    invalid_bit_vector(16)<=PM_down_invalid;
end if;
end process;

latch17 : process(gclk17)
begin
if gclk17'event and gclk17 ='1' then
    invalid_bit_vector(17)<=PM_up_invalid;
end if;
end process;

latch18 : process(gclk18)
begin
if gclk18'event and gclk18 ='1' then
    invalid_bit_vector(18)<=PM_up_invalid;
end if;
end process;

latch19 : process(gclk19)
begin
if gclk19'event and gclk19 ='1' then
    invalid_bit_vector(19)<=PM_down_invalid;
end if;
end process;

latch20 : process(gclk20)
begin
if gclk20'event and gclk20 ='1' then
    invalid_bit_vector(20)<=PM_up_invalid;
end if;
end process;

latch21 : process(gclk21)
```

```
begin
if gclk21'event and gclk21 ='1' then
    invalid_bit_vector(21)<=PM_down_invalid;
end if;
end process;

latch22 : process(gclk22)
begin
if gclk22'event and gclk22='1' then
    invalid_bit_vector(22)<=PM_down_invalid;
end if;
end process;

latch23 : process(gclk23)
begin
if gclk23'event and gclk23 ='1' then
    invalid_bit_vector(23)<=PM_up_invalid;
end if;
end process;

latch24 : process(gclk24)
begin
if gclk24'event and gclk24 ='1' then
    invalid_bit_vector(24)<=PM_up_invalid;
end if;
end process;

latch25 : process(gclk25)
begin
if gclk25'event and gclk25 ='1' then
    invalid_bit_vector(25)<=PM_down_invalid;
end if;
end process;

latch26 : process(gclk26)
begin
if gclk26'event and gclk26='1' then
    invalid_bit_vector(26)<=PM_down_invalid;
end if;
end process;

latch27 : process(gclk27)
begin
if gclk27'event and gclk27 ='1' then
    invalid_bit_vector(27)<=PM_up_invalid;
end if;
end process;

latch28 : process(gclk28)
begin
if gclk28'event and gclk28 ='1' then
    invalid_bit_vector(28)<=PM_down_invalid;
end if;
end process;

latch29 : process(gclk29)
begin
if gclk29'event and gclk29 ='1' then
    invalid_bit_vector(29)<=PM_up_invalid;
end if;
end process;

latch30 : process(gclk30)
```

```
begin
if gclk30'event and gclk30 ='1' then
    invalid_bit_vector(30)<=PM_up_invalid;
end if;
end process;

latch31 : process(gclk31)
begin
if gclk31'event and gclk31 ='1' then
    invalid_bit_vector(31)<=PM_down_invalid;
end if;
end process;


latch32 : process(gclk32)
begin
if gclk32'event and gclk32 ='1' then
    invalid_bit_vector(32)<=PM_down_invalid;
end if;
end process;

latch33 : process(gclk33)
begin
if gclk33'event and gclk33 ='1' then
    invalid_bit_vector(33)<=PM_up_invalid;
end if;
end process;

latch34 : process(gclk34)
begin
if gclk34'event and gclk34 ='1' then
    invalid_bit_vector(34)<=PM_up_invalid;
end if;
end process;

latch35 : process(gclk35)
begin
if gclk35'event and gclk35 ='1' then
    invalid_bit_vector(35)<=PM_down_invalid;
end if;
end process;

latch36 : process(gclk36)
begin
if gclk36'event and gclk36 ='1' then
    invalid_bit_vector(36)<=PM_up_invalid;
end if;
end process;

latch37 : process(gclk37)
begin
if gclk37'event and gclk37='1' then
    invalid_bit_vector(37)<=PM_down_invalid;
end if;
end process;

latch38 : process(gclk38)
begin
if gclk38'event and gclk38 ='1' then
    invalid_bit_vector(38)<=PM_down_invalid;
end if;
end process;
```

```
latch39 : process(gclk39)
begin
if gclk39'event and gclk39 ='1' then
    invalid_bit_vector(39)<=PM_up_invalid;
end if;
end process;

latch40 : process(gclk40)
begin
if gclk40'event and gclk40 ='1' then
    invalid_bit_vector(40)<=PM_up_invalid;
end if;
end process;

latch41 : process(gclk41)
begin
if gclk41'event and gclk41 ='1' then
    invalid_bit_vector(41)<=PM_down_invalid;
end if;
end process;

latch42 : process(gclk42)
begin
if gclk42'event and gclk42 ='1' then
    invalid_bit_vector(42)<=PM_down_invalid;
end if;
end process;

latch43 : process(gclk43)
begin
if gclk43'event and gclk43 ='1' then
    invalid_bit_vector(43)<=PM_up_invalid;
end if;
end process;

latch44 : process(gclk44)
begin
if gclk44'event and gclk44 ='1' then
    invalid_bit_vector(44)<=PM_down_invalid;
end if;
end process;

latch45 : process(gclk45)
begin
if gclk45'event and gclk45 ='1' then
    invalid_bit_vector(45)<=PM_up_invalid;
end if;
end process;

latch46 : process(gclk46)
begin
if gclk46'event and gclk46='1' then
    invalid_bit_vector(46)<=PM_up_invalid;
end if;
end process;

latch47 : process(gclk47)
begin
if gclk47'event and gclk47 ='1' then
    invalid_bit_vector(47)<=PM_down_invalid;
end if;
end process;
```

```
latch48 : process(gclk48)
begin
if gclk48'event and gclk48 ='1' then
    invalid_bit_vector(48)<=PM_up_invalid;
end if;
end process;

latch49 : process(gclk49)
begin
if gclk49'event and gclk49 ='1' then
    invalid_bit_vector(49)<=PM_down_invalid;
end if;
end process;

latch50 : process(gclk50)
begin
if gclk50'event and gclk50 ='1' then
    invalid_bit_vector(50)<=PM_down_invalid;
end if;
end process;

latch51 : process(gclk51)
begin
if gclk51'event and gclk51 ='1' then
    invalid_bit_vector(51)<=PM_up_invalid;
end if;
end process;

latch52 : process(gclk52)
begin
if gclk52'event and gclk52 ='1' then
    invalid_bit_vector(52)<=PM_down_invalid;
end if;
end process;

latch53 : process(gclk53)
begin
if gclk53'event and gclk53 ='1' then
    invalid_bit_vector(53)<=PM_up_invalid;
end if;
end process;

latch54 : process(gclk54)
begin
if gclk54'event and gclk54='1' then
    invalid_bit_vector(54)<=PM_up_invalid;
end if;
end process;

latch55 : process(gclk55)
begin
if gclk55'event and gclk55 ='1' then
    invalid_bit_vector(55)<=PM_down_invalid;
end if;
end process;

latch56 : process(gclk56)
begin
if gclk56'event and gclk56 ='1' then
    invalid_bit_vector(56)<=PM_down_invalid;
end if;
end process;
```

```
    latch57 : process(gclk57)
    begin
    if gclk57'event and gclk57 ='1' then
        invalid_bit_vector(57)<=PM_up_invalid;
    end if;
    end process;

    latch58 : process(gclk58)
    begin
    if gclk58'event and gclk58='1' then
        invalid_bit_vector(58)<=PM_up_invalid;
    end if;
    end process;

    latch59 : process(gclk59)
    begin
    if gclk59'event and gclk59 ='1' then
        invalid_bit_vector(59)<=PM_down_invalid;
    end if;
    end process;

    latch60 : process(gclk60)
    begin
    if gclk60'event and gclk60 ='1' then
        invalid_bit_vector(60)<=PM_up_invalid;
    end if;
    end process;

    latch61 : process(gclk61)
    begin
    if gclk61'event and gclk61 ='1' then
        invalid_bit_vector(61)<=PM_down_invalid;
    end if;
    end process;

    latch62 : process(gclk62)
    begin
    if gclk62'event and gclk62 ='1' then
        invalid_bit_vector(62)<=PM_down_invalid;
    end if;
    end process;

    latch63 : process(gclk63)
    begin
    if gclk63'event and gclk63 ='1' then
        invalid_bit_vector(63)<=PM_up_invalid;
    end if;
    end process;

