Technical University of Denmark.

Master Thesis

Cache for reducing power consumption of a hearing instrument

Author:
Claus Lopera Andersen
s942602

Supervisor DTU:
Alberto Nannarelli

Supervisor GNR:
Kai Harrekilde-Petersen
Kashif Virk

January 31, 2012

DTU Informatics
Department of Informatics and Mathematical Modelling

Technical University of Denmark
Department of Informatics and Mathematical Modeling
Building 321
DK-2800 Kongens Lyngby
Denmark
Phone +45 45253351
Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

Power consumption have and may all ways be a issue when designing hearing aid. Today the functionality in a hearing aid is much more the a couple of filter and an amplifier. And with the introduction of wireless communication to and from a hearing aid the requirement for power keeps growing. There by the need to save power where ever it is possible get more importen.

In a system with a processor a significant part of the power is used to fetch instruction and data from memory. Fetching from a big memory cost more power then fetching from a small memory. Do this mean that the processor system is able to save power all over by adding a small cache to its memory system?

This thesis will show how a software model of a processors memory system. Along the model a file containing an address and data trace for same processor system. These two combined can be used to predict the behaviour of same system with different caches. By adding a cost function for cache hit, miss ect. it is the hope the model can calculate which cache result the lowest total power use.

First the software model is designed and build, for its result 3 caches is chosen, for implementation in VHDL. From power simulation of the VHDL the actual power use of these caches is found. The models result is the compared to this numbers.

Resume

Strøm forbruger har altid været en faktor i design af høreapparater. Kravene for at spare strøm i et høreapparat stiger. Dette er på grung af de funktioner der er i et moderne høreapparat og introduktion af trådløs kommunikation i dem.

En stor del af strøm forbrug i et processor system, bliver brugt til at hente instruktioner og jo større en memory er, desto større er strøm forbrug. Vil introduktion af en cache i sådan et system kunne mindske det totale strøm forbrug?

I denne afhandling vil en software model af et processor system blive præsenteret. Denne software model kan sammen med en fil der indholder et mønster af memory adgange.
Ønsket er at modelen skal kunne beregne, hvilken cache der resulterer i det mindst totale strøm forbrug. Dette gøres ved at have en funktion, der kan beregne strøm forbruget ved et cache hit, et cache miss, osv.

Først vil en software model blive præsenteret, ud fra de resultater denne model giver, vil 3 cache blive skrevet i VHDL. Disse 3 caches strøm forbrug vil blive fundet ved hjælp af power simulering. Resultatet fra denne power simulering vil så blive sammenlignet med resultatet fra software modellen.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in collaboration with GN Resound, to fulfil the requirements for acquiring the Master degree in engineering.

This thesis deals with hearing aid, more specific the memory system of the Digital signal processing in this hearing aid. Accessing a memory cost power and the goal of this thesis is to design a software model, which from a address and data trace from the system, can calculate the memory systems power use for different caches.

Acknowledgements

I would like to thank my supervisor at Danmarks Tekniske Universitet, Prof. Dr. Alberto Nannarelli for his help and inspiration, not just doing this thesis but though the last 2 years.

I would also like to thank Kai Harrekilde-Petersen, Team Manager for DSP Core Team at GN ReSound. Along Kai I like to thank the entire DSP Core Team for the help they have given me in the day to day work. With out then I would have been stocked.

Last but not least, I would like to thank my friends and family for their help in proofreading and commenting this thesis.

Contents

Chapter 1

Introduction

With the demand for more and more features in a hearing aid while still maintaining a long battery time, the need for power saving grows.

1.1 Project Description

Reducing the use of power in a hearing aid is becoming increasingly important and one place to to save power is in the processor and memory system.
A large part of the dynamic power is consumed by fetching data from memory, a special instructions since this is done in near to all cycles, but the same is the case for data fetching, although this is done less frequent.
In a system with out any cache, like the one in this thesis, the fetching of data (instruction or data) is done from main memory. More background information on the system in chapter 3. The question is then, can a cache help reduce the overall power used to fetch data?
A cache is normally associated with trying to achieve higher speed, by faster memory access. But in this case the access time to main memory is one cycle, witch leaves nothing to be gained in speed. But adding a cache may be able to lower the collected power used by the memory system. If the power saved in main memory access, by use of a cache is lower than the power used by the cache, then there is a total power save, see formula 1.1.

$$Power_{Total} = Power_{Mainmemory} + Power_{Cache} \tag{1.1}$$

That is assuming that main memory is one memory, i.e. it use the same power per access regardless where in the memory space the access is, this is

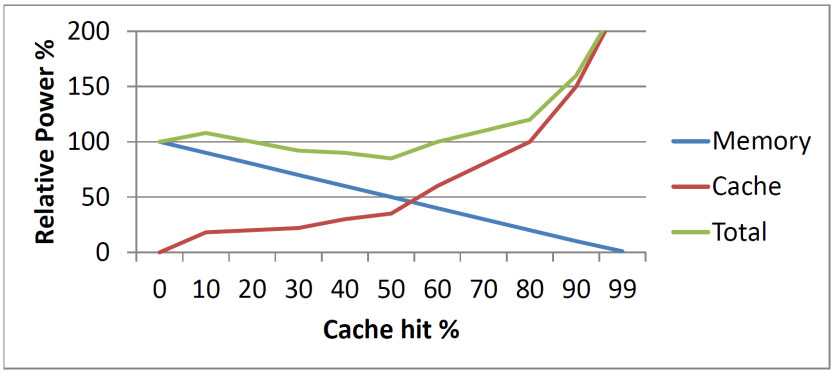


Figure 1.1: Power use in a system

not true for close to all systems, but it simplify the explication. Then the power used by main memory, is inverse to the cache hit rate. Power used by the cache is on the other hand not linear dependent with the cache hit rate, as hit rate depend on cache type, size and the access patten of the addresses.

Figure 1.1 is an illustration of what it could look like and is only based on the author's feeling. But the figure illustrates what this thesis is about, to figure out for which cache size and/or type the total power is at a minimum. In this thought up example, the solution would be the cache giving a 50 percent hit rate. This is of cause simplified.

The above defines the problem, how to find out where $Power_{Total}$ is at its minimum. This thesis presenter a possible solution to this problem. A software model of the memory system as it is in the current design, on top of this a software model of a cache will be added. As this thesis is about finding the best cache for a specific DSP, the input data use by the model should of course be from the same DSP.

A trace of addresses, showing the access patten. It will then for each of the memory access find where the requested access hits - in the main memory or in the cache. This information is then used to calculate the power used for the specific access, the total power is then given by summing up power for all access. Having a setup in the model with out a cache, makes for a comparisons.

The input data have to be as realistic as possible, as the patten of accesses have a big influence on the efficiency of the cache. Background on why will be explained in chapter 2. The power cost function is going to be based on

the author's estimate of what each function will cost when implemented in hardware. In order not just to take the author's word for granted, a VHDL implementation of some of the cache organization will be designed. These will be used for a back annotated power analysis, based on the same input as the model. The back annotated power number can then be compared to the power number from the model and the model can be aligned.

1.2 Structure of the rapport

I chapter 1.1 the project including an idea for a solution was presented. In order to make it easy for the reader a chronological order of the thesis is given below, this includes a short description of the content in each.

- **Background on cache:** An introduction to cache, why and how they work.
- **Background on current design:** The overview of the current processors design is given, with the main focus on the memory system.
- **Input data:** What is the input data and how to come by it.
- **The model:** The actual software model is presented, on a block function level and the cost function for power is described.
- **VHDL model:** Here the VHDL caches and the power analysis of the synthesised VHDL is presented.
- **Calibration of the model:** The power number from the VHDL is used to see how accurate the model is and to calibrate. After calibration the end result is presented.
- **Conclusion:** Did the model work as intender and is a cache feasible.
- **Appendix:** Here the reader will find diagram, code examples etc. The entire source code developed for this thesis is not in here, but can be found on the attached USB key, along trace files diagram etc.

With this the reader should have a good idea about the content and structure of this thesis.

1.3 GN ReSound

GN ReSound develops, manufactures and sells hearing aids. To quote ReSound from their own webpage.

Our company

ReSound provides excellent sound by offering innovative hearing aids that combine original thinking and design with solid technology – all based on deep audiological insight and understanding of users.

Our mission

To create innovative hearing solutions that constantly increase user satisfaction and acceptance – making ReSound the natural choice for hearing care professionals.

Our vision

Every day sounds are lost for millions of people with hearing challenges. ReSound will continuously develop solutions to help these people rediscover hearing, so they can live rich, active and fulfilling lives.

1.4 Tools used

TexMaker 3.1, MikTex 2.9 and Microsoft Visio

This report was made using LaTeX. MikTex was used to compile the .tex files. TexMaker 3.1 was used to edit the .tex files. The design template (pre make) is a changed version of one provided by the Department of Informatics and Mathematical Modelling (IMM) under which this thesis is written. Flowchart and diagram was make in Microsoft Visio.

Eclipse IDE for Java (Indigo) and Java SE 7 JDK

The first attempt at a model of the cache and memory system was written in Java and run using Java SE 7 JDK. All Java code was written and edited in Eclipse, which was used as debugging environment as well. Do to Java not having unsigned 32 bit integer, the model was rewritten to C++.

Visual Studio 2010

The coding and debugging of all C++ code was done in Visual Studio 2010.

Perl

The trace from the Logic analyser was rewritten to a format more suitable for the eye and for the model.