end behavioral;
```

A.3.7 Storage 2

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- one of the 64-bit storages
entity invalid_flag_2 is
port
(
    clk                 : in  std_logic;
    clock_enable        : in  std_logic_vector(0 to 31);
    PM_up_invalid       : in std_logic;
    PM_down_invalid     : in std_logic;
    invalid_bit_vector  : out std_logic_vector(0 to 63)
);
end invalid_flag_2;

architecture behavioral of invalid_flag_2 is
    signal gated_clock  : std_logic_vector(0 to 63);
    signal gclk00       : std_logic;
    signal gclk01       : std_logic;
    signal gclk02       : std_logic;
    signal gclk03       : std_logic;
    signal gclk04       : std_logic;
    signal gclk05       : std_logic;
    signal gclk06       : std_logic;
    signal gclk07       : std_logic;
    signal gclk08       : std_logic;
    signal gclk09       : std_logic;

    signal gclk10       : std_logic;
    signal gclk11       : std_logic;
    signal gclk12       : std_logic;
    signal gclk13       : std_logic;
    signal gclk14       : std_logic;
    signal gclk15       : std_logic;
    signal gclk16       : std_logic;
    signal gclk17       : std_logic;
    signal gclk18       : std_logic;
    signal gclk19       : std_logic;

    signal gclk20       : std_logic;
    signal gclk21       : std_logic;
    signal gclk22       : std_logic;
    signal gclk23       : std_logic;
    signal gclk24       : std_logic;
    signal gclk25       : std_logic;
    signal gclk26       : std_logic;
    signal gclk27       : std_logic;
    signal gclk28       : std_logic;
    signal gclk29       : std_logic;

    signal gclk30       : std_logic;
    signal gclk31       : std_logic;
    signal gclk32       : std_logic;
    signal gclk33       : std_logic;
    signal gclk34       : std_logic;
    signal gclk35       : std_logic;
    signal gclk36       : std_logic;
    signal gclk37       : std_logic;
    signal gclk38       : std_logic;
    signal gclk39       : std_logic;

    signal gclk40       : std_logic;
```

```
    signal gclk41       : std_logic;
    signal gclk42       : std_logic;
    signal gclk43       : std_logic;
    signal gclk44       : std_logic;
    signal gclk45       : std_logic;
    signal gclk46       : std_logic;
    signal gclk47       : std_logic;
    signal gclk48       : std_logic;
    signal gclk49       : std_logic;

    signal gclk50       : std_logic;
    signal gclk51       : std_logic;
    signal gclk52       : std_logic;
    signal gclk53       : std_logic;
    signal gclk54       : std_logic;
    signal gclk55       : std_logic;
    signal gclk56       : std_logic;
    signal gclk57       : std_logic;
    signal gclk58       : std_logic;
    signal gclk59       : std_logic;

    signal gclk60       : std_logic;
    signal gclk61       : std_logic;
    signal gclk62       : std_logic;
    signal gclk63       : std_logic;

begin

clock_gate: process (clock_enable, clk)
variable i :integer range 0 to 31;
variable i_v :std_logic_vector(4 downto 0);
variable i_tail :std_logic;
variable double_i_G :integer range 0 to 63;
variable double_i_G_inc :integer range 0 to 63;
begin

-- setup connection of the clock gate signal
-- ONLY WIRES. NO EFFICIENCY ISSUE


for i in 0 to 31 loop
    i_v:=conv_std_logic_vector(i,5);
    i_tail:=i_v(0) xor i_v(1) xor i_v(2) xor i_v(3) xor i_v(4);
    double_i_G:=conv_integer(unsigned(i_v & i_tail));
    double_i_G_inc:=conv_integer(unsigned(i_v & not i_tail));
    gated_clock(double_i_G)<=clk and clock_enable(i);
    gated_clock(double_i_G_inc)<=clk and clock_enable(i);
end loop;

--stored_invalid_bit<=gated_clock;

end process;

clk_conv: process (gated_clock)
begin

--Dummy connections for DPFLI
--not sure if work

    gclk00<=gated_clock(00);
    gclk01<=gated_clock(01);
    gclk02<=gated_clock(02);
    gclk03<=gated_clock(03);
```

```
gclk04<=gated_clock(04);
gclk05<=gated_clock(05);
gclk06<=gated_clock(06);
gclk07<=gated_clock(07);
gclk08<=gated_clock(08);
gclk09<=gated_clock(09);

gclk10<=gated_clock(10);
gclk11<=gated_clock(11);
gclk12<=gated_clock(12);
gclk13<=gated_clock(13);
gclk14<=gated_clock(14);
gclk15<=gated_clock(15);
gclk16<=gated_clock(16);
gclk17<=gated_clock(17);
gclk18<=gated_clock(18);
gclk19<=gated_clock(19);

gclk20<=gated_clock(20);
gclk21<=gated_clock(21);
gclk22<=gated_clock(22);
gclk23<=gated_clock(23);
gclk24<=gated_clock(24);
gclk25<=gated_clock(25);
gclk26<=gated_clock(26);
gclk27<=gated_clock(27);
gclk28<=gated_clock(28);
gclk29<=gated_clock(29);

gclk30<=gated_clock(30);
gclk31<=gated_clock(31);
gclk32<=gated_clock(32);
gclk33<=gated_clock(33);
gclk34<=gated_clock(34);
gclk35<=gated_clock(35);
gclk36<=gated_clock(36);
gclk37<=gated_clock(37);
gclk38<=gated_clock(38);
gclk39<=gated_clock(39);

gclk40<=gated_clock(40);
gclk41<=gated_clock(41);
gclk42<=gated_clock(42);
gclk43<=gated_clock(43);
gclk44<=gated_clock(44);
gclk45<=gated_clock(45);
gclk46<=gated_clock(46);
gclk47<=gated_clock(47);
gclk48<=gated_clock(48);
gclk49<=gated_clock(49);

gclk50<=gated_clock(50);
gclk51<=gated_clock(51);
gclk52<=gated_clock(52);
gclk53<=gated_clock(53);
gclk54<=gated_clock(54);
gclk55<=gated_clock(55);
gclk56<=gated_clock(56);
gclk57<=gated_clock(57);
gclk58<=gated_clock(58);
gclk59<=gated_clock(59);

gclk60<=gated_clock(60);
```

```
    gclk61<=gated_clock(61);
    gclk62<=gated_clock(62);
    gclk63<=gated_clock(63);

end process;

--ALL the latches and clock gates

    latch00 : process(gclk00)
    begin
    if gclk00'event and gclk00 ='1' then
        invalid_bit_vector(0)<=PM_up_invalid;
    end if;
    end process;

    latch01 : process(gclk01)
    begin
    if gclk01'event and gclk01 ='1' then
        invalid_bit_vector(1)<=PM_down_invalid;
    end if;
    end process;

    latch02 : process(gclk02)
    begin
    if gclk02'event and gclk02 ='1' then
        invalid_bit_vector(2)<=PM_down_invalid;
    end if;
    end process;

    latch03 : process(gclk03)
    begin
    if gclk03'event and gclk03 ='1' then
        invalid_bit_vector(3)<=PM_up_invalid;
    end if;
    end process;

    latch04 : process(gclk04)
    begin
    if gclk04'event and gclk04 ='1' then
        invalid_bit_vector(4)<=PM_down_invalid;
    end if;
    end process;

    latch05 : process(gclk05)
    begin
    if gclk05'event and gclk05='1' then
        invalid_bit_vector(5)<=PM_up_invalid;
    end if;
    end process;

    latch06 : process(gclk06)
    begin
    if gclk06'event and gclk06 ='1' then
        invalid_bit_vector(6)<=PM_up_invalid;
    end if;
    end process;

    latch07 : process(gclk07)
    begin
    if gclk07'event and gclk07 ='1' then
        invalid_bit_vector(7)<=PM_down_invalid;
    end if;
    end process;
```

```
latch08 : process(gclk08)
begin
if gclk08'event and gclk08 ='1' then
    invalid_bit_vector(8)<=PM_down_invalid;
end if;
end process;

latch09 : process(gclk09)
begin
if gclk09'event and gclk09 ='1' then
    invalid_bit_vector(9)<=PM_up_invalid;
end if;
end process;

latch10 : process(gclk10)
begin
if gclk10'event and gclk10 ='1' then
    invalid_bit_vector(10)<=PM_up_invalid;
end if;
end process;

latch11 : process(gclk11)
begin
if gclk11'event and gclk11 ='1' then
    invalid_bit_vector(11)<=PM_down_invalid;
end if;
end process;

latch12 : process(gclk12)
begin
if gclk12'event and gclk12 ='1' then
    invalid_bit_vector(12)<=PM_up_invalid;
end if;
end process;

latch13 : process(gclk13)
begin
if gclk13'event and gclk13 ='1' then
    invalid_bit_vector(13)<=PM_down_invalid;
end if;
end process;

latch14 : process(gclk14)
begin
if gclk14'event and gclk14='1' then
    invalid_bit_vector(14)<=PM_down_invalid;
end if;
end process;

latch15 : process(gclk15)
begin
if gclk15'event and gclk15 ='1' then
    invalid_bit_vector(15)<=PM_up_invalid;
end if;
end process;

latch16 : process(gclk16)
begin
if gclk16'event and gclk16 ='1' then
    invalid_bit_vector(16)<=PM_down_invalid;
end if;
end process;
```

```
latch17 : process(gclk17)
begin
if gclk17'event and gclk17 ='1' then
    invalid_bit_vector(17)<=PM_up_invalid;
end if;
end process;

latch18 : process(gclk18)
begin
if gclk18'event and gclk18 ='1' then
    invalid_bit_vector(18)<=PM_up_invalid;
end if;
end process;

latch19 : process(gclk19)
begin
if gclk19'event and gclk19 ='1' then
    invalid_bit_vector(19)<=PM_down_invalid;
end if;
end process;

latch20 : process(gclk20)
begin
if gclk20'event and gclk20 ='1' then
    invalid_bit_vector(20)<=PM_up_invalid;
end if;
end process;

latch21 : process(gclk21)
begin
if gclk21'event and gclk21 ='1' then
    invalid_bit_vector(21)<=PM_down_invalid;
end if;
end process;

latch22 : process(gclk22)
begin
if gclk22'event and gclk22='1' then
    invalid_bit_vector(22)<=PM_down_invalid;
end if;
end process;

latch23 : process(gclk23)
begin
if gclk23'event and gclk23 ='1' then
    invalid_bit_vector(23)<=PM_up_invalid;
end if;
end process;

latch24 : process(gclk24)
begin
if gclk24'event and gclk24 ='1' then
    invalid_bit_vector(24)<=PM_up_invalid;
end if;
end process;

latch25 : process(gclk25)
begin
if gclk25'event and gclk25 ='1' then
    invalid_bit_vector(25)<=PM_down_invalid;
end if;
end process;
```

```
latch26 : process(gclk26)
begin
if gclk26'event and gclk26='1' then
    invalid_bit_vector(26)<=PM_down_invalid;
end if;
end process;

latch27 : process(gclk27)
begin
if gclk27'event and gclk27 ='1' then
    invalid_bit_vector(27)<=PM_up_invalid;
end if;
end process;

latch28 : process(gclk28)
begin
if gclk28'event and gclk28 ='1' then
    invalid_bit_vector(28)<=PM_down_invalid;
end if;
end process;

latch29 : process(gclk29)
begin
if gclk29'event and gclk29 ='1' then
    invalid_bit_vector(29)<=PM_up_invalid;
end if;
end process;

latch30 : process(gclk30)
begin
if gclk30'event and gclk30 ='1' then
    invalid_bit_vector(30)<=PM_up_invalid;
end if;
end process;

latch31 : process(gclk31)
begin
if gclk31'event and gclk31 ='1' then
    invalid_bit_vector(31)<=PM_down_invalid;
end if;
end process;


latch32 : process(gclk32)
begin
if gclk32'event and gclk32 ='1' then
    invalid_bit_vector(32)<=PM_down_invalid;
end if;
end process;

latch33 : process(gclk33)
begin
if gclk33'event and gclk33 ='1' then
    invalid_bit_vector(33)<=PM_up_invalid;
end if;
end process;

latch34 : process(gclk34)
begin
if gclk34'event and gclk34 ='1' then
    invalid_bit_vector(34)<=PM_up_invalid;
end if;
```

```
end process;

latch35 : process(gclk35)
begin
if gclk35'event and gclk35 ='1' then
    invalid_bit_vector(35)<=PM_down_invalid;
end if;
end process;

latch36 : process(gclk36)
begin
if gclk36'event and gclk36 ='1' then
    invalid_bit_vector(36)<=PM_up_invalid;
end if;
end process;

latch37 : process(gclk37)
begin
if gclk37'event and gclk37='1' then
    invalid_bit_vector(37)<=PM_down_invalid;
end if;
end process;

latch38 : process(gclk38)
begin
if gclk38'event and gclk38 ='1' then
    invalid_bit_vector(38)<=PM_down_invalid;
end if;
end process;

latch39 : process(gclk39)
begin
if gclk39'event and gclk39 ='1' then
    invalid_bit_vector(39)<=PM_up_invalid;
end if;
end process;

latch40 : process(gclk40)
begin
if gclk40'event and gclk40 ='1' then
    invalid_bit_vector(40)<=PM_up_invalid;
end if;
end process;

latch41 : process(gclk41)
begin
if gclk41'event and gclk41 ='1' then
    invalid_bit_vector(41)<=PM_down_invalid;
end if;
end process;

latch42 : process(gclk42)
begin
if gclk42'event and gclk42 ='1' then
    invalid_bit_vector(42)<=PM_down_invalid;
end if;
end process;

latch43 : process(gclk43)
begin
if gclk43'event and gclk43 ='1' then
    invalid_bit_vector(43)<=PM_up_invalid;
end if;
```

```
end process;

latch44 : process(gclk44)
begin
if gclk44'event and gclk44 ='1' then
    invalid_bit_vector(44)<=PM_down_invalid;
end if;
end process;

latch45 : process(gclk45)
begin
if gclk45'event and gclk45 ='1' then
    invalid_bit_vector(45)<=PM_up_invalid;
end if;
end process;

latch46 : process(gclk46)
begin
if gclk46'event and gclk46='1' then
    invalid_bit_vector(46)<=PM_up_invalid;
end if;
end process;

latch47 : process(gclk47)
begin
if gclk47'event and gclk47 ='1' then
    invalid_bit_vector(47)<=PM_down_invalid;
end if;
end process;

latch48 : process(gclk48)
begin
if gclk48'event and gclk48 ='1' then
    invalid_bit_vector(48)<=PM_up_invalid;
end if;
end process;

latch49 : process(gclk49)
begin
if gclk49'event and gclk49 ='1' then
    invalid_bit_vector(49)<=PM_down_invalid;
end if;
end process;

latch50 : process(gclk50)
begin
if gclk50'event and gclk50 ='1' then
    invalid_bit_vector(50)<=PM_down_invalid;
end if;
end process;

latch51 : process(gclk51)
begin
if gclk51'event and gclk51 ='1' then
    invalid_bit_vector(51)<=PM_up_invalid;
end if;
end process;

latch52 : process(gclk52)
begin
if gclk52'event and gclk52 ='1' then
    invalid_bit_vector(52)<=PM_down_invalid;
end if;
```

```
end process;

latch53 : process(gclk53)
begin
if gclk53'event and gclk53 ='1' then
    invalid_bit_vector(53)<=PM_up_invalid;
end if;
end process;

latch54 : process(gclk54)
begin
if gclk54'event and gclk54='1' then
    invalid_bit_vector(54)<=PM_up_invalid;
end if;
end process;

latch55 : process(gclk55)
begin
if gclk55'event and gclk55 ='1' then
    invalid_bit_vector(55)<=PM_down_invalid;
end if;
end process;

latch56 : process(gclk56)
begin
if gclk56'event and gclk56 ='1' then
    invalid_bit_vector(56)<=PM_down_invalid;
end if;
end process;

latch57 : process(gclk57)
begin
if gclk57'event and gclk57 ='1' then
    invalid_bit_vector(57)<=PM_up_invalid;
end if;
end process;

latch58 : process(gclk58)
begin
if gclk58'event and gclk58='1' then
    invalid_bit_vector(58)<=PM_up_invalid;
end if;
end process;

latch59 : process(gclk59)
begin
if gclk59'event and gclk59 ='1' then
    invalid_bit_vector(59)<=PM_down_invalid;
end if;
end process;

latch60 : process(gclk60)
begin
if gclk60'event and gclk60 ='1' then
    invalid_bit_vector(60)<=PM_up_invalid;
end if;
end process;

latch61 : process(gclk61)
begin
if gclk61'event and gclk61 ='1' then
    invalid_bit_vector(61)<=PM_down_invalid;
end if;
```

```
    end process;

    latch62 : process(gclk62)
    begin
    if gclk62'event and gclk62 ='1' then
        invalid_bit_vector(62)<=PM_down_invalid;
    end if;
    end process;

    latch63 : process(gclk63)
    begin
    if gclk63'event and gclk63 ='1' then
        invalid_bit_vector(63)<=PM_up_invalid;
    end if;
    end process;

end behavioral;
```

A.3.8 Read MUX

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- 64 2-to-1 mux
entity read_mux is
port
(
        selection                :in  std_logic;
        invalid_bit_vector_in_1  : in std_logic_vector(0 to 63);
        invalid_bit_vector_in_2  : in std_logic_vector(0 to 63);
        invalid_bit_vector_out   : out std_logic_vector(0 to 63)
);
end read_mux;

architecture behavioral of read_mux is
begin

main: process (selection,invalid_bit_vector_in_1,invalid_bit_vector_in_2)
begin

--multiplexing
    if selection='1' then
        invalid_bit_vector_out<=invalid_bit_vector_in_2;
    else
        invalid_bit_vector_out<=invalid_bit_vector_in_1;
    end if;

end process;

end behavioral;
```

A.3.9 Output

```
--Author Kehuai Wu
--s010596@student.dtu.dk
--version: final
--date 28/07/03


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- The OUTPUT entity in the report
-- 2*32-to-1

entity read_bits is
port
(
        invalid_bit_vector  : in std_logic_vector(0 to 63);
        step_gray           : in integer range 0 to 31;

        PM_invalid_up_out   : out std_logic;
        PM_invalid_down_out   : out std_logic
);
end read_bits;

architecture behavioral of read_bits is
begin

main: process (invalid_bit_vector, step_gray)
variable step_gray_v: std_logic_vector (4 downto 0);
variable index_up: integer range 0 to 31;
variable index_down: integer range 32 to 63;
begin



    step_gray_v:=conv_std_logic_vector(step_gray, 5);
-- J and J+32
    index_up:=step_gray;
    index_down:= conv_integer(unsigned('1' & not step_gray_v(4)
            & step_gray_v(3 downto 0)));

-- 2*32-to-1 mux
    PM_invalid_up_out<=invalid_bit_vector(index_up);
    PM_invalid_down_out<=invalid_bit_vector(index_down);


end process;

end behavioral;
```

A.3.10 Test bench and stimuli

```
--Author Kehuai Wu
```

```
--s010596@student.dtu.dk
--version: final
--date 28/07/03


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- test stimuli of the circuit

entity stimuli is
port
(
        clk                 : out  std_logic;
        step_write              : out    integer range 0 to 31;
        step_read              : out    integer range 0 to 31;
        selection               :out  std_logic;

        PM_up_invalid       : out std_logic;
        PM_down_invalid     : out std_logic;

        PM_invalid_up_out   : in std_logic;
        PM_invalid_down_out   : in std_logic



);
end stimuli;

architecture behavioral of stimuli is

signal sent_data : std_logic_vector(0 to 320);
signal received_data : std_logic_vector(0 to 320);
signal match_flag: std_logic;

begin

--create the clock signal
clock_driver: process
constant Tclk : time := 10 ns;
begin
    clk<='0';
    wait for Tclk;
    loop
        clk<='1', '0' after Tclk/2;
        wait for Tclk;
        end loop;
end process;


--flip the selection signal every 32 clock cycles

selector: process
variable temp:std_logic:='0';
begin
    wait for 11 ns;
    loop
        selection<=not temp;
        wait for 320 ns;
        temp:=not temp;
    end loop;
end process;
```

```
--data output process
write_flag: process
variable test_data : std_logic_vector(0 to 320):= "10111000101011000001
0010101010101001101010010101010100101010101011111001011111000000000000101
11000000111000101001011010100000011111110000001000011110010101111100010
000100000101100101010010001110010010110001010010111000010001110001010010
01010101010100101010110001010010101010101101010101010101010101010000110
10011100000110110";
variable i   :  integer range 0 to 40000:=0;
begin
sent_data<=test_data;
    wait for 11 ns;
    while i < 40000 loop
--address of write
        step_write<=i mod 32;
-- send data output
        PM_up_invalid<=test_data((i mod 160)*2);
        PM_down_invalid<=test_data(((i mod 160)*2)+1);
        wait for 10 ns;
        i:=i+1;
    end loop;
end process;

--read and compare
read_flag: process
variable i              :  integer range 0 to 40000:=0;
variable retrieved_data :   std_logic_vector(0 to 320);
variable j              :   integer range 0 to 40000:=0;
begin
retrieved_data(320):='0';
    wait for 331 ns;
    loop
-- address for read
        step_read<=i mod 32;
        wait for 10 ns;

--store data in buffer
        if j<5 then
        retrieved_data((i mod 32)+j*64):=PM_invalid_up_out;
        retrieved_data((i mod 32)+j*64+32):=PM_invalid_down_out;
        end if;

        if (i mod 32) = 31 then
            j := j+1;
        end if;
        i:=i+1;
    received_data<=retrieved_data;
    end loop;
end process;

compare:process(received_data,sent_data)
begin
--compare the read and write data
if received_data=sent_data then
    match_flag<='1';
else
    match_flag<='0';
end if;
end process;

end behavioral;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

-- test circuit structure
entity test_bench is
end test_bench;

architecture structural of test_bench is


component invalid_bit_backup
port
(
        clk                  : in  std_logic;
        step_write               : in    integer range 0 to 31;
        step_read              : in    integer range 0 to 31;
        selection               :in  std_logic;

        PM_up_invalid       : in std_logic;
        PM_down_invalid     : in std_logic;

        PM_invalid_up_out   : out std_logic;
        PM_invalid_down_out   : out std_logic



);
end component;


component stimuli
port
(
        clk                  : out  std_logic;
        step_write               : out    integer range 0 to 31;
        step_read              : out    integer range 0 to 31;
        selection               :out  std_logic;

        PM_up_invalid       : out std_logic;
        PM_down_invalid     : out std_logic;

        PM_invalid_up_out   : in std_logic;
        PM_invalid_down_out   : in std_logic



);
end component;


signal        clk                  : std_logic;
signal        step_write           : integer range 0 to 31;
signal        step_read            : integer range 0 to 31;
signal        selection            : std_logic;

signal        PM_up_invalid        : std_logic;
signal        PM_down_invalid      : std_logic;

signal        PM_invalid_up_out    : std_logic;
signal        PM_invalid_down_out : std_logic;
```

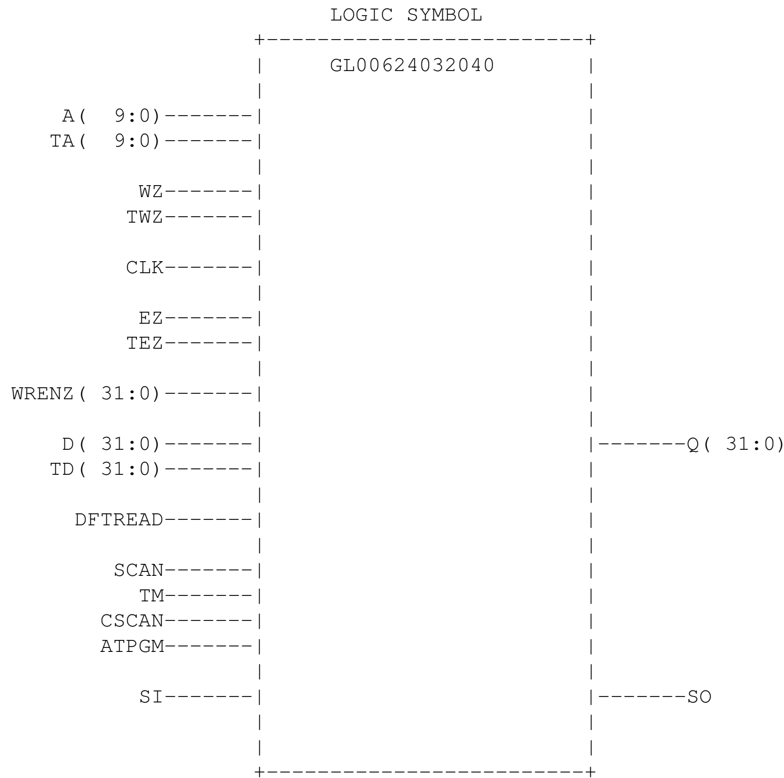
```
begin

tester: stimuli  port map(clk,step_write,step_read,selection,PM_up_invalid,
PM_down_invalid,PM_invalid_up_out,PM_invalid_down_out);
dut: invalid_bit_backup port map(clk,step_write,step_read,selection,
PM_up_invalid,PM_down_invalid,PM_invalid_up_out,PM_invalid_down_out);

end structural;
```

B Data sheet

B.1 GL00624032040 data sheet

```
             TEXAS INSTRUMENTS INCORPORATED
        APPLICATION SPECIFIC INTEGRATED CIRCUITS

               SINGLE PORT CMOS CLOCKED RAM
            FOR USE WITH THE GS50 ASIC PRODUCT

    COPYRIGHT 2001 BY TEXAS INSTRUMENTS INCORPORATED


    @(#) GL_SINGLE_PORT CRAM Datasheet Template version 1.00
      @(#) DISS Datasheet Generator version 1.0.1 Dec 13, 2002
           @(#) Created Wed Apr  2 05:32:36 2003



    +-------------------------------------------------+
    | Texas Instruments reserves the right to change |
    | or discontinue this product without notice.    |
    +-------------------------------------------------+


+-------------------------------------------------------------+
| Implementation of compacted/embedded functions will impact |
| overall development cycle time  without prior TI knowledge |
| and commitment  to support. TI reserves the  right to seek |
| additional non-recurring engineering expense  to cover the |
| costs  associated with  all-level, embedded array designs. |
+-------------------------------------------------------------+
```

```
GS50                                                     CRAM GL00624032040
SERIES                                      624-WORDS BY 32-BITs CLOCKED RAM
-------------------------------------------------------------------------
```




```
GS50                                                     CRAM GL00624032040
```

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

FUNCTION TABLE

Functional Modes (SCAN,ATPGM,CSCAN,TM,DFTREAD = L)

INPUTS				OUTPUTS	RAM
CLK EZ WZ	A(9:0)	WRENZ(31:0)	D(31:0)	Q(31:0)	MODE
/ H X	X	X	X	Q	Deselected
/ L L	Valid Address	Valid WRENZ	Data W	Data W/Q	Write
/ L H	Valid Address	X	X	=MEM[A]	Read

Test Modes

INPUTS										OUTPUTS		RAM
CLK	SCAN	ATPGM	CSCAN	TM	EZ	WZ	TEZ	TWZ	SCAN FLOPS	SO	Q(31:0)	MODE
/	L	L	L	H	X	X	L	L	Disabled	SO	TData W/Q	Test Write
/	L	L	L	H	X	X	L	H	Disabled	SO	=MEM[TA]	Test Read
/	L	L	H	L	L	L	X	X	Enabled	=SFn	Data W/Q	Capture Wt
/	L	L	H	L	L	H	X	X	Enabled	=SFn	=MEM[A]	Capture Rd
/	L	L	H	H	X	X	L	L	Enabled	=SFn	TData W/Q	Cap/Test Wt
/	L	L	H	H	X	X	L	H	Enabled	=SFn	=MEM[TA]	Cap/Test Rd
/	L	H	X	L	X	X	X	X	Enabled	=SFn	Data->Q	ATPG
/	L	H	X	H	X	X	X	X	Enabled	=SFn	TData->Q	ATPG/Test
/	H	X	X	X	X	X	X	X	Enabled	=SFn	=ScanChain	Scan

H = high logic level
L = low logic level
X = irrelevant including transitions
/ = rising edge of CLK
Q = the operation effects no change on the pin
SO = the operation effects no change on the pin
=MEM[A] = Contents of memory location [A]
=MEM[TA] = Contents of memory location [TA] (Test Address Input)
=Scanchain = Contents of scan flop(s) in the scan chain are propagated to the corresponding Q output(s).
=SFn = Contents of the last scan flop in the scan chain are propagated to the SO scan ouput pin.
Data W = Logic levels present on the D data inputs.
Data W/Q = Logic levels present on the D data input(s) of the bit(s) enabled by the WRENZ inputs are propagated to the corresponding Q output(s). The Q outputs of bits which are not enabled continue to output data from the previous read or write cycle.
Data->Q = Logic levels present on the D data input(s) are propagated to the corresponding Q output(s). All WRENZ inputs are forced active so all Q(s) effected.

```
  TData->Q = Logic levels present on the TD test data input(s) are
             propagated to the corresponding Q output(s).  All WRENZ
             inputs are forced active so all Q(s) effected.

  Valid Data/Address/WRENZ = Any combination of H or L logic levels
          which meet constraint (tsu, th) timing requirements.

NOTE: At power up the  logical value stored in the address input, data
      input, write enable input, EZ, WZ,  and output latches  is unknown.
      The RAM could be in any functional  mode.  Clocks  and  control
      inputs must be applied to put the module into a known state.

GS50                                                CRAM GL00624032040
```

```
 SERIES                                  624-WORDS BY 32-BITs CLOCKED RAM
 -----------------------------------------------------------------------

DFTMODE

There is one DFT mode available for use.  This mode is controlled by the
DFTREAD pin.  The function of this pin is described in the table below.