Emacs 21, NCsim and ModelSim

Emacs was used as VHDL editor under Linux while inside GN ReSounds environment, here NCsim was used as simulator. Under On Windows and for test in the start phase of designing the cache in VHDL ModelSim was used as editor and simulator.

Synplify, Xilinx and FPGA editor

For synthesis Synplify in cooperation with Xilinx, was used via GN ReSounds flow. Xilinx FPGA editor was used to debug the design after place and route.

ReSound software

A piece software used to setup the algorithm in a hearing aid. And to be used at debugging by interacting with register and the processor.

1.5 Hardware used

FPGA board

An Xilinx board with a Virtex 4, connected to it an add on an board with a chip having peripherals etc. And an other add on board with radio for wireless communication.

ReSound Unite^{TM}TV

This is a box taking in a sound signal and streaming it out wirelessly in a format understood by a ReSound hearing aid.

Logic analyser

Agilent 1681AD 102-channel logic analyzer.

Chapter 2

Background on cache

In this chapter the reader will find that some of the key parameters which need to be considered when designing a cache are presented. Before going into that, an explanation of the aspects of program code, which makes the use of cache possible is presented. Chapters 2.1 to 2.4 are written with great inspiration from [2][3][4]. At the end of this chapter other kinds of cache architecture and ways to save power are presented.

2.1 Locality

Temporal Locality

Since the memory access of a program is not random, a piece of program code it will contain loops. This is instructions which are used again and again and the same goes for the data item, that being a constant read several times or a variable read then updated then read over and over again. This is what is called Temporal Locality and the main thing which makes a cache worth implementing. The idea is that when the processor asks for an instruction or a data word, it is likely that it will ask for it again, this is due to the fact that nearly all (if not all) programs contain loop. When a piece or data or an instructions is fetched from the backing store[1] for the processor a copy is kept in the cache, so next time the processor asks for that piece it can be read from the cache and thereby saving time.

Spatial Locality

Programmers tend to cluster data, meaning that a sub part of a program tend to address data items within a narrow address space. A list or an array

[1] The memory behind the cache

is a good example of this, item i+1 is in most cases processed after item i. Instructions tent to come in sequence, for example in a loop instruction i is follow by instructions i+1 until the loop is restarted and then the sequence starts over. This is one reason for having cache block[2] when one item in the block is asked for, the rest of the block is fetched, exploiting Spatial Locality.

Algorithmic Locality

Algorithmic locality arises when a program repeatedly accesses data item or program instruction, which are spread throughout the memory space. A good example is an interrupt driven application. For example a program code where the main program runs, while every 1/10 sec an interrupt function is called which runs though a loop updating a screen. In the same code an interrupt routine can be used to empty a communication buffer. These three blocks of code are not necessarily placed close to each other, but they may be accessed frequently. This is in fact three sets of code which each explore temporal or spatial locality or both, with the issue that each can be located far from the other.

2.2 Miss

Cache miss can be divided in three groups, which is what in the literature is called The Three C's.

- **Compulsory:** This kind of miss is inevitable, as it refers to a miss caused by it being the first time that address is referenced. There will always be such misses.
- **Capacity:** Occurs because the cache can not contain the whole program. An infinite cache will not have any Capacity miss.
- **Conflict:** These are collision misses and occur in direct or set associative cache when blocks are mapped to the same place in the cache. In a full associative cache there are no Conflict misses.

[2]Having one or more word in each cache location

2.3 Cache

2.3.1 Cache groups

This section will start by defining two main groups of Caches. In table 2.1

Cache Type	Addressing Scheme	Management Scheme
Transparent cache	Transparent	Cache
Software-managed cache	Transparent	Application
Self-managed scratch-pad	Non-transparent	Cache
Scratch pad	Non-transparent	Application

Table 2.1: Main cache groups

there are two types of cache - a transparent cache and a scratch pad. A scratch pad in a small memory, having its own address space. The idea with a scratch pad is that it is small and close to the processor, which means fast access. That is has its own address space means that the application needs to know it is there, i.e. the programmer needs to think about using it during the development of the application.

A Transparent cache on the other hand is a copy of a part of the backing store. And as such the application needs not be aware that there is any cache.

In both cases above the cache and its contents can be managed by either the application or the cache itself. Both have its advantages and disadvantages.

2.3.2 Cache organization

Since a cache is a sub part of the backing store, it can not be addressed directly using the address, what is done is that the address is divided into fields as shown in table 2.2. The fields in the cache address in relations to the memory space is shown in figure 2.1.

The cache tag points to a block, and this block has the size (number of cache lines) 2^n where n is the number of bit in the index field. The field index points to the line within the cache, this can be one or more word, in table 2.2 it is 2 bit equal to four words. In figure 2.2 a block diagram of a 2-way associative cache, with a block size of 4 is shown. The list below

32 bit address:

28 bit		2 bit
Block ID		block
28-n bit	n-bit	2 bit
Cache tag	Index	block

Table 2.2: Address division in a cache

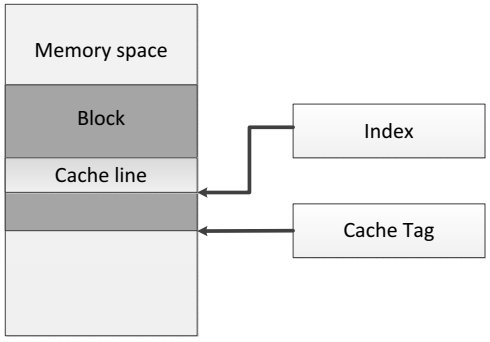


Figure 2.1: Cache fields with in the memory space

gives a description of each part and how it is connected to what was shown in figure 2.1

- **Tag:** As the cache does not contain the entire memory, a part of the address needs to be saved. As shown in figure 2.1 Tag point to a block of the size 2^index.
- **Valid:** This is a bit telling is the data in this block can be used, i.e. the data in the cache are the same as in the backing store.
- **Index:** This is one of the parameters which define the size of the cache. The index explores temporal locality.
- **Block:** This is what is refereed to as a cache block and can be one or more word. Here it is Spatial Locality which is explored.
- **Associative:** In section 2.1 the issue or Algorithmic Locality was presented, associativity solves this, by giving more small block with same index but the tag map the block to a different location.
- **Hit:** This bit shows if the address requested is in the cache and if the data is valid.

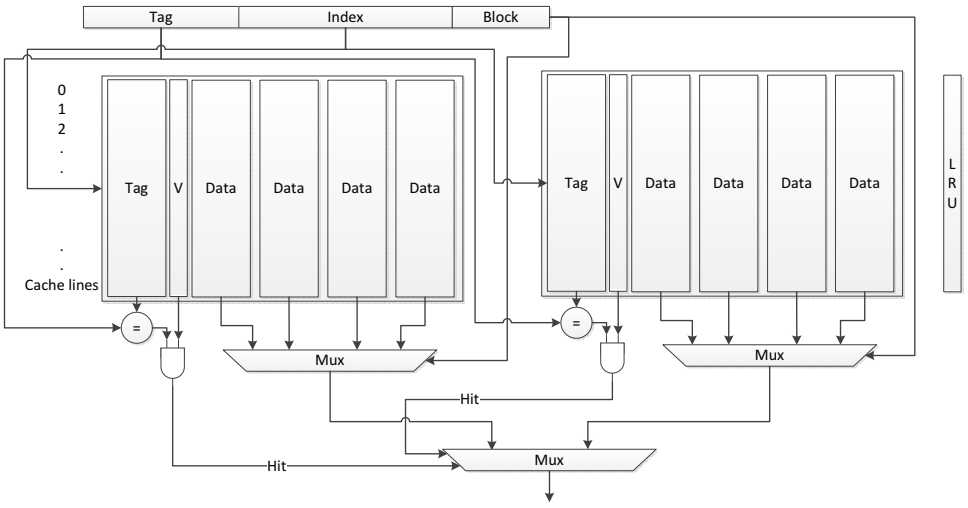


Figure 2.2: Block diagram of a 2-way associative cache, with a 4 word block

- **Set:** One set contains 0-Cache line tags and the same amount of cache blocks, there can be one or more sets in a cache. Each set can point to a different block, see figure 2.1.
- **LRU:** Least Recently Used, when having more than one set, the cache needs to have a way to choose, which set to overwrite when having to fetch new data.

Taking the cache shown in figure 2.2 and put some numbers on. The block is 2 bit, the index 4 bit and the tag 10 bit. This gives a address of 16 bit and an address space of 64k. The cache is a 2-way set associative, there are 2 block each having its own tag and data. The index on 4 bit gives 16 cache lines and the cache block size of 2 bit gives 4 word in each cache line. In total this cache will hold $cacheline * block * set = word$ in this case $16 * 2 * 4 = 256$

2.4 Design considerations

When having to design a cache there are several parameter to consider, some of them are the ones presented in section 2.3.1. Lets go through the most important.

- Where can a block be placed?
  - One place (direct mapped), this means one set.

  - A few places (set associative), in the case presented in figure 2.2 a block can be placed two places.
  - Any place (fully associative), in this case there are no index as there are only one cache line but many set in fact $\frac{Cachesize}{blocksize}$.

- How to find a block?
  - Indexing (direct mapped), here there is only one set, so index point directly to the block.
  - Indexing + Limited search (set associative), each set is checked to see if it holds the tag address requested, if one set do then index with in that set point to the requested block.
  - Full search (fully associative), here there is no index so all set is check to see if one of them holds the address requested.

- What to replaced on a miss?
  - Only one place is possible (direct mapped), in case of a direct map cache index point to one place, so there is no choice.
  - LRU (Least recently used), here the set which haven't been accessed for the longest time is replaced.
  - Random, randomly select a set and replaced the given block in that set.
  - And much more.

- What to do on a write?
  - Write-through, here the data is written to the cache and to the backing store.
  - Write-back, data is only written to the cache, then at the point where the cache block is needed for other data the data is written to the backing store.
  - And other variants of this.

All these considerations have their pro and cons, both depending on the application, the performance wanted, the power allowed, price and more.
To give an example, if a cache size of 64 block is needed why make it a direct mapped cache. A fully associative have to much better as all block can be placed anywhere. The down side is that for each block in the cache there will be an overhead of the tag plus some compare logic and it needs to

do 64 compare each time. This costs area and power, which is money and heat/battery time.
Another example is Write-back, if that is chosen, an application can write to a word and read it again without the cache having to fetch it from the backing store. But if the cache needs to save a new item at the place it has been written to. Then it needs to write it to the backing store first, which will take time and a delay is introduced (which may let to stalling the processor). In some cases this may not be a problem but in other it may be an impossible thing to ask.

2.5 Other cache architecture

In section 2.3 the standard cache with all its parameters was presented, in this section some other ways of designing a cache are introduced.

2.5.1 Filter cache

A filter cache is in fact a direct mapped cache, which is small in comparison to the other caches in the system, this is a cache architecture used to reduce power at the expense of performance. The architecture introduces a level 0 cache on the instructions cache, this cache use relative less power than the L1 cache sitting behind it, this idea was presented in [9].

2.5.2 Loop cache

In [12] an instruction cache where only loops are cache is presented. This is a small tag less direct mapped cache. It works by looking at the op-code for instructions which fall within the category short backward branch instructions. When one of those instructions are taking a small jump backward is done. At this point the cache starts caching instructions, when the same short backward branch instructions is taken again the loop cache take over and instructions are fetched from the cache. When the short backward branch instructions or any other instructions changes the instructions flow, instructions are fetch from memory again. Figure 2.3 shows a block diagram of the loop cache architecture.

2.5.3 Cache Line Buffering

As many caches use SRAM as memory, the idea of caching a line of the cache in register was found in [11]. Figure 2.4 shows a block diagram of a direct

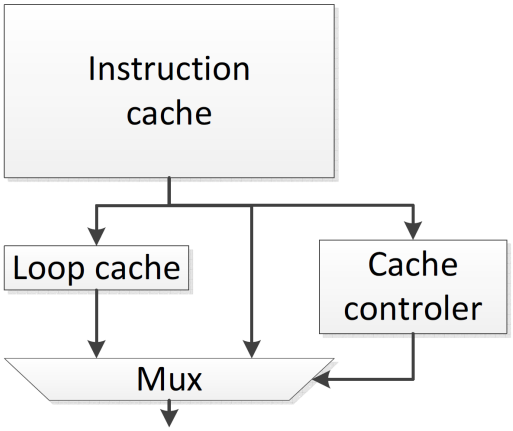


Figure 2.3: Loop cache architecture

mapped cache with two cache line buffers. The idea is that if index matches the index of one of the cache line buffers then the access to the SRAM is aborted. This has the possibility of lowering power without any performance penalty.

2.6 Other way to get low power in a memory/memory system

2.6.1 Memory partition

One well-known and well used technique to save power is to divide the memory into smaller partition called banks. And then only actives the addressed bank. The reason this reduces power is that the amount of bit cell and capacitance of the wordline reduced. The memory can be split recusived, but at some point the power overhead do to the sense amplifier, control logic etc. will become the dominating part of the power and it will no longer be desirable.

2.6.2 Latching a block of word

This can be thought of as a resented used cache within the memory and work by keeping the last used wordline in a latch. The memory needs a bit more control logic for checking if the line address matches the last used line address, the gain is performance and power in the cases where they match.

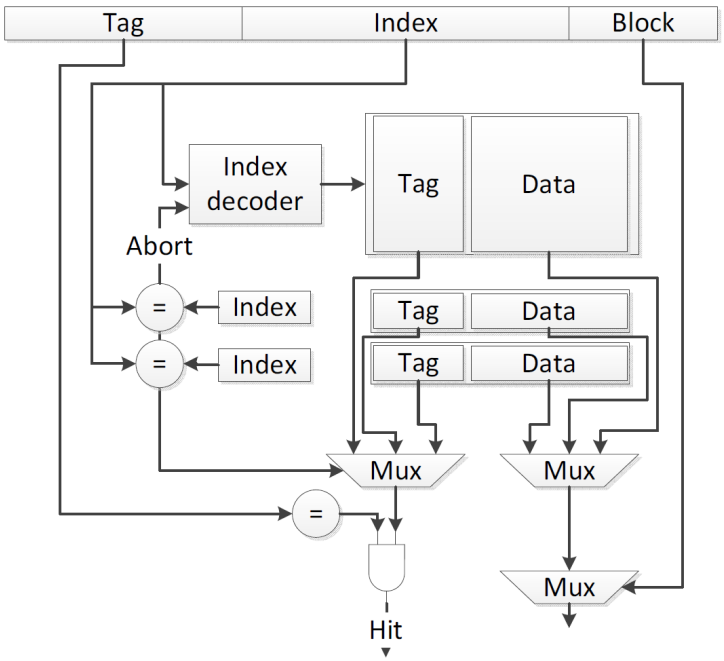


Figure 2.4: Cache with line buffer

This chapter have given the reader a background knowledges, which will make it easier to understand what is presented in this thesis. And to understand why the choice taken.

Chapter 3

Current system

In this chapter the reader will find background information regarding the DSP[1] part of GN ReSound's current hearing aids. Later in this chapter some limitations regarding IC design within GN ReSound will be mentioned, as this has influence on some of the design choices presented in this thesis.

3.1 System overview

The DSP, which is designed by GN ReSound, is a Harvard architectures. As shown in figure 3.1 this architecture has separate memory for instruction

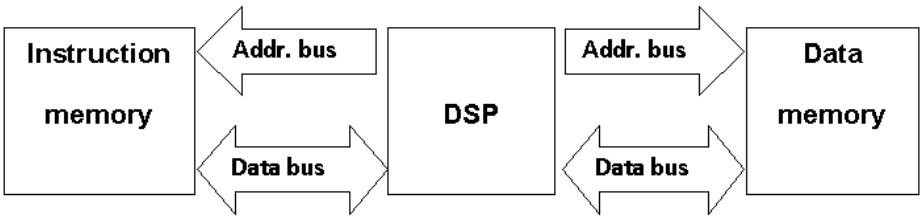


Figure 3.1: DSP Harvard architectures

and data, each with its own address and data bus. On top of that, this DSP has two data memory - a Xmem[2] and a Ymem[3], the two data memory have separated address and data bus. Even though this may sound complex, it makes the task of adding cache to the memory system simpler, because each memory can have a cache added with no regard to the other memory. The DSP with the three memory and its pipeline state is shown in figure 3.2. As

[1]Digital signal processors
[2]X memory, one of the data memory
[3]Y memory, one of the data memory

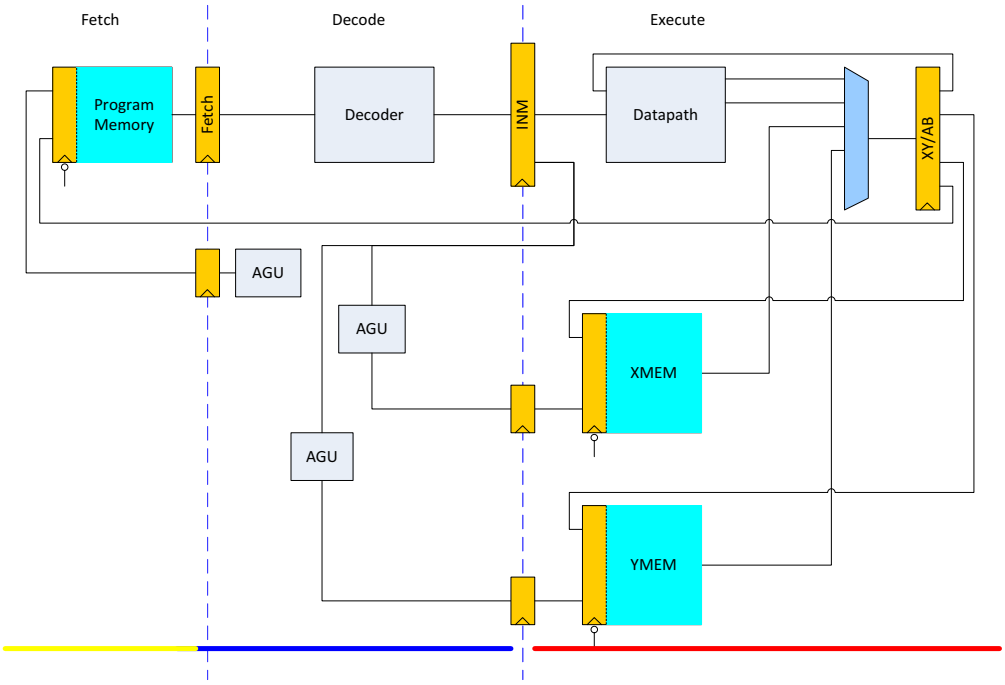


Figure 3.2: DSP block diagram

shown in figure 3.2, each memory can be considered a separate stand alone memory, in the context of memory accesses and adding cache to it. The only difference is an addition in delay since data from Xmem and Ymem have to pass through a mux.