DFTREAD pin controls whether the RAM  is in 'use' or 'screen' mode.
-----------------------------------------------------------------------------
'Screen' mode - used during memory test (MBIST, CPUBIST, etc) only.
                This mode makes the internal performance requirements more
                stringent than normal mode to enable improved screening
                capability of weak or defective circuits.  Memory test
                patterns should be run in this mode to fully test the RAM.
                Timing must be closed at this corner to assure chip functionality.

'Use'  mode   - Product timing closure at all corners must be done in this mode.


Notes: 1)  'Screen' and 'Use' mode only affect delay and cycle times.  Setup and Hold
           times are not effected.  See timing tables for more details.

       2)  Timing must be closed using 'Screen' and 'Use' modes.


DFT mode truth table


  +---+----------------------------------+--------------------------+
  | D |                                  |                          |
  | F |                                  |                          |
  | T |                                  |                          |
  | R |                                  |                          |
  | E |                                  |                          |
  | A |                                  |                          |
  | D |            RAM MODE              |  Message from Models     |
  |   |                                  |                          |
  |   |                                  |                          |
  +---+----------------------------------+--------------------------+
  | L | 'use' mode                       |N O N E                   |
  |   |                                  |                          |
  +---+----------------------------------+--------------------------+
  | H | 'screen' mode                    |Warning: DFT Screen/test  |
  |   |                                  |is active                 |
  +---+----------------------------------+--------------------------+


You may tie like pins of different RAM instances together (i.e. DFTREAD
of one RAM, may be tied to DFTREAD of another instance).  Also, the
instances may be of different RAM types (e.g. MVF_SINGLE_PORT DFTREAD0
maybe be tied to a GL_SINGLE_PORT memory DFTREAD).  Access to these pins
must be made availableto the tester through direct access or scan register.

 GS50                                                CRAM GL00624032040
```

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

macro characteristics

PARAMETER	TYP	UNIT
X-dimension	240.39	um
Y-dimension	170.18	um
Area	0.0409	mm^2
Density	488215	bits/mm^2
Equivalent logic displaced	13501	gates

electrical characteristics

PARAMETER		TYP	UNIT
	Any A	1.23	SL
	Any TA	1.26	SL
	CLK	1.18	SL
	Any D	1.20	SL
	Any TD	1.23	SL
FI Input loading factor	EZ	1.24	SL
	TEZ	1.14	SL
	WZ	1.21	SL
	TWZ	1.19	SL
	Any WRENZ	1.20	SL
	Any DFTREAD	1.13	SL
	SCAN	1.20	SL
	CSCAN	1.18	SL
	ATPGM	1.22	SL
	TM	1.21	SL
	SI	1.22	SL
Cpd Equivalent power dissipation capacitance,Vdd=1.30,Tj=25C			
Read mode		8.839	pF
Write mode		10.31	pF
Deselected mode		1.752	pF

GS50 CRAM GL00624032040

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

timing requirements at specified conditions

		PARAMETER	FIGURE	NORMAL	CSCAN	ATPG	UNIT
tc	Cycle time		1,2	1.62	1.62	1.62	ns
tw	Pulse duration	CLK low	1	0.74	0.74	0.74	ns
		CLK high	1,2	0.88	0.88	0.88	ns
th	Hold time	A after CLK /	1	-0.16	0.02	0.02	ns
		TA after CLK /	1	-0.16	0.02	0.02	ns
		D after CLK /	1	-0.21	-0.09	-0.09	ns
		TD after CLK /	1	-0.21	-0.09	-0.09	ns
		EZ after CLK /	1	-0.10	0.36	0.36	ns
		TEZ after CLK /	1	-0.10	0.36	0.36	ns
		WRENZ after CLK /	1	-0.24	-0.12	-0.12	ns
		WZ after CLK /	1	-0.15	0.14	0.14	ns
		TWZ after CLK /	1	-0.15	0.14	0.14	ns
		SCAN after CLK /	4	0.28	NA	NA	ns
		CSCAN after CLK /	1	0.28	NA	NA	ns
		ATPGM after CLK /	5	0.28	NA	NA	ns
		TM after CLK /	1	0.28	0.28	0.28	ns

GS50 CRAM GL00624032040

SERIES 624-WORDS BY 32-BITs MULTI-STROBE RAM

---

timing requirements at specified conditions

	PARAMETER	FIGURE	NORMAL	CSCAN	ATPG	UNIT
tsu setup time	A before CLK /	1	0.67	0.67	0.16	ns
	TA before CLK /	1	0.67	0.67	0.16	ns
	D before CLK /	1	0.47	0.47	0.43	ns
	TD before CLK /	1	0.47	0.47	0.43	ns
	EZ before CLK /	1	0.57	0.57	-0.01	ns
	TEZ before CLK /	1	0.57	0.57	-0.01	ns
	WRENZ before CLK /	1	0.48	0.47	0.46	ns
	WZ before CLK /	1	0.53	0.53	0.24	ns
	TWZ before CLK /	1	0.53	0.53	0.24	ns
	SCAN before CLK /	4	0.06	NA	NA	ns
	CSCAN before CLK /	1	0.06	NA	NA	ns
	ATPGM before CLK /	5	0.06	NA	NA	ns
	TM before CLK /	1	0.90	0.90	0.87	ns

/ = rising edge of CLK
Constraints are always minimums.
All input slews = 0.2ns

maximum switching characteristics,Vcc = 1.30 V,Tj = 25 degrees C.

PARAM-ETER		DELAY (ns)
tr	Output rise time for any Q	-1.00
tf	Output fall time for any Q	-1.00
trs	Output rise time for SO	-1.00
tfs	Output fall time for SO	-1.00

GS50 CRAM GL00624032040

SERIES 624-WORDS BY 32-BITs MULTI-STROBE RAM

Normal/CSCAN Mode with DFT

PARAMETER	FROM (INPUT)	TO (OUTPUT)	DELAY (ns)	
			Use	Screen
tarh	CLK	Any Q	-1.00	-1.00
tarl	CLK	Any Q	-1.00	-1.00
tawh	CLK	Any Q	-1.00	NA
tawl	CLK	Any Q	-1.00	NA

SCAN MODE

PARAMETER		FIGURE	NORMAL	UNIT
th hold time	SI after CLK /	4	-0.13	ns
tsu setup time	SI before CLK /	4	0.46	ns

PARAMETER	FROM (INPUT)	TO (OUTPUT)	DELAY (ns)
tasoh	CLK	SO	-1.00
tasol	CLK	SO	-1.00
tash	CLK	Any Q	-1.00
tasl	CLK	Any Q	-1.00

ATPG MODE

PARAMETER	FROM (INPUT)	TO (OUTPUT)	DELAY (ns)
tagh	CLK	Any Q	-1.00
tagl	CLK	Any Q	-1.00

All delay numbers at input slew = 0.2nS, Load = 0.3 SL

tr = output rising slew
tf = output falling slew

```
 tas  = SCAN mode access time (CLK   -> Q)
 tag  = ATPG mode access time (CLK   -> Q)
 taso = SCAN mode access time (CLK   -> SO)
 tar  = read access time (WZ=1)
 taw  = write access time (WZ=0)

GS50                                                      CRAM GL00624032040
```

```
SERIES                                 624-WORDS BY 32-BITs CLOCKED RAM
------------------------------------------------------------------------

Features
========
 *Full Parallel Access with Separate Inputs and Outputs

 *Up to 128 Kbits

 *Clocked Operation

 *No DC Standby Power

 *Optimized for density at small to medium bit counts

 *Integrated BIST/SCAN collar

Description
===========
  The Single-Port, Random Access Memory (SPRAM) is an all level  comp-
  iler function  implemented in  TI's sub-micron CMOS technology.  The
  SPRAM is intended for use in designs utilizing  the  GS50     Series
  ASIC library.

  The logic levels applied at the address bus (A), memory enable (EZ),
  bit write enable (WRENZ) and read/write (WZ) inputs are  clocked into
  the RAM on the positive going edge of the clock (CLK) input.

  The write cycle is initiated by having the WZ input and  the desired
  WRENZ  input(s) low  prior  to the positive-going edge of the clock.
  During the  write  cycle,  the outputs (Q) of the bit(s)  which have
  been enabled  by the  WRENZ input(s)  will mirror the data  that  is
  placed on the respective data (D) inputs. The outputs of the  bit(s)
  which were not enabled by the WRENZ input(s) will continue to output
  data from the previous read or write cycle. With bit write inputs
  (WRENZ) low, the data applied at the corresponding data (D) inputs is
  written  into the memory on the  positive-going transition  of the CLK
  input. For data inputs with the bit write disabled, data will not be
  written into memory.

  A read function is performed by taking WZ high prior to the positive
  going edge of the clock.  The data at the  outputs will become valid
  after a delay.

  A sense screening mode, where DFTREAD high, the sensing enable signal
  is slightly faster than in the normal mode to screen out potential
  fail chip due to weak memory cells. In this mode, any other control
  pins are functioned same as the normal mode. Only DFTREAD set to high.

  The SPRAM has the  following range of sizes:  16 to 2K  words  and 2
  to 128  bits per word.  Within  these limits,  the total SPRAM  size
  may very between  32 and 128K total bits,depending on the mux  factor
  that is used.  The memory contains embedded buffers  to reduce input
  loading. The SPRAM name, BLwwwwwbbbmm, is  determined by  the  size
  and organization, where wwwww,  bbb, mm, and k represent words, bits
  per word, and mux respectively.

GS50                                               CRAM GL00624032040
```

```
SERIES                                  624-WORDS BY 32-BITs CLOCKED RAM
------------------------------------------------------------------------

Muxed Input Control
===================

  The muxed input control consists of a single test control pin and a
  mux on each address, data and control pin (excluding the DFTREAD and
  WRENZ inputs).  Existing inputs are muxed with an additional set of
  test inputs, which are driven by the BIST controller.
  In BIST mode, all data inputs are enabled for writing.
  An active high test mode input pin (TM) selects which inputs (test
  or normal) to use.  The logic level of TM is latched into the RAM on the
  rising edge of CLK. When the stored logic level of TM is high, the
  logic levels applied at all test input pins will be latched into the
  RAM on the rising edge of CLK.  When the stored logic level of TM is
  low, the logic levels applied at all functional input pins will be
  latched into the RAM on the rising edge of CLK.  If TM is not used
  (i.e. TM and no test inputs are applied) then it must be tied to a
  logic low level.  Additionally, if TM is not used, then all test inputs
  (TA, TD, TEZ and TWZ) must also be tied to a known logic level
  (high or low).

SCAN Chain
==========

  The scan chain consists of serially connected scan flops (flip-flops)
  on each address, data, and control pin input scan I/O pins (SI/SO), and
  a scan enable input (SCAN).  DFT pins are not included in the scan chain.
  An active high scan enable input pin (SCAN) enables the scan chain and
  allows scan input data to be shifted in and scan output data to be
  shifted out.  The logic level of SCAN is latched into the RAM on
  the rising edge of CLK.

  A scan cycle is initiated by having SCAN input high prior to the rising
  edge of CLK.  During the scan cycle, the logic level applied at the
  scan input data pin (SI) will be latched into the first scan flop of the
  scan chain.  Each scan flop will latch the value stored in its adjacent
  flop in the chain.  Thus, the data will be shifted one flop towards the
  scan output in the chain on every rising edge of CLK. The last scan
  flop (nearest SO) in the chain will shift out its data to the scan output
  data pin (SO). In scan mode, the values stored in the scan chain and SO
  pin will remain unchanged until the next rising edge of CLK.  Also,
  the RAM will be disabled when the stored logic level of SCAN is high.
  On the rising edge of CLK, the output data pins (Q) will reflect the
  values stored in the scan chain.  They will continue to reflect whatever
  data is captured in the flops (using CLK) until a read operation, a
  write-through operation, an ATPG operation (discussed below), or another
  scan operation occurs.  When the stored logic level of SCAN is low, the
  scan chain will be disabled.  Thus, scan data will not be shifted in or
  out on the rising edge of CLK.

  Since the RAM is disabled during scan mode, the logic levels applied
  on the address, data, and control inputs will not be latched in memory,
  only latched in the scan chain.  Thus they must be setup to the next
  rising edge of CLK for functional operation following a scan operation.