3.2 Memory map

The memory space is divided into two halves, lower half for RAM and upper half for ROM, as shown in figure 3.3.

The handling of accessing the right memory is handled by the DSP and the cache will look at the memory space as one big memory. In the lower address space some addresses are reserved. Table 3.1 shows the address space and what the address is reserved for.

Figure 3.3: Memory space

Address start	Address end	Name
0x0015	0x0000	CORE
0x001f	0x001a	IRQ_CTRL
0x0043	0x0040	SENSE_PORTS
0x0051	0x0050	CLK_CTRL
0x0052		CHIP_REV
0x006f	0x0060	IIC_HOST
0x0073	0x0070	SYS_SURV
0x0087	0x0080	BATMON
0x00b0	0x00a0	IIS_HOST
0x00cf	0x00c0	IIC_MASTER
0x00df	0x00d0	AUDIO_OUTPUT
0x00ff	0x00e0	WL
0x0149	0x0100	SPI_MASTER
0x018f	0x0180	CLK_GEN
0x0196	0x0190	LIMITREG
0x0198		ADR
0x0199		AUX
0x01df	0x01c0	IIS_MASTER
0x0203	0x0200	COM
0x0300		TEST
0x04ff	0x0400	BIQUAD
0xf0ff	0xf000	FPGA

Table 3.1: Peripherial memory space

The model is not going to take this into account, although this will result in caching where caching may not be necessary, and there by may add to cache

miss by overwriting a value which could be used later.

3.3 Timing

As mentioned in 3.1, each memory (Pmem[4], Xmem and Ymem) can be considered a separate memory and in context of accessing them they act in the same way. Taking the Program Memory part from figure 3.2 and unfolding it, will result in figure 3.4.

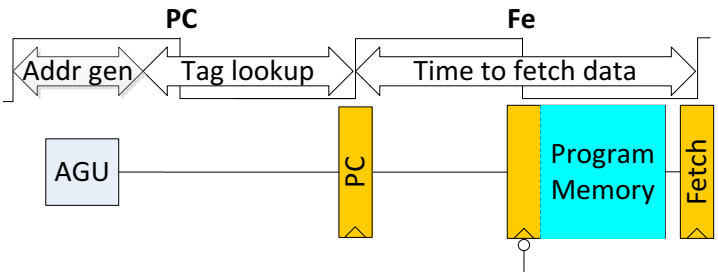


Figure 3.4: DSP memory pipeline state

As shown in figure 3.4, there are two states in the pipeline in which there is room for the cache. The author has chosen to call the first state PC[5], this state is where the address is generated. The other state is the fetch state Fe[6]. The remaining time in the PC state, after the AGU[7] have generated the address, can be used to do the tag lookup tag lookup and control logic. The result from the tag lookup are necessary in order to decide where to fetch data from.

	Miss	Hit
Read	Fetch the data from PM, save Tag and data in cache	Fetch data from cache
Write	Write data to PM	Write data to PM. Depending on writing policies write tag and data to cache, or invalided

Table 3.2: Actions to be done in RW state on memory access

[4]Program memory
[5]Program Counter
[6]Fetch
[7]Address generation unit

Table 3.2 shows the four different states a cache can be in, here cache refers to the type in 2.3. The same timing is required for Xmem and Ymem and in the Fe state they have a tighter timing requirement as there is a mux in the part, as shown in figure 3.2.

3.4 Do loop instruction

Even though this is just an overview there are one instruction which need a explanation, as it is going to be use later. This is the Do loop instruction. This instruction indicate that a loop followers, the syntax is [*do count*, *label*] where do is the 8 bit opcode, followed by count. Count can be a 8 bit value holding the numbers of time to stay in the loop, or it can be a register holding the same information. It is worth noticing that it is not allowed to have jmp as the last instruction in a do loop, but it is allowed else where in the loop.

3.5 Design limitations do to IC design rule

As the thesis is done at and for GN ReSound the design have to follow GN ReSound's rules.

- **Clocking:** All clocking has to be on the rising edge of the clock, this does not include the memory, as shown in 3.2 the memory is clocked on falling edge.

- **Timing:** Has to follow what was shown in section 3.3 and stalling the DSP due to memory read/write, is not permitted.

- **Cache size:** Due to chip size and the fact that the main memory space is only 16k words, the cache has to be small. Here small is below 32 words or in that range.

- **Memory cell:** Normal the memory in a cache is a SRAM, but since there are no commercial SRAM in that size. Even if there was, the power used in such small memories, is dominated by the sense amplifiers, see 10. This leads to a flip-flop as memory cell.

- **Tri state bus:** With flip-flops as memory a register file would be a choice, except GN ReSound does not use tri-state buses, which lead to use of mux.

3.6 Design choice of cache memory

These requirement lead to a memory array design similar to what is shown in figure 3.5. Different things can be done to save power in this design, like only changing mux depending on index. This thesis is not going to that into account. The reason being that it will make the power cost function in the

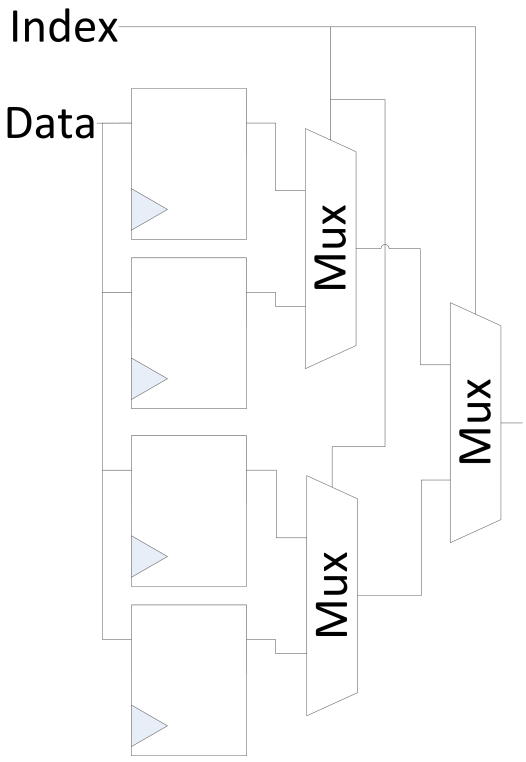


Figure 3.5: Memory array in flip-flop

model a lot more complex and that most approaches to lower power, gives better result as the number of cache lines goes up, i.e. will not result in much power saving in this design.

This should give the reader enough background knowledge to be able to follow the rest of this thesis, should the reader like to get a better insight to the design and architecture of the DSP please take a look at [13].

Chapter 4

Input data

In this chapter the reader can find information about the input data needed by the model. The reader will find information on what data is needed and how the author produced the given data.

4.1 What is meant by input data?

The model has to act as the memory system and in doing that it needs input. For a read it needs an address and some control signals and for write some data as well. Here some pattens of address access could be generated along with random generated data. The author used this approach when testing the function of the model, for example to check the function of an n-way cache, with $n > 1$.

The model has to find the best solution for the DSP running like it would in a real life situation. Then the best input to the model, will be an access patten from the real DPS, i.e. a trace from the DSP while running like a hearing aid.

4.2 Background

GN ReSound has a flow which can generate a FPGA version of the DSP, along with some other hardware. This results in a hearing instrument which can run on a FPGA board, shown in 4.2. With use of a FPGA board and a logic analyser generation of real live traces is possible.

The reason for using a FPGA board to generate a trace file is to achieve the most accurate trace. It is possible to generate an address trace using a

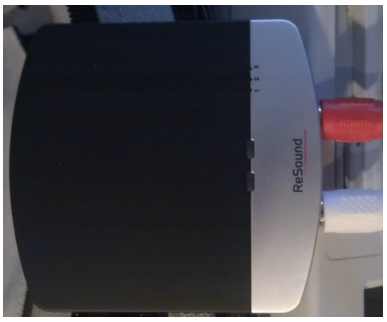


Figure 4.1: ReSound Unite^{TM}TV

software model of the DSP. While this is easy, it do not give a full picture of the behaviour, due to the fact the model do not have the full hardware or software support and it does not get interrupts, which pollute the access patten.