  Scan chain input bus order will not be consistant among different memory
  configurations.  It will not be ordered MSB to LSB or vice-versa.  It
  will be ordered according to the I/O pin locations for the existing
  configuration.  If scan mode is not used (i.e. SCAN is not applied), then
  it must be tied to a logic low level.  Additionally, if SCAN is not used,
  then SI must also be tied to a known logic level (high or low).
```

GS50 CRAM GL00624032040

SCAN Capture

An additional active high scan capture input pin (CSCAN) will be included to conserve power when not in scan or ATPG modes. The logic level of CSCAN is latched into the RAM on the rising edge of CLK. When the stored logic level of CSCAN is high, the scan flops will be enabled and the logic levels applied at the input pin of each scan flop will be latched on the rising edge of CLK. This allows the input values to be captured during RAM functional operation, when scan flops would normally be disabled to reduce power. Note that since the value of TM will determine whether functional input values or test input values are latched, BIST inputs can be captured as well. The output data pins (Q) are not affected by CSCAN. The scan output pin (SO) will reflect the new data latched in the last scan flop of the scan chain on the rising edge of CLK. When the stored logic level of CSCAN is low, the scan flops will be disabled. If capture mode is not used (i.e. CSCAN not applied), then it must be tied to a logic low level.

This is functionally different from scan mode since only the flops are enabled, not the scan chain. Thus, scan data will not be shifted in or out and the RAM will not be disabled during capture mode. Also, the data captured is not propagated to the output data pins, as it is in scan mode or ATPG mode. If a rising edge of CLK occurs while CSCAN is high, a normal read or write cycle will occur, but the scan flops will remain disabled and not capture any input values. All strobe to strobe rising edge setup constraints must be met in capture mode, unless the memory enable input (EZ) is low disabling its corresponding strobe input.

ATPG Mode

An active high ATPG mode input pin (ATPGM) is included to allow write through functionality independent of a memory's functional operation. The logic level of ATPGM will be latched into the RAM on the rising edge of CLK. ATPGM will be includeed on all RAMs, even ones which already have functional write-through capability.

When the stored level of ATPGM is high, the logic levels applied at the input of each scan flop will be latched on the rising edge of CLK. The captured value on each data input pin will also propagate to its corresponding output data pin (Q), thus allowing write-through functionality. Also, the scan output pin(SO) will reflect the new data latched in the last scan flop of the scan chain on the rising edge of CLK. The RAM will be disabled while ATPGM is latched high. When the stored logic level of ATPGM is low, the state of the scan flops will be disabled. If ATPG mode is not used (i.e. ATPGM not applied), then it must be tied to a logic low level.

This is functionally different from the scan mode since only the flops are enabled, not the scan chain. Thus, scan data will not be shifted in or out during ATPGM mode. However, the data captured will be propagated to the output data pins and the RAM is disabled, as it is in scan mode.

```
SERIES                                  624-WORDS BY 32-BITs CLOCKED RAM
-------------------------------------------------------------------------

Test Input (TM, SCAN, CSCAN, and ATPGM) Notes
=============================================

  SCAN will take precedence over all other test mode pins (TM, CSCSAN,
  and ATPGM).  Thus, when SCAN is high, the RAM will be in scan mode and
  the other test mode pin values will be ignored.  However,  during capture
  and ATPG modes, the BIST test pin (TM) will effect which input is latched.
  When the scan flops are disabled, they will retain their current latched
  value from a previous scan, capture, or ATPG mode.

Size Range
==========

  The SPRAM has the  following range of sizes:  16 to 2K  words  and 2
  to 128  bits per word.  Within  these limits,  the total SPRAM  size
  may very between  32 and 128K total bits,depending on the mux  factor
  that is used.  The memory contains embedded buffers  to reduce input
  loading. The SPRAM name, BLwwwwwbbbmm, is  determined by  the  size
  and organization, where wwwww,  bbb, mm, and k represent words, bits
  per word, and mux respectively.

GS50                                                CRAM GL00624032040
```

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

SIGNAL DESCRIPTIONS

NODE		
NAME(S)	TITLE	FUNCTION
A(9:0)	Address	Address inputs. A(0) is the least significant bit. The address is latched into the RAM on the rising edge of CLK.
TA(9:0)	Test Address	Test Address inputs. These inputs are muxed with address. If TM is high, the TA inputs are latched on the rising edge of STRBx. If TM is low, the A inputs are latched on the rising edge of CLK. Note: TA(0) is muxed with A(0), TA(1) is muxed with A(1), etc.
D(31:0)	Data	Data inputs. Data is latched on the rising edge of CLK.
TD(31:0)	Test Data	Test data inputs. These inputs are muxed with data. If TM is high, the TD inputs are latched on the rising edge of STRBx. If TM is low, the D inputs are latched on the rising edge of CLK. Note: TD(0) is muxed with D(0), TD(1) is muxed with D(1), etc.
EZ	Memory Enable	Active low memory enable input. The logic level of EZ is latched into the RAM on the rising edge of CLK. When the stored logic level of EZ is low, the RAM is enabled for writing or reading depending on the stored logic level of WZ. When the stored logic level of EZ is high, the RAM is disabled for reading or writing and the outputs are left in the state that existed at the end of the last valid read or write operation.
TEZ	Test Memory Enable	Test memory enable select input. This input is muxed with the EZ input. If TM is high, TEZ is latched on the rising edge of CLK. If TM is low, EZ is latched on the rising edge of CLK.
CLK	Clock	Clock input. Address inputs, data inputs, EZ, and WZ logic levels are latched into the RAM on the rising edge of CLK. If EZ and WZ are low on the rising edge of CLK, the RAM is in the write mode. If EZ is low and WZ is high on the rising edge of CLK, the RAM is in the read mode. Data from the addressed location will be placed on the outputs where it will remain until a different address is selected on a subsequent CLK low to high transition.

GS50 CRAM GL00624032040

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

NODE		
NAME(S)	TITLE	FUNCTION
Q(31:0)	Output	Data outputs are valid after the rising edge of CLK while the enabled RAM is in a read mode. Data outputs are equal to data inputs present at the rising edge of CLK while the enabled ram is in write mode.
WRENZ(31:0)	Bit Write Enable	Bit write enable inputs. The logic levels of the WRENZ inputs are latched on the rising edge of CLK. When the stored logic level of a WRENZ input is low, the data on the corresponding write data input is written into the addressed location and the Q output reflects the input data. When the stored logic level of a WRENZ input is high, writing to the corresponding bit is inhibited. The outputs of the disabled bits retain the data from the previous write or read operation. The WRENZ inputs are not active in the read mode.
WZ	Read/Write	Read or write mode select input. The logic level of WZ is latched into the RAM on the rising edge of CLK. When the stored logic level of WZ is low, data is written into the addressed location and the Q outputs are equal to input data. When the stored logic level of WZ is high, writing is in- hibited and, if the RAM is enabled, data from the addressed word is present at the Q outputs.
TWZ	Test Read/Write	Test Read or write mode select input. This input is muxed with the WZ input. If TM is high, TWZ is latched on the rising edge of CLK. If TM is low, WZ is latched on the rising edge of CLK.
SI	Scan Input Data	Scan data input. During scan mode (SCAN=H), the logic level of SI is latched into the first scan flop of the scan chain on the rising edge of CLK. If SCAN is low, SI is ignored.
SO	Scan Output Data	Scan data output. During scan mode (SCAN=H), capture mode (CSCAN=H), or ATPG mode (ATPGM=H), the logic level stored in the last scan flop of the scan chain is reflec- ted on SO after the rising edge of CLK. If SCAN, CSCAN, and ATPGM are low, SO remains unchanged.
DFTREAD	Read sense fast pin	When active, is used to fast internal self timing read control. The primary purpose for this pin is to screen out potencial fail due to weak memory cells.

```
 +------------+-----------+--------------------------------------------+

GS50                                                  CRAM GL00624032040
```

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

NODE		
NAME(S)	TITLE	FUNCTION
SCAN	Scan	Active high scan enable input. The logic level of SCAN is latched into the RAM on the rising edge of CLK. When the stored logic level of SCAN is high, the internal scan flops are active to operate in a scan chain with SI as the scan input pin and SO as the scan ouput pin. Shifted scan data is latched in the scan flops and and propagated to the data outputs. The RAM is disabled during scan mode. Functional and test muxed inputs are ignored. When the stored logic of SCAN is low, the scan chain is disabled.
TM	Test Mode	Active high BIST test mode input. The logic level of TM is latched into the RAM on the rising edge of CLK. Address, data, and control inputs are muxed with test address, data, and control inputs. When the stored logic level of TM is high, test inputs are selected in place of the functional inputs. They behave exctly the same and are latched into the RAM on the rising edge of CLK. When the stored logic level of TM is low, functional inputs are selected and test inputs are ignored.
CSCAN	Capture Scan	Active high scan capture input. The logic level of CSCAN is latched into the RAM on the rising edge of CLK When the stored logic level of CSCAN is high, the internal scan flops are enabled to latch the address, data, and cotrol inputs present on the rising edge of CLK. The scan chain is disabled. SI and SO are ignored and scan data is not shifted. The data output pins are not affected. When the stored logic level of CSCAN is low, the scan flops are disabled.(NOTE:If TM is high, the test inputs will be selected and therfore latched in the scan flops on the rising edge of CLK.)
ATPGM	ATPG Mode	Active high ATPG mode input. The logic level of ATPGM is latched into the RAM on the rising edge of CLK. When the stored logic of ATPGM is high, the internal scan flops are enabled to latch the address, data, and control inputs present on the rising edge of CLK. The latched values of data inputs are then propagated to the data output pins. The RAM is disabled during write through mode. The scan chain is disabled. SI and SO are ignored and scan data is not shifted. When the stored logic level of ATPGM is low, the scan flops are disabled.(NOTE:If TM is high,

```
|            |           | the test inputs will be selected and       |
|            |           | therefore latched in the scan flops on the |
|            |           | rising edge of CLK.)                       |
+------------+-----------+--------------------------------------------+

GS50                                                 CRAM GL00624032040
```

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

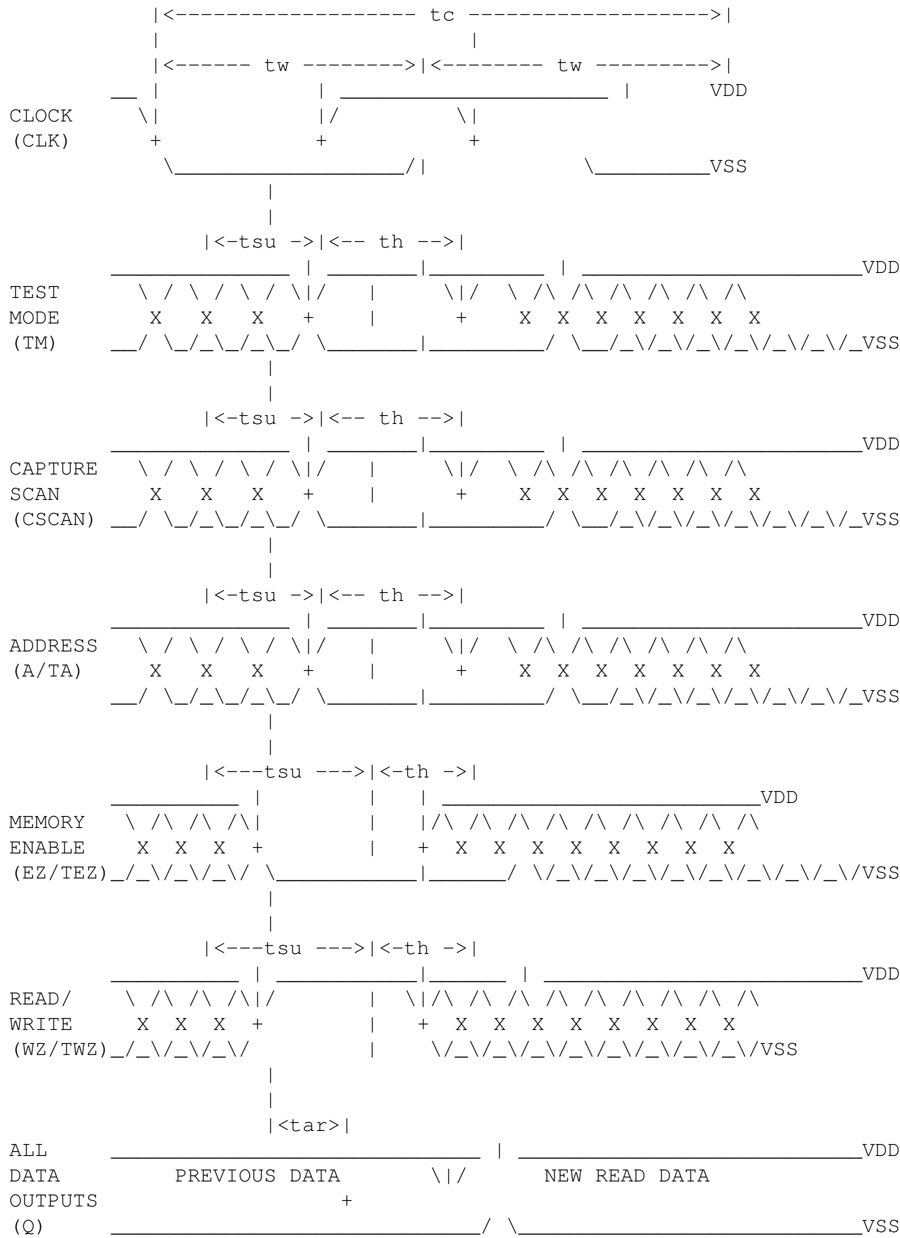


FIGURE 1. READ CYCLE

GS50 CRAM GL00624032040

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

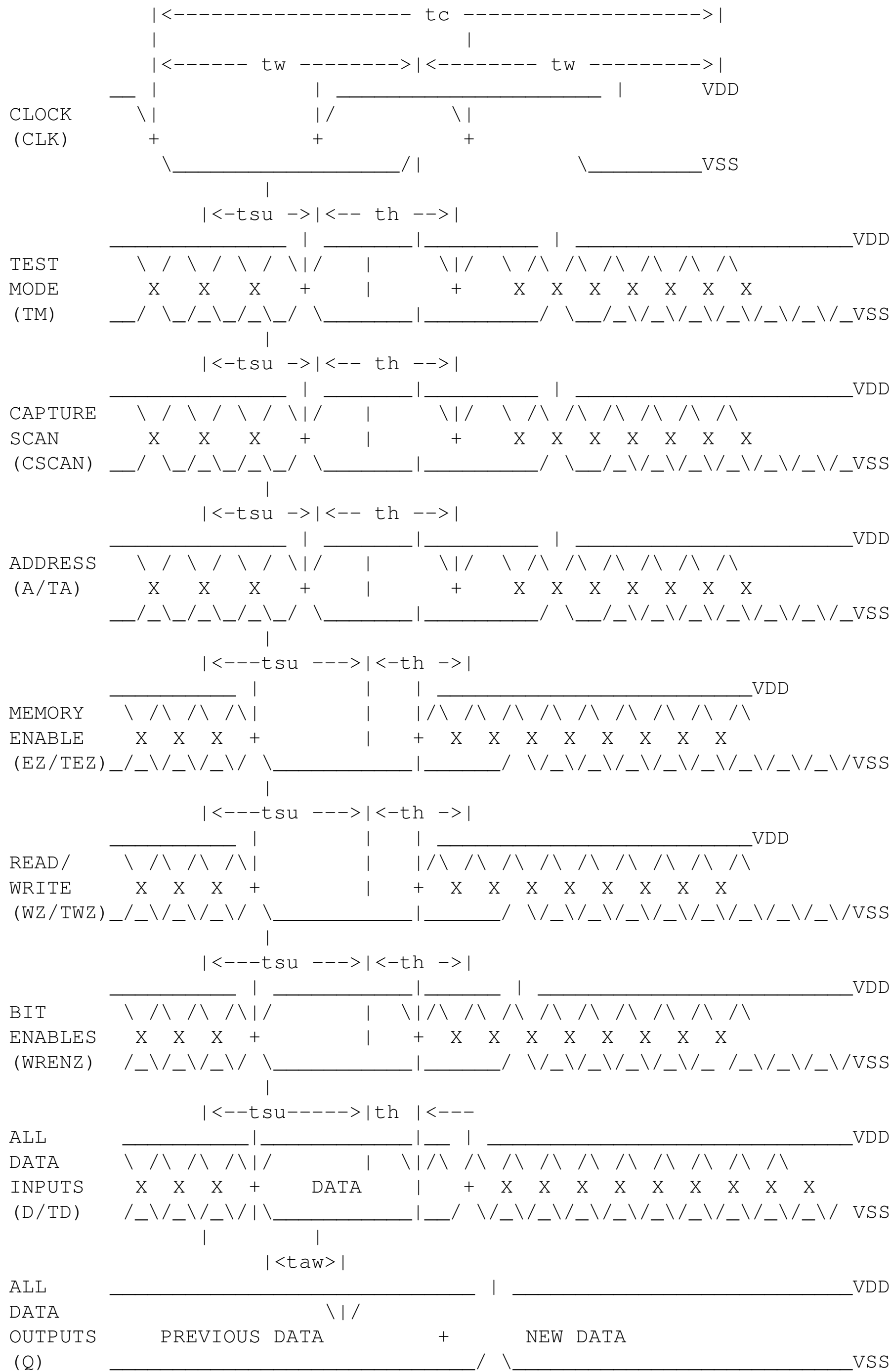


FIGURE 2. WRITE CYCLE

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

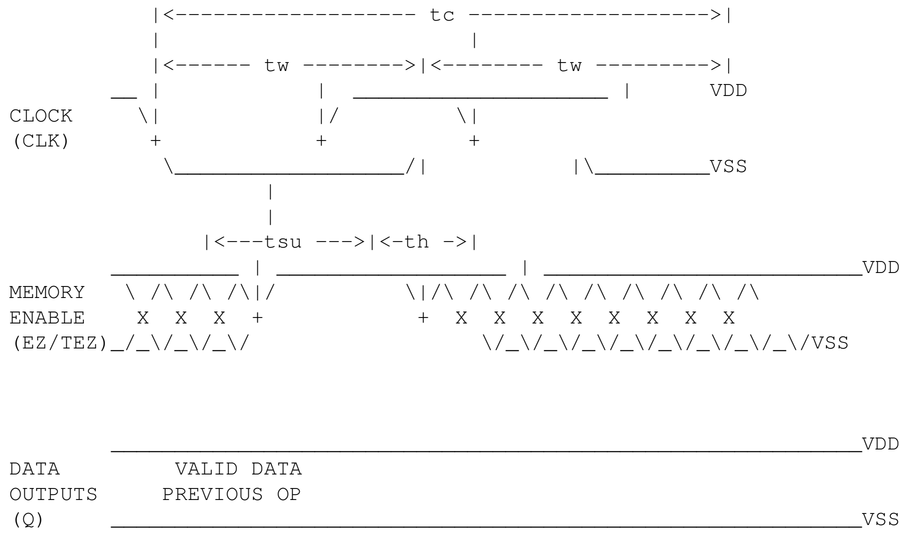


FIGURE 3. DISABLE FROM EZ/TEZ

GS50 CRAM GL00624032040

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

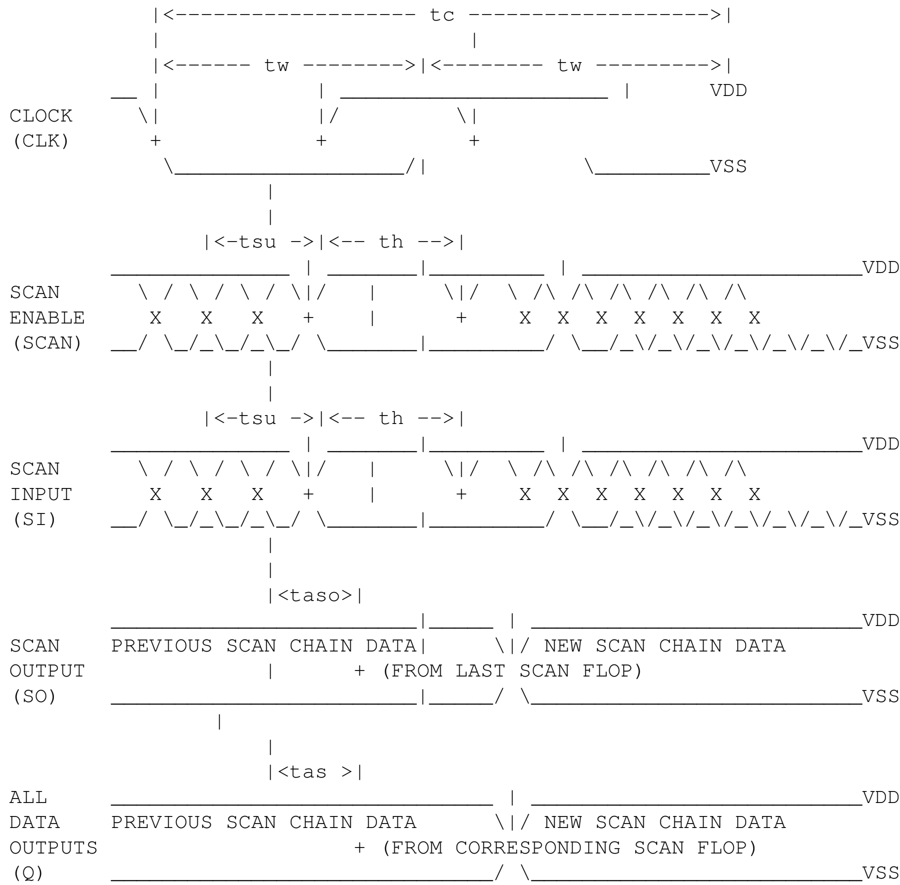


FIGURE 4. SCAN CYCLE

GS50 CRAM GL00624032040

SERIES 624-WORDS BY 32-BITs CLOCKED RAM

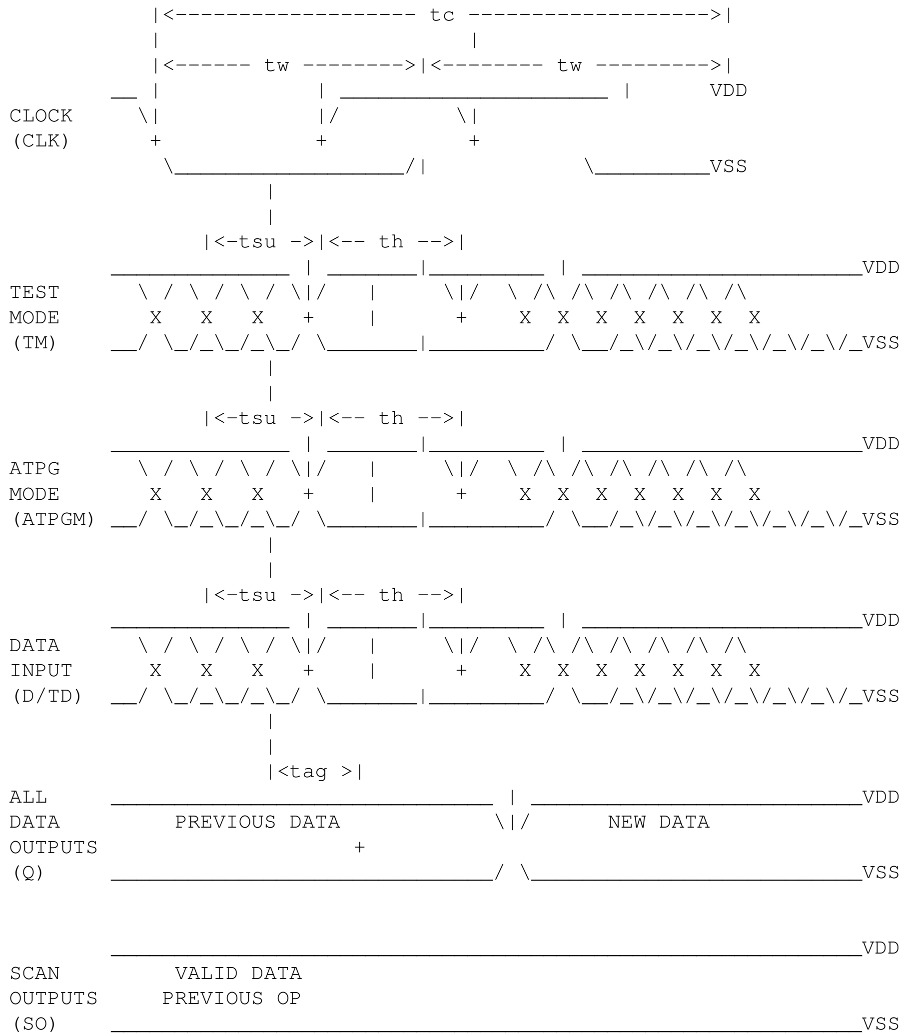
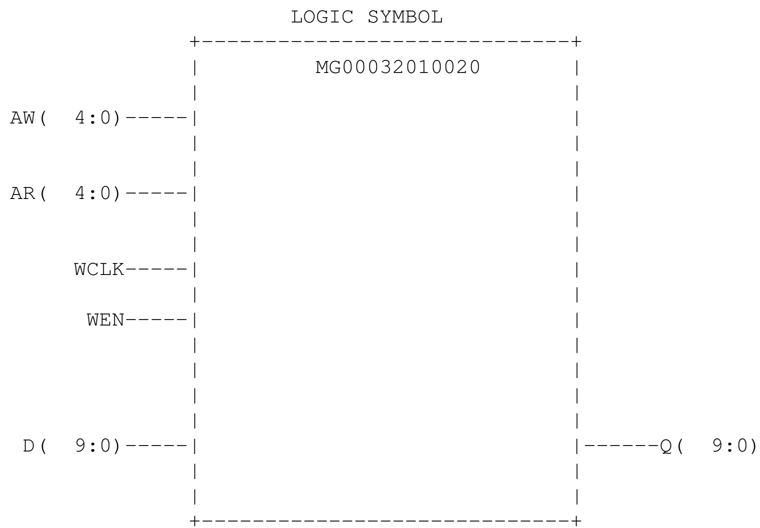


FIGURE 5. ATPG MODE

B.2 MG00032010020 data sheet