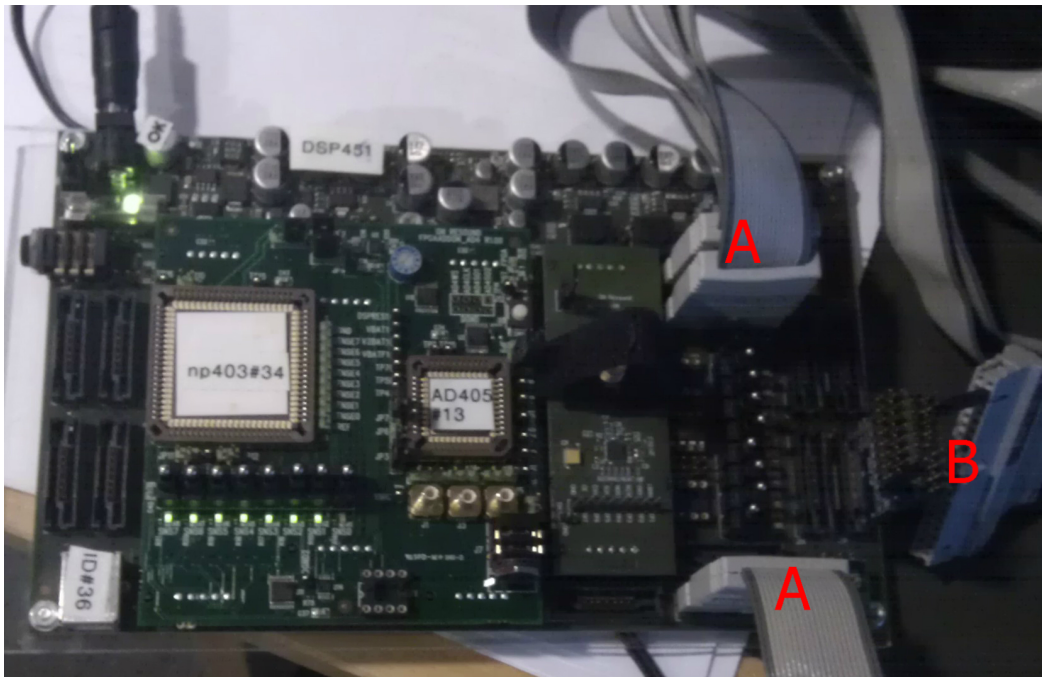


Figure 4.2: FPGA board with logic analyser connected

The wireless part of the hearing aid needs an input, for this the ReSound Unite^{TM}TV shown in figure 4.1 is used.

4.3 Signal of interest

Below is listed the signals required to check for memory access and which kind of access it is. Along with the signal is a short explanation of why the signal is needed.

- **Program memory address bus:** This is needed in order to see if the same address is read several times
- **Program memory data bus:** The data is needed for two reasons, to look for loop instruction and to generate switching activity for the power calculation done in VHDL.
- **Program memory enable signal:** The enable signal tells whether a memory access is actually needed.
- **Program memory write/read signal:** The cache has to act differently for a read compared to a write.
- **DSP halt signal:** This signal is needed to verify whether the DSP is a sleep or not
- **DSP clock:** The DSP can take a shout interrupt without having to wake up (halt signal change), for that reason this clock is needed to be able to see memory access when the DSP is in halt.
- **Clock:** This clock is intended for sampling in the logic analyser.

4.4 Change in the VHDL and synthesis for FPGA

The signals of interest can all be found in the entity cpuCore in the VHDL code. Within this entity the signal of interest will be copyed to a register, with the exception of the DSP clock as it is a gated version of the clock used to clock the signal with.

Listing 4.1: VHDL code for gathering the signals

```
1 test_port_o     <= test_port_s;
2 test_port_s(64) <= reset_ni;
3 test_port_s(65) <= clkDSPAclk_i;
4 test_port_s(81) <= clkInt_i;
5 process (clkInt_i, reset_ni)
6 begin
```

```
7   if reset_ni = '0' then
8     test_port_s(95 downto 82)  <= (others => '0') ;
9     test_port_s(80 downto 68)  <= (others => '0') ;
10    test_port_s(63 downto 0)   <= (others => '0') ;
11  elsif rising_edge(clkInt_i) then
12    test_port_s(15 downto 0)   <= tAddrP;
13    test_port_s(47 downto 16)  <= pmIn;
14    test_port_s(48)            <= PRamEn or PRomEn or
          debugRomRamEn;
15    test_port_s(49)            <= PRamWr;
16    test_port_s(66)            <= cHalt;
17    end if;
18  end process;
```

The output of the register plus the clocks are then led up through the hierarchy to the top component.
In order to synthesise the design with the change made, GN Resounds flow is needed. Before starting the synthesis, the port created in 4.1 needs to be assigned to the right pin on the FPGA, which is done in the constraint file. Synthesis is done using Synplify and Xilinx. Synplify controls the flow doing the first place and route, parsing it to Xilinx and finalising the Synthesis itself. In order to generate the bit file, Xilinx ISE is needed.

4.5 Setting up the board and generating a trace

With the bitfile created in section 4.4 on the FPGA board the work is not done. The board needs an OS[1] and an algorithm pack. This algorithm pack needs to be set up to run with a microphone and wireless as input. The reason for this is to make a trace in which the DSP can fetch data from the AD[2] converter, this is done as interrupt. After fetching the data the DSP has to process it. A streamer (see section 1.5) is set up to stream music, which will generate interrupt where the OS has to handle the data coming in from the wireless radio.

This setup is going to run like a fully functional hearing aid, and as such it will generate a trace which will be similar to what happens in a hearing aid, used in an every day situation.

[1] Operating System
[2] Analog-to-digital

In order to create and save a trace the following steps are needed.

1. Connect the logic analyser to the FPGA board, see figure 4.2. A is the data signal and B is the clock (clkInt_i) which is used as sample clock in the logic analyser.
2. Start the logic analyser.
3. Boot the FPGA board.
4. Set-up of the algorithm pack to process the data. This should only be done the first time, or when new settings are wanted. For this a GN ReSound program was used. The set-up is shown in appendix A.
5. Start streaming data via ReSound Unite^{TM}TV.
6. Reset the FPGA board.
7. Pair the FPGA board with the ReSound Unite^{TM}TV. Think of this a pairing two blue-tooth devises.
8. Save a set of sample.

The set-up of the algorithm was chosen in cooperation with the Chief System Architect at GN ReSound. Four set of samples was make, two with the algorithm running and wireless streaming and two with out the wireless streaming.

An overview of a trace can be seen in figure 4.3, and as can be seen the DSP is in halt (active high) for a significant part of the time. While the DSP

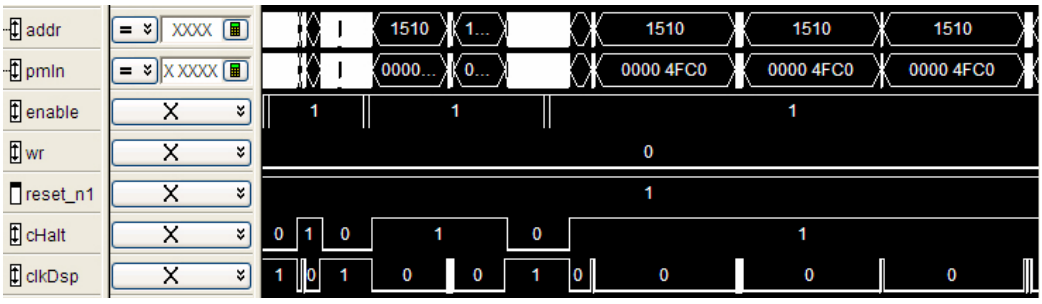


Figure 4.3: Trace showing halt

are in halt, there are time when it runs. This is doing short interrupts, like fetching data from the AD converter, which is an one instruction interrupt.

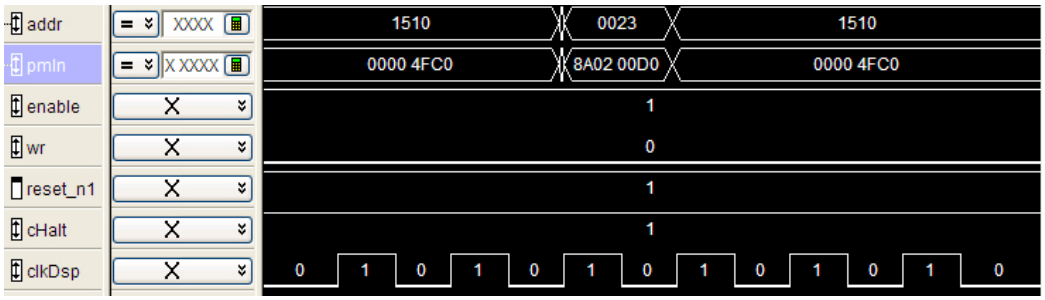


Figure 4.4: Trace showing one instruction interrupt

As shown in figure 4.4, the execution of this one instruction costs six memory accesses, in order to fill the pipeline.

In order to make the model run faster, and because the model's interface was designed before the trace file was created. The trace file is cleaned for data where there is not any memory access and data which ain't necessary for the model.

This is done in a Perl script, it could be done in the model. The reason for cleaning it in advance is that reading in ascii text files into the model takes time. This is only a issue under development, as the model was executed more then ones.

Listing 4.2: Raw trace file

```
1 "SampleNumber","addr","Time","pmIn","enable","wr","reset_n1","
    cHalt","clkDsp","clkInt\_i"
2 -523437,1510,-32.714952 ms,00004FC0,1,0,1,1,1,0
3 -523436,0023,-32.714892 ms,8A0200D0,1,0,1,1,1,0
4 -523435,1510,-32.714828 ms,00004FC0,1,0,1,1,1,0
5 -523434,1510,-32.714768 ms,00004FC0,1,0,1,1,1,0
6 -523433,1510,-32.714704 ms,00004FC0,1,0,1,1,0,0
7 -523432,1510,-32.714640 ms,00004FC0,1,0,1,1,1,0
8 -523431,1510,-32.714580 ms,00004FC0,1,0,1,1,1,0
9 -523430,1510,-32.714516 ms,00004FC0,1,0,1,1,1,0
10 -523429,0024,-32.714452 ms,AA8200D4,1,0,1,1,1,0
11 -523428,1510,-32.714392 ms,00004FC0,1,0,1,1,1,0
```

Listing 4.2 shows the raw data from the logic analyser and listing 4.3 shows the trace file after cleaning it. The cleaned file only contains data when there is a memory access.