```
              TEXAS INSTRUMENTS INCORPORATED
         APPLICATION SPECIFIC INTEGRATED CIRCUITS

                   TWO PORT CMOS REGISTER FILE
           WITH ASYNCHRONOUS READ AND CLOCKED WRITE
              FOR USE WITH THE GS50 ASIC PRODUCT

     COPYRIGHT 2001 BY TEXAS INSTRUMENTS INCORPORATED


     @(#) MG_TWO_PORT SRAM Datasheet Template version 1.0
       @(#) DISS Datasheet Generator version 1.2.4 May 29, 2003
            @(#) Created Fri Jun 13 15:10:25 2003


    +----------------------------------------------------+
    |  Texas Instruments reserves the right to change   |
    |  or discontinue this product without notice.      |
    +----------------------------------------------------+

+-------------------------------------------------------------+
| Implementation of compacted/embedded functions will impact |
| overall development cycle time  without prior TI knowledge |
| and commitment  to support. TI reserves the  right to seek |
| additional non-recurring engineering expense  to cover the |
| costs  associated with  all-level, embedded array designs. |
+-------------------------------------------------------------+
```

```
GS50                                                  RegFile MG00032010020
 SERIES                          32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
 ------------------------------------------------------------------------
```



```
GS50                                             RegFile MG00032010020
SERIES                        32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
-----------------------------------------------------------------------

WRITE FUNCTION TABLE
 +-------------------------------------+----------+-----------+
 |           INPUTS                    |  OUTPUTS |           |
 +----------+-------------+-----------+----------+    RAM     |
 | WCLK WEN | AW(  4:0)   |  D(  9:0) |  Q(  9:0)|    MODE    |
 +----------+-------------+-----------+----------+-----------+
 |  /    L  |      X      |     X     |= MEM(AR) |           |
 +----------+-------------+-----------+----------+-----------+
 |  /    H  |Valid Address| Valid Data|= MEM(AR) |   Write   |
 +----------+-------------+-----------+----------+-----------+
 |  H    X  |      X      |     X     |= MEM(AR) |           |
 +----------+-------------+-----------+----------+-----------+
 |  L    X  |      X      |     X     |= MEM(AR) |           |
 +----------+-------------+-----------+----------+-----------+
 |  /    H  |   AW = AR   | Valid Data| =Data In |   Write   |
 +----------+-------------+-----------+----------+-----------+


READ FUNCTION TABLE
 +-----------------------------------+-----------+----------+
 |           INPUTS                  |  OUTPUTS  |          |
 +---------+-------------+-----------+-----------+   RAM    |
 | WCLK WEN| AR(  4:0)   |  D(  9:0) |  Q(  9:0) |   MODE   |
 +---------+-------------+-----------+-----------+----------+
 |  X    X |Valid Address|     X     | = MEM(AR) |  Read    |
 +---------+-------------+-----------+-----------+----------+

 H       = high logic level
 L       = low logic level
 X       = irrelevant including transitions
 \       = falling edge of WCLK
 /       = rising edge of WCLK
 MEM(AR) = Contents of memory location addressed by AR
 Data In = Data at the D inputs


 Note: At power up, the logical value stored in each and every memory
       location is stable but unknown.
```

GS50 RegFile MG00032010020
SERIES 32-WORD BY 10-BIT TWO-PORT REGISTER FILE

macro size

PARAMETER	TYP	UNIT
X-dimension	74.68	um
Y-dimension	67.93	um
Area	0.0051	mm^2
Density	62745	bits/mm^2
Equivalent logic displaced	1675	gates

Note : The actual X-Y dimensions extracted from the layout that appear in the LEF file will vary slightly (no more than a few microns) from those shown above due to macro interface overhead.

electrical characteristics

PARAMETER		TYP	UNIT
Fi Input Loading Factor	ANY AW	1.62	SL
	ANY AR	2.70	
	ANY D	1.39	
	WCLK	2.47	
	WEN	1.46	
Cpd_read Equivalent power dissipation capacitance for read operation @Vdd=1.30,Tj=25C (See Note 1 below)		0.4407	pF
Cpd_write Equivalent power dissipation capacitance for write operation @Vdd=1.30, Tj=25C (See Note 1 below)		0.911	pF
Cpd_standby Equivalent power dissipation capacitance consumed by write address activity when write clock (WCLK) is idle. @Vdd=1.30, Tj=25C (See Note 2 below)		0.6129	pF
Leakage		3.3E-2	mW

Note 1: Due to the functionality of the two port, CPD is measured separately for read and write operations. If the frequency of reads and writes is expected to be equal, simply take the sum of the read and write Cpd values. This is the Cpd value reported in the ACE pop-up window.

Note 2: If write address and input data busses transition only once per rising edge of write clock, no Cpd_standby allocation is necessary. That is, an average amount of address and input data activity is included in the Cpd_write value. If input bus activity is present and clock is idle, a Cpd_standby should be charged.

GS50 RegFile MG00032010020
SERIES 32-WORD BY 10-BIT TWO-PORT REGISTER FILE

timing requirements at specified conditions

PARAMETER			FIGURE	COND MIN	UNIT
tc Write Cycle Time (See Note1 below)		(tcW)	2	1.16	ns
tw Pulse Duration	WCLK low	(twlW)	2	0.79	ns
	WCLK high	(twhW)	2	0.37	ns
th Hold time	All AW after WCLK /	(thAW)	2,3	-0.12	ns
	ALL D after WCLK /	(thD)	2,3	-0.41	ns
	WEN after WCLK /	(thWEN)	2,3	-0.02	ns
tsu Setup time	All AW before WCLK /	(tsuAW)	2,3	0.44	ns
	ALL D before WCLK /	(tsuD)	2,3	0.58	ns
	WEN before WCLK /	(tsuWEN)	2,3	0.38	ns

/ = rising edge

Note1: The cycle time (tc) reported in this table is the sum of twlW and twhW, which is clock slew dependant only. However, the actual minimum allowable period may be determined by the sum of the maximum setup and hold times, which is data and clock slew dependant. The fact that twlW and twhW sum to a value less than the worst case sum of setup and hold times does not imply a longer possible twlW or twhW, but rather allows for a flexible duty cycle.

Constraints are always minimums. Constraints given are measured using 0.2ns input rise and fall times.

```
GS50                                            RegFile MG00032010020
SERIES                          32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
-----------------------------------------------------------------------
```

maximum switching characteristics

PARAMETER	FROM (INPUT)	TO (OUTPUT)	DELAY (ns)
tah	Any AR	Any Q	0.38
tal	Any AR	Any Q	0.34
tawh	WCLK	Any Q	0.96
tawl	WCLK	Any Q	0.98
tr	Output rise time for any Q		0.03
tf	Output fall time for any Q		0.03

All delays calculated at 0.2nS slew and 0.3 SL load.

```
GS50                                              RegFile MG00032010020
SERIES                         32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
------------------------------------------------------------------------

 Features
 ========
 *Independent Read and Write Address Busses

 *One Asynchronous Read-only Port and one Clocked Write-only Port

 *No DC current

 *Up to 32,768 bits of total memory

 *Three mux options (1,2,4:1)

 *Optimized for area and speed at low to moderate bit counts


Description
===========

The MG00032010020 is a dedicated 32-word by 10-bit two-port register
file. The RegFile is implemented in TI's submicron CMOS technology and
is intended for use in designs utilizing the GS50 Series ASIC library.

The read and write ports are completely independent and can access any
location simultaneously.

Writing to the RegFile is controlled by  the  write clock  input (WCLK)
and the write enable input (WEN).
On the positive (rising) edge of WCLK, the write  address bus (AW)
inputs, write data bus (D) inputs, and the value of the write enable
(WEN) input are latched.  If the latched value of WEN is high, data
will  be  written to the memory location indexed by AW.  If the
latched value of WEN is low, no write will occur.

A read function  is performed  according to logic  levels applied at
the read address bus (AR). After an access time the data in the memory
location indexed by AR will appear at the output bus (Q).  If the
memory location being accessed by the read port is written to, the
new data written into the memory location will propagate to the read
output bus (Q).  No output enable control is provided for the read
ports.  The read output ports are always active.

The RegFile has  the  following  range of  sizes:  2 to 512 words, and 1
to 256  bits per word. The RegFile contains embedded  buffers to reduce
input loading.  The generic logic symbol,  input  and output pin
descriptions,  and  function  tables  are provided on  the  preceding
pages.  The  RegFile name, MG00032010020, is  determined by the memory's
size and organization.
```

```
GS50                                          RegFile MG00032010020
SERIES                          32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
------------------------------------------------------------------------
```

SIGNAL DESCRIPTIONS

NODE		
NAME(S)	TITLE	FUNCTION
AW(4:0)	Write Address	Write address inputs. AW(0) is the least significant bit. The address is latched into the RegFile on the rising edge of WCLK.
AR(4:0)	Read Address	Read address inputs. AR(0) is the least significant bit. Read operations from AR are independent from write operations.
D(9:0)	Write Data	Write Data inputs. Data is written into the memory on the rising edge of WCLK. The data is latched into the RegFile on the rising edge of WCLK.
Q(9:0)	Read Data Output	Read Data outputs reflect the data in the memory location indexed by AR and are valid after changes in read address bus. If the values of the addressed memory location change due to a write, the read data outputs will change as well.
WCLK	Write Clock	Write clock controls write operations to the RegFile. Write address inputs, write data inputs, and the write enable input are latched on the rising edge of WCLK.
WEN	Write Enable	Write Enable is Active High. If WEN is high, the write function is enabled, such that a write will occur on the positive edge of WCLK.

```
GS50                                                        RegFile MG00032010020
SERIES                               32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
-------------------------------------------------------------------------
```

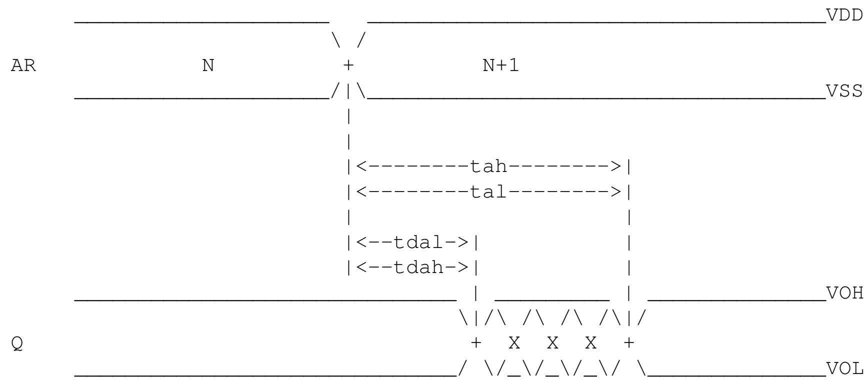


FIGURE 1. READ CYCLE

```
GS50                                                  RegFile MG00032010020
SERIES                             32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
-------------------------------------------------------------------------
```

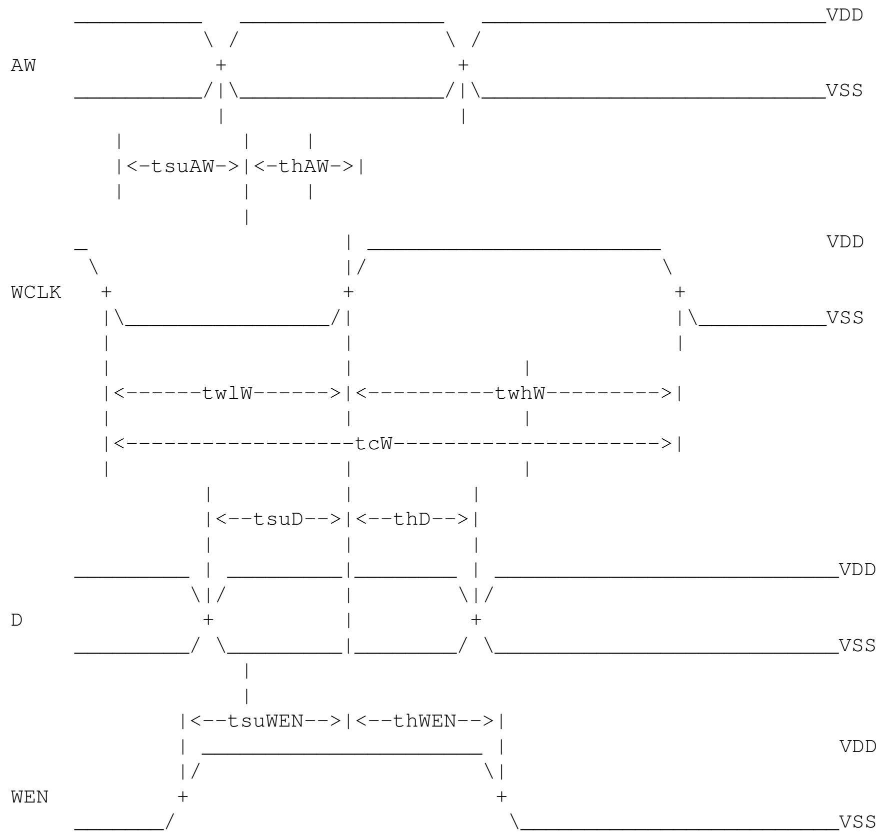


FIGURE 2. WRITE CYCLE