Listing 4.3: Formatted trace file

```
address;data;wr;en
00001510;00004FC0;0;1
00000023;8A0200D0;0;1
00001510;00004FC0;0;1
00001510;00004FC0;0;1
00001510;00004FC0;0;1
00001510;00004FC0;0;1
00001510;00004FC0;0;1
00000024;AA8200D4;0;1
00001510;00004FC0;0;1
```

4.6 Sub conclusion

When the author started this project, the expectation was not that this part should result in major issues, but it turned out to be a confirmation that Murphy's law still is in existence, "Anything that can go wrong will go wrong". The challenges have been countless including for example user rights, tool problems and issues in the version of the DSP the author was meant to use.

While this did set back the project, it gave the author a change to get familiar whit the design of the DSP, the tools used with in GN ReSound and debugging.

Two different traces was saved, both with the same algorithm setting, one where the hearing aid receives and processed wireless audio and one whit out any wireless connection. Through out this thesis if nothing else is mentioned it is the trace with a wireless connection which is used.

It was planed to create traces from the data memoirs. But do to the complication in getting the FPGA flow to work, only traces from the program memory was made.

Chapter 5

Model of the memory system

In this chapter a model to calculate the power used in the memory system will be presented. This model has been implemented in C++. It is possible to measure, or rather count, the required access requests to both the main memory and the cache which in return can be used to calculate the power of the system. As input the model will take a trace file, of the format shown in listing 4.3.

The model should act as the system cycle to cycle, or as the is not pipelined it is going to act as the system, memory access to memory access.

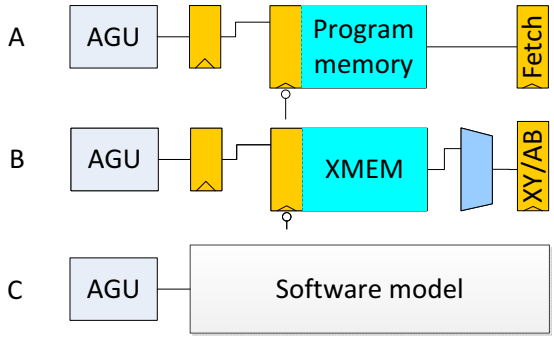


Figure 5.1: Memory model used in the C++ model

The model the model can be used on both the program and data memories. As mentioned in chapter 3.1, they are the same in context of memory accesses. Figure 5.1 A: Shows the program memory, B: The data memory and C: The cache and memory system, from a software perspective.

In this chapter result from the model showing the power used is presented, the cost function for calculation the power used is not presented on till chapter 6.0.1.

5.1 Overview and structure of the model

On a top level the model is simple. A block diagram is shown in figure 5.2, main is block from where all action is initiated. Main will initialize a cache controller, either with no cache, a cache or a loop cache. These cache controller will then initialize the RAM, ROM and cache needed, which the initialize what they need and ect.

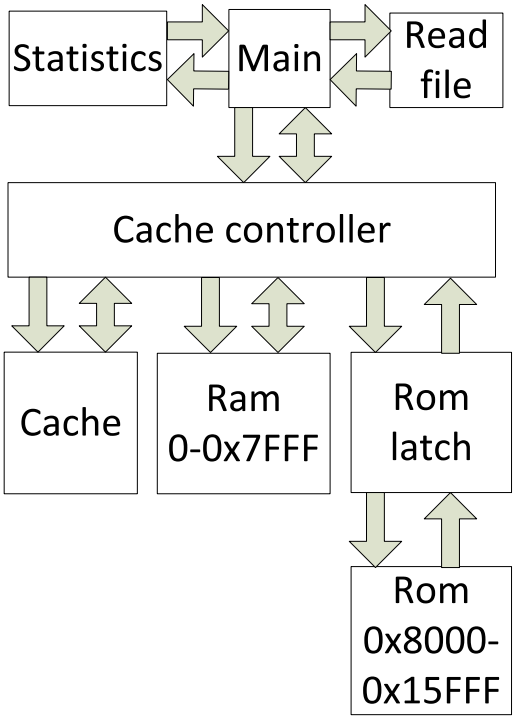


Figure 5.2: Block diagram of the model

A class diagram for the cache is shown in the appendix figure B.2, as can be seen the cache initialize a object of the class Ram, a object of the class Rom, and a 2 dimensional array of object of the class CacheLine.

A flow chart of mains functionality is shown in figure 5.3.

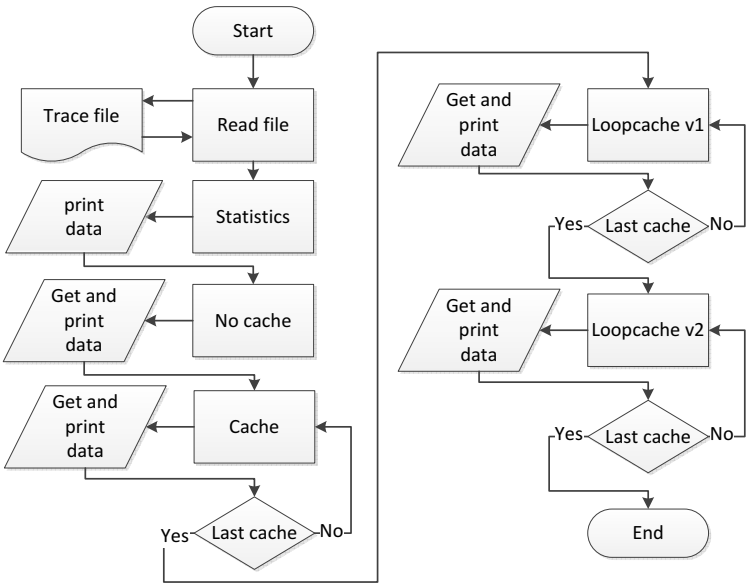


Figure 5.3: Simple flow chart of the model

In order to explain what happens in main, a step by step explanation of a part of the flow chart is given below.

1. Read in a trace file, this is saved in a array.
2. Analyse the trace, size most used address etc. The print out the result.
3. Initialize a memory system with out a cache and run the trace through it, this is done as to have a reference.
4. Initialize a memory system with a cache of some configuration and run the trace through it.
5. Format and print data from this run.
6. If there are more configuration of caches which need to be runned return to 4.
7. 4,5,6 for both loop caches
8. End.

The above explain the function of main and the flow through the model. In the following sections, each sub part (like statistics) will be explained.

5.2 Statistics done on the input file

When the trace file is read in, the model pulls out some statistics. The numbers presented in this section are from a trace file collected when streaming sound to the hearing aid.

- **Trace file size: 715816.** Number of memory accesses.
- **Unik address: 6744.** This is what in chapter 2.2 is referred to as Compulsory miss.
- **Read: 715730.** Number of memory read.
- **Write: 86.** Number of memory writes.
- **Do instructions: 7008.** Number of instructions in the whole trace file, which is a Do instruction, remember chapter 3.4.
- **Individual Do instructions: 649.** Number of different Do instructions.
- **Do instructions count: 471979.** Number of instructions in the Do loop.
- **Number of address under 0x400:** 21466. These map to the peripheral, but the model handle them as all other RAM accesses.
- **Number of address under 0x8000:** 248067. Access to RAM.

Some of these numbers are more interesting then others. For example the number of writes being so low should not come as a surprise, since the trace file is from the program memory. That more than 65% of the instructions is inside a loop is an information which is useful. This is in fact what in chapter 2.3 was called Spatial Locality.

Figure 5.4 shows the most used addresses, what can be seen in the figure is that the most used address is at address H'1510. The instructions at that address is in fact a nop[1], and it is the nop shown in figure 4.4.

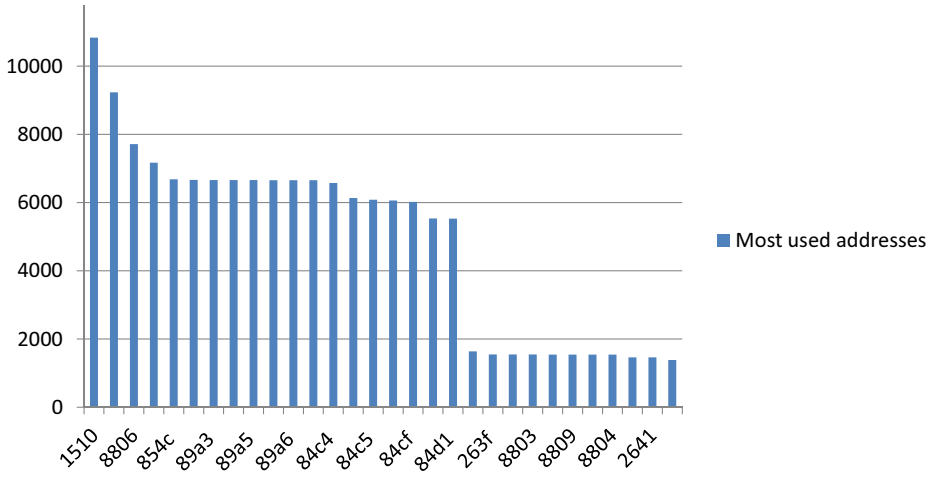


Figure 5.4: Addresses most used

In figure 5.5 is shown the distribution of number of Do loop instructions over the size of the loops. This is a dynamic distribution, as the same Do loop instructions may be repeated several times. As expected there are a major part of the instructions, with a small size. These loops are for the major part filter.