```
GS50                                               RegFile MG00032010020
SERIES                         32-WORD BY 10-BIT  TWO-PORT REGISTER FILE
------------------------------------------------------------------------
```

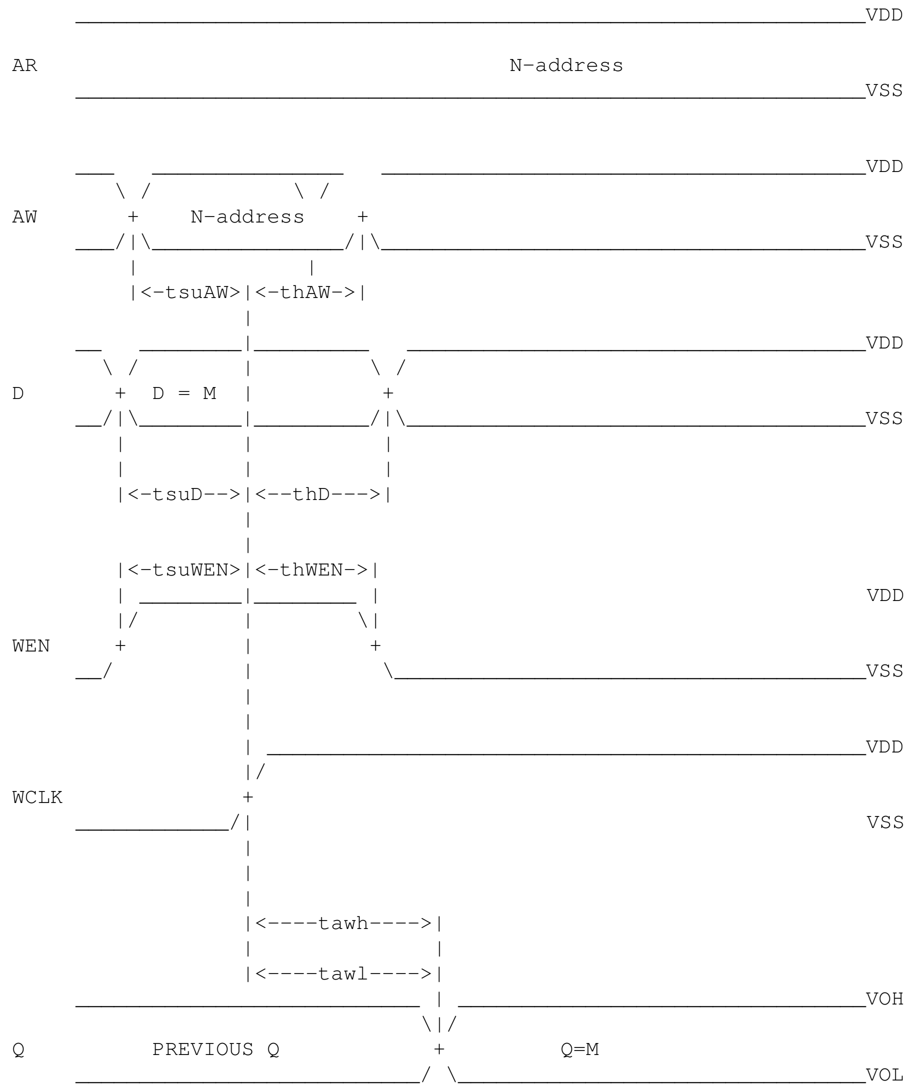


FIGURE 3. READ CYCLE USING WCLK

C Scripts

C.1 Forward SAIF file creation

```
/*Author: kehuai Wu*/
/*e-mail: s010596@student.dtu.dk*/
/*date:   31/07/03*/
/*file name: forward.do*/

/*the forward saif file generation script in dc_shell*/
/*execute this script in DC shell like "include forward.do"*/
/*before this script is executed, make sure all the entities */
/*in the decoder design are analyzed and elaborated*/


/*set up a parameter, better be set in the dc setup file */
power_preserve_rtl_hier_name=true

/* analyze the design*/
analyze -format vhdl -lib WORK {"/home/kewu/project_ver2/vhdl/src/decoder.vhd"}

/*elaborate the design*/
elaborate decoder -arch "structural" -lib DEFAULT -update

/*change name is needed when creating gate level model*/
/* but it MUST NOT BE DONE for the power analysis*/

/*change_names -rules ti_core_rules -verbose -hierarchy
change_names -rules ti_top_rules  -verbose
change_names -rules ti_net_rules -verbose -hierarchy
change_names -rules vhdl -verbose -hierarchy*/
/* link the design*/
link

/*save the forward SAIF file to the directory*/
/*where the design analyzer is started*/
rtl2saif -output fwd.saif

/*Do not forget to copy the saif file to the modelsim project directory*/
```

C.2 Activity monitoring

```
#Author: kehuai Wu
#e-mail: s010596@student.dtu.dk
#date:   31/07/03
#file name: activity.do
#The modelsim script that can collect the activity in the circuit


#start VSIM by using option (vsim -foreign "dpfli_init $SYNOPSYS/
#auxx/syn/power/dpfli/lib-hp32/dpfli.so" decoder_test_bench_config&)
#$SYNOPSYS should refer to the installation directory of synopsys
#e.g. /nokia/co_nmp/apps/synopsys.2002.05/

#Copy the forward saif file to the directory where the modelsim.ini is created.
#DO this file in the VSIM command interface .e.g "do activity.do"
```

```
#read the forward saif file created from synopsys command rtl2saif.
#dut is the entity that you want to monitor.
read_rtl_saif fwd.saif decoder_bench/dut

#set the part of the design that you want to monitor
set_toggle_region decoder_bench/dut

#run until the test bench put the data into the input buffer
run 12182

#start the activity recording
toggle_start

#run until the decoding process is finished
run 263600

#stop the activity recording
toggle_stop

view signals
view variables
view process

#let the test bench fetch the data out of the input buffer
#check the correctness of the decoding unit (signal success in the test bench)
run 10000

#save the activity report to backXXX.saif. 1e-9 is the time unit ns
toggle_report back100.saif 1e-9 decoder_bench/dut

#copy the backsaif file to the SYNOPSYS directory
```

C.3 Backward SAIF file creation

```
/*Author: kehuai Wu*/
/*e-mail: s010596@student.dtu.dk*/
/*date:   31/07/03*/
/*File name: backward.do*/

/*The script used to synthesize the design*/

/*prepare to compile the design in the SYNOPSYS dc_shell*/
/*select the clock signal*/
create_clock -name "clk" -period 10 -waveform {   "0" "5" } {   "clk" }

/*parameter must be set in order to get the SAIF working*/
find_ignore_case=true

/*read the backward saif file created from the modelsim*/
read_saif -input back100.saif -instance decoder_bench/dut -verbose

/*synthesize the design*/
compile  -map_effort medium

/*set current design*/
current_design = "/home/kewu/project_ver2/synth/dcs/decoder.db:decoder"

/*IMPORTANT!!!! design must be saved since power compiler OFTEN CRASH*/
write -format db -hierarchy -output "/home/kewu/project_ver2/synth/etc/syn_decoder.db"
{"/home/kewu/project_ver2/synth/dcs/decoder.db:decoder"}
```

```
/*Go to crash.do*/
```

C.4 Power analysis and report generation

```
/*Author: kehuai Wu*/
/*e-mail: s010596@student.dtu.dk*/
/*date:   31/07/03*/
/*filename: crashed.do*/

/*If the power compiler crashes, you should start from here.*/

/*read the backup files*/
read -format db {"/home/kewu/project_ver2/synth/etc/syn_decoder.db"}

/*read the saif file*/
read_saif -input back100.saif -instance decoder_bench/dut

/*report the power in 4-level hierarchy*/
/*don't forget to exclude the mysterious boundary net*/
report_power -hier -hier_level 4 -exclude_boundary_nets
```

Bibliography

[1] Synopsys on-line documentation sold 2002.05 and 2002.03 vol 1.

[2] 3rd Generation Partnership Project Technical Specification Group GERAN. *GSM Standard Release 3GPP TS 05, 03 v8.6.1 (2001-01) Channel Code Technical specification*. 1999.

[3] P. J. Ashenden. *The designed's guide to VHDL*. Morgan Kaufmann, 2nd edition, 2002.

[4] P. A. Bengough and S. J. Simmons. Sorting-based vlsi architectures for the m-algorithm and t-algorithm trellis decoders. *IEEE Transaction On Communications.*, 43, 1995.

[5] E. Boutillon and L. Gonzalez. Trace back techniques adapted to the surviving memory management in the m algorithm. *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2000 ICASSP*, 6(6777383):3366–3369, June 2000.

[6] H. Bustamante, I. Kang, C. Nguyen, and R. E. Peile. Stanford telecom vlsi design of a convolutional decoder. *Military Communications Conference, MILCOM '89. Conference Record*, 1(89CH2681-5):171–178, 15-18 October 1989.

[7] A. P. Chandrakasan and R. W. Brodersen. *Low Power Digital CMOS Design*. KLUWER ACADEMIC PUBLISHERS ISBN 0-7923-9576-X, 1995.

[8] M. Goel and N. R. Shanbhag. Low-power channel coding via dynamic reconfiguration. *IEEE International Conference on Acoustics, Speech, and Signal Processing, 1999. ICASSP '99*, 4(6364286):1893–1896, March 1999.

[9] R. Henning and C. Chakrabarti. Low-power approach for decoding convolutional codes with adaptive viterbi algorithm approximations. *ISLPED '02*, (7502571):68–71, August 2002.

[10] I. Kang and A. N. Willson. Low-power viterbi decoder for cdma mobile terminals. *IEEE Journal of Solid-State Circuits*, 33(5859362):473–481, March 1998.

[11] F. Ma and J. Knight. Convolutional codes. 2001.

[12] B. K. Min and N. Demassieux. A versatile architecture for vlsi implementation of the viterbi algorithm. *International Conference on Acoustics, Speech, and Signal Processing, 1991.*, 2(91CH2977-7):1101–1104, April 1991.

[13] W. Nebel and J. Mermet. *Low Power Design in Deep Submicron Electronics*. KLUWER ACADEMIC PUBLISHERS ISBN 0-7923-4569-X, 1997.

[14] J. M. Rabaey and M. Pedram. *Low Power Design Methodologies*. KLUWER ACADEMIC PUBLISHERS ISBN 0-7923-9630-8, 1996.

[15] C. M. Rader. Memory management in a viterbi decoder. *IEEE Transactions on Communications*, COM-29(9):1399–1401, September 1981.

[16] C. B. Shung, P. H. Siegel, G. Ungerboeck, and H. K. Thapar. Vlsi architectures for metric normalization in the viterbi algorithm. *IEEE Communications, 1990.*, 4(3842621):1723–1728, April 1990.

[17] S. J. Simmons. A bitonic-sorter based vlsi implementation of the m-algorithm. *IEEE Pacific Rim Conference On Communications, Computers and Signal Processing*, (89CH2691-4):337–340, June 1st-2nd 1989.

[18] S. J. Simmons. Breadth-first trellis decoding with adaptive effort. *IEEE Transaction On Communications.*, 38(1):3–12, January 1990.

[19] B. Sklar. *Digital Communications - Fundamentals and Applications*. Prentice Hall ISBN 0-13-211939-0, 2nd edition, 1988.

[20] F. Stassen and S. Pedersen. *Design of Intergrated Circuits course note 3: Practical Classes*. Informatics and Mathematical Modelling, Technical University of Denmark, January 2002.

[21] T. K. Truong, M.-T. Shih, I. S. Reed, and E. H. Satorius. A vlsi design for a trace-back viterbi decoder. *IEEE Transactions on Communications*, 40(4175703):616–624, March 1992.

[22] J. F. Wakerly. *Digital design principles and practices*. Prentice Hall, 3rd edition, 2000.

[23] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI design*. Addison Wesley, 2nd edition, 1993.

[24] S. G. Wilson. *Digital Modulation and Coding*. Prentice Hall, 1st edition, 10 1998.

[25] K. Wu. Power/area optimized channel decoding unit - requirements specification. version 1.2. February 2003.