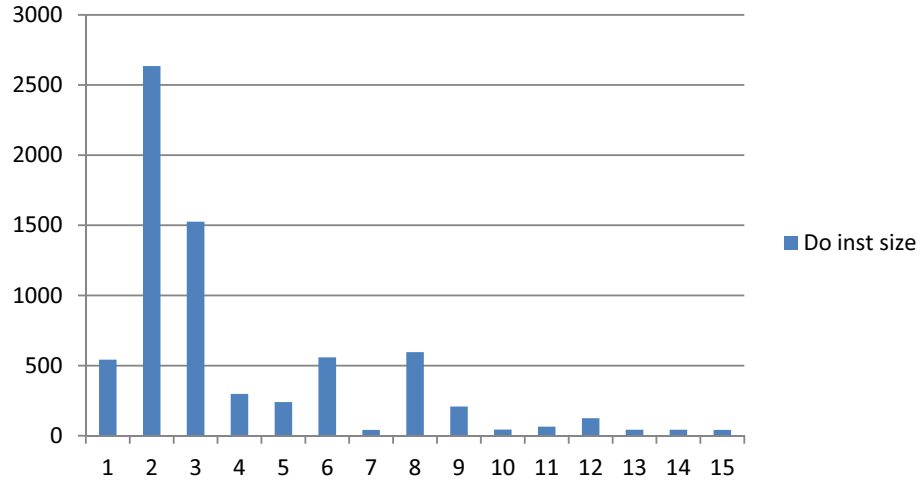
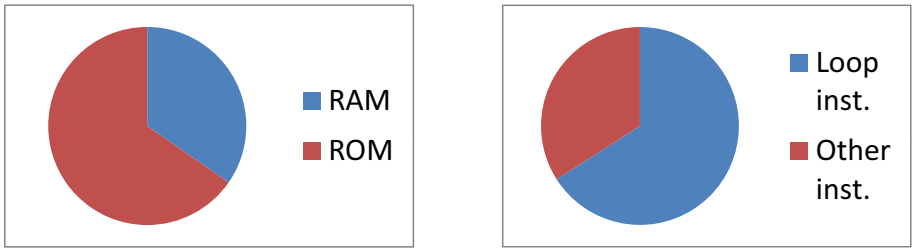


Figure 5.5: Number of Do instructions with a given size

[1]A instructions not doing any thing



(a) RAM vs. ROM accesses (b) Inst. in a loop vs. not in a loop

Figure 5.6

Figure 5.6 (a) shows the division between RAM and ROM memory accesses. The major part of the memory access is in the ROM and as this is a trace from program memory this was expected. The figure 5.6 (b) shows the division between instructions in a loop and those not in a loop, and here it is interesting that more than $\frac{2}{3}$ of all instructions fetched is in a loop.

5.3 Model of the memory

I order for the cost function, coming in chapter 6.0.1, to be realistic, the memory access has to access the memory it was meant for. In chapter 3 figure 3.3 is shown that the memory space is divided into two, a RAM and a ROM part. As these two memories are not identical in use of power they have to be separate classes.

5.3.1 Model of the Ram and Rom

A RAM implemented in software is just an array, once the control signal is removed. The class RAM has a function for reading and one for writing, the control desiding if it is a read or write is in the cache controller.

The ROM is built in much the same way, with the exception of a latch, see figure 5.2. This latch is used to save power, this is what was presented in chapter 2.6.2.

Listing 5.1: C++ ROM latch code

```
1  unsigned int Rom::Read(unsigned int Address){
2    unsigned int data;
3    if((Address & ~this->AddressMask) == this->CurrentLineAddress
         ){
4      data = line[Address & this->AddressMask];
5    }else{
6      this->CurrentLineAddress = Address & ~this->AddressMask;
7      for(int i=0; i<8; i++){
8        this->line[i] = this->array[this->CurrentLineAddress + i];
9      }
10   }
11   return data;
12 }
```

The implementation of the latch can be seen in listing 5.1. In the listing line 3 there is a check if the current address falls into the address space of the latch. If not, a new line is fetch and saved in the latch.

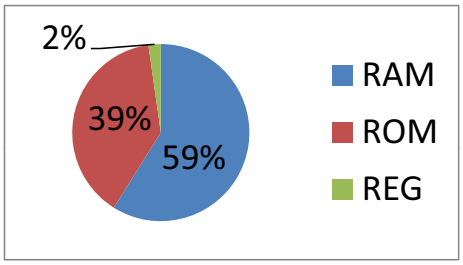


Figure 5.7: Power for RAM vs. ROM

Shown in figure 5.7 is the power used by a system without a cache. Reg is the power used by the register PC and Fetch in figure 3.4. The thing to notice is that even though the RAM only stands for $\frac{1}{3}$ of the memory access, it uses $\frac{2}{3}$ of the power. The cost function for the power will be presented in chapter 6.

5.4 Model of the caches

In the model there are 2 different cache types, each in different configuration. There is a cache like the one presented in chapter 2.3, and two different loop cache, designed with inspiration from chapter 2.5.2.

5.4.1 Model of a cache

In order to test a variety of configurations, remember in chapter 2.3.2, the cache memory need to be configurable in all the possible combinations. This is done by having a class for the cache line, shown in figure 5.8 and listing 5.2.

Figure 5.8: Block diagram for Class CacheLine

Listing 5.2: Code for Class CacheLine

```
unsigned int Tag;
unsigned int * Data;
bool Valid;
```

Creating a two-dimensional array of an object, of the class CacheLine, results in a cache with one dimension for the set, one dimension for the cache line and in each cache line a tag, a valid bit and a block (array) of data word. This is the data part of the cache.

The cache controller is implemented as the class Cache, for which a simple class diagram is shown in figure B.2.This class instanciered one object of the class RAM, one of the class ROM and a two-dimensional array of class CacheLine. The function of the class is described as a flow chart shown in figure B.1 in the appendix.

Listing 5.3: C++ code for tag lookup

```
pair<bool,unsigned int> Cache::TagLookup(unsigned int Index,
    unsigned int Tag){
  pair<bool,unsigned int> hit (false,0x0);
  for(unsigned int i=0;i<this->set;i++){
    if( (array[i][Index].Valid) && (array[i][Index].Tag == Tag)
        ){
      hit.first = true;
      hit.second = i;
      this->CacheHit++;
    }
  }
  return hit;
}
```

The flow chart in figure B.1 has been implemented as a nest of if else statement, with call to function like the one shown in listing 5.3.

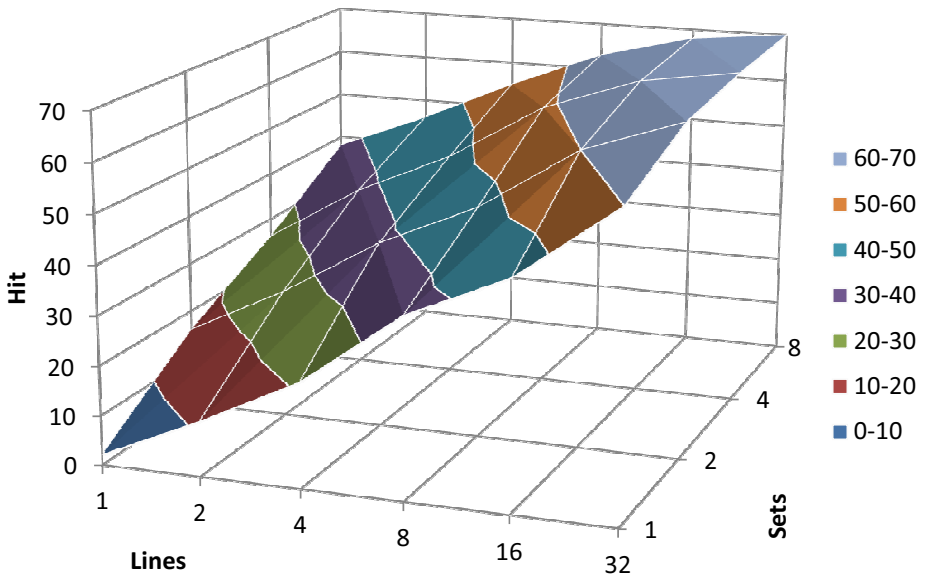


Figure 5.9: Hit rate as function of cache lines and sets

In figure 5.9 is shown the hit rate in % as function of the number of cache lines and the number of sets. One thing to observe here is that the gain from going from a direct mapped cache to a full associative cache is not that big.

Think back to The Three C's in chapter 2.2, the difference from the full associative cache to the direct mapped cache is Conflict miss. The gain in

Cache	Compulsory	Capacity	Conflict
Direct mapped	6744	443739	12458
Full associative	6744	443739	0

Table 5.1: The Three C's in a 8 word cache

a Full associative being $\frac{12458}{715816} = 1,74\%$ more hits, this do not seam high enough, when compared to having to do 8 compare instead of 1 for each memory access.

Cache Write Policies

For the program memory this is not so important, as the amount of writes are limited (86 out of 715816). The model should be able to take traces from the data memory as well, and in that case it becomes important. From a power saving point the best choice would be a Write-Back Cache, but it is not possible in this system as a Write-Back Cache can end up stalling memory accesses while writing back to main memory. A Write-Through Cache on the other hand will not stall memory accesses, in this kind of cache the cache can be updated or invalidated. In the model it is always updated, which makes the valid bit the model have useless, as soon as the cache has been filled. The author knows this but has chosen to leave the valid bit, just in case there would be time to add a invaliding cache function at some point.

Cache replacement policy

When having more than 1 set in a cache leads to the question, what to replace when writing new data to the cache. There are a lot of replacement policies, what they all try to do is to predict what data is needed in the future.

- **Random:** As the name suggests, this policy chooses a set random.
- **Least Recently Used:** Here there is kept track of when an item is used, leading to complex control logic as number of set goes up.
- **Round-robin:** Choose item i now and i+1 next time.
- **Random Round-robin:** As a Round-robin but is not updated when used but at a random time.

These are the four cache replacement policies implemented in the model.

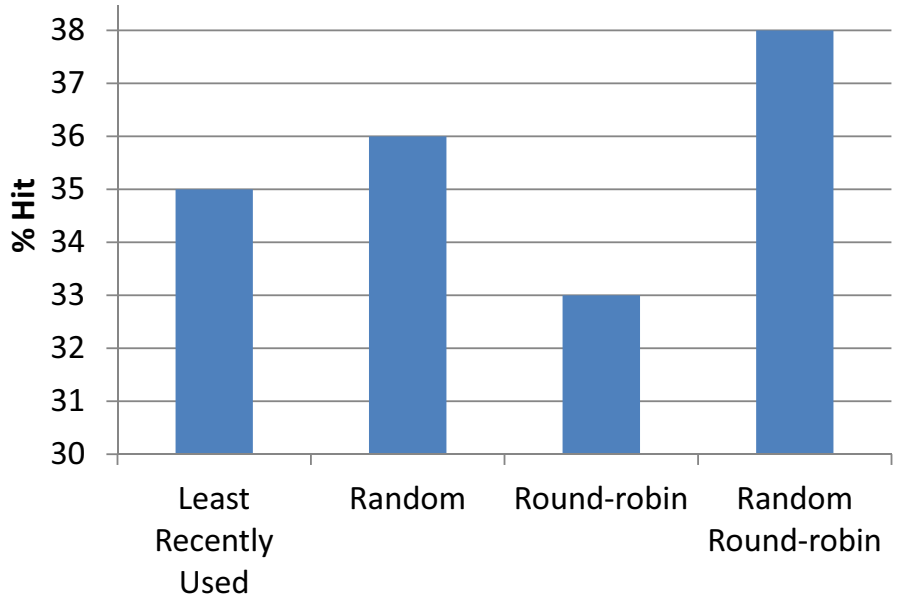


Figure 5.10: Hit as function of replacement policy in a 4 line 2 way cache

Figure 5.10 shows how the hit rate changes in a cache as a result of changing the replacement policy. One thing to notice is that both Random Round-robin and Random is better than the Least Recently Used, all though not by much. This means that the access pattern of the used trace file, is not in a way where throwing out old data is necessarily the best choice.

5.4.2 Model of a loop cache

In chapter 5.2 was shown that if the code is looked at dynamic is has a lot of instructions which are in a loop. This should make a loop cache a good solution. There are some aspects which need to be clarified before designing a loop cache. Se chapter 2.5.2 or [12.

How to locate a loop?

The instruction set hold an instruction for a loop, see 3.4. By looking for the opcode for this instruction in the data word the start of a loop can be located. The instruction holds the last address of the loop as well, giving the start, end and by subtraction the size.

How to locate change in the instructions flow?

Cof[2] is possible since jump, call ect. are allowed with in a loop, this mean that some way to detect if the address coming from the AGU still is in the loop is needed. There are several solutions for this. For example check all data words for opcode equal to all instruction capable of jumping out of the loop. In this case some link to the link register holding the return address is needed. In this thesis the cof has been chosen as $Start_{adr} \geq Current_{adr} \geq End_{adr}$, meaning if this is true there is no change of control flow.

What to do in case of cof?

When leaving the loop, all data will be fetched from main memory, until returning to the loop. If the cof resulting in leaving the loop is a interrupt with a reasonable size and it includes loop, it may be preferable to discard what is in the loop cache and start looking for a new loop.

- **Counter:** Here a counter is used, counting the number of instructions fetched when outside the loop. If it reaches the chosen value, the loop cache resets and starts looking for a Do loop.

- **Do loop:** When there is a cof and data is fetched from addresses outside the loop, the loop cache looks for Do instructions in these data words. If there is a Do instruction it discards what it has and starts loading the loop cache again.

Loop size vs. loop cache size

If a loop can not fix with in the given size of the loop cache, two things can be done.

First - only a part of the loop is saved and each time the part is accessed, the data word is fetched from the loop cache. The rest of the time the data word is fetched from the main memory. This means at least some part of all loops is cased, but the cost is added control.

Second - As the start and end of the loop is available, the loop cache can just discard loops too big to fit in the loop cache, this is the approach used in this thesis. If the loop is bigger than the cache, it is posibel to have a part of the loop in the cache, down side for this is more control logic. The approach

[2]Change of control flow

of discard loop bigger than the cache was chosen, because the major part of the loops are small, 8 words or less as seen in figure 5.5.

Version 1, Counter

This version of a loop cache uses a counter to reset which is shown in the flow chart in figure B.3 which do to size is in the appendix. As the cache presented in chapter 5.4.1 the flow chart has been implemented as nested if else statements. The code for resetting the loop cache is shown in listing 5.4

Listing 5.4: Reset cache in counter version

```
1 if(Address.address == this->End || this->stop > 31){
2   this->CacheOn = false;
3   this->CacheLoad = false;
4 }
```

In listing 5.5 is shown the code for finding a Do instruction, once a Do instruction is found, the end address is extracted and the start address and loop size is calculated.

Listing 5.5: Finding a Do instruction

```
1 data = Read(Address.address);
2 unsigned int inst = data & 0xC03E0000;
3 if( inst == this->DoLoopInst){
4   this->Last = data & 0x0000FFFF;
5   this->Start = Address.address + 1;
6   unsigned int size = this->Last - this->Start;
7   if( size <= this->CacheSize+1){
8     this->CacheLoad = true;
9   }
10 }
```

The cache is only set in load state, in the case where the entire loop fits in cache.

Version 2, Do instruction

Flow chart is in the appendix in figure B.4. Most of the functionality, and thereby the code, is the same as for the version 1 loop cache. The place where this differs is how the cache is reset. This is identical to listing 5.5 with the addition $this- > CacheOn = false$; in the case where a new loop fitting in the cache is found.

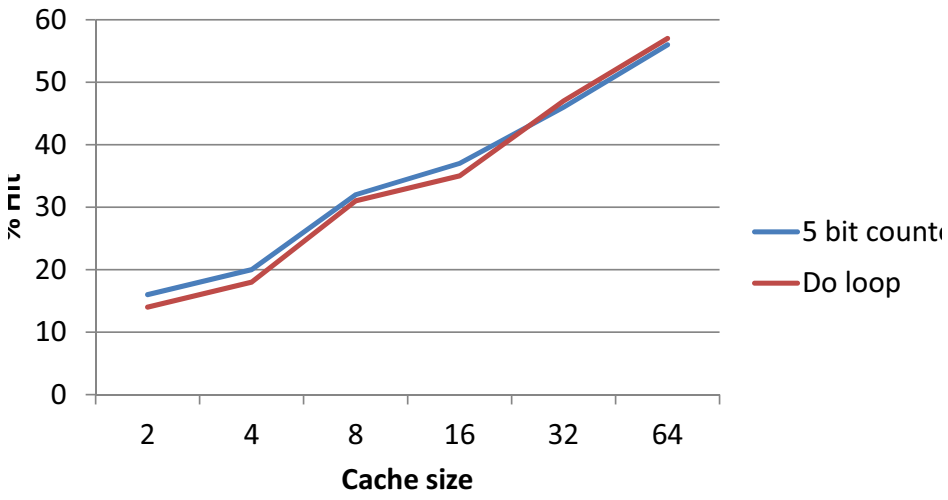


Figure 5.11: Hit as function of size in loop caches

Shown in figure 5.11 is a comparison of the version of loop caches implemented.

5.5 Sub conclusion

The result shown in table 5.2, leads to the conclusion that it is not necessarily better to have more ways if the total cache size remains the same.

Lines	Direct	2-way	4-way	8-way	16-way
1	2	16	26	38	43
2	11	26	38	46	52
4	21	38	47	55	59
8	36	47	59	63	65
16	45	60	66	68	69
32	60	67	69	70	71

Table 5.2: Hit rates for different configurations of a cache using Random Round-robin replacement policy

In table 5.3 is shown the hit rates for the two loop caches. By comparing a 16 word direct mapped cache with a loop cache of same size (45% vs. 37% or 35%), it is easy to see that a cache gets hits on data which are not in a Do loop. This is for example the nops show in figure 4.4.

Size	Counter	Do inst
2	16	14
4	20	18
8	32	31
16	37	35
32	46	47
64	56	57

Table 5.3: Hit rates for the two loop caches

Although a high hit rate is desirable, a higher hit rate will in most case lead to a higher power use. In the next chapter the power cost function will be presented and this will become obvious.

Chapter 6

Power cost function in the model

In this chapter the power cost function will be presented along the number used in the model. At the end of the chapter results from the model including the cost function will be presented.

6.0.1 Cost function

The cost function is divided into two part, one for the caches and one for RAM and ROM.

Caches

The cost function only covers the dynamic power dissipation. The energy for per operation is shown in equation 6.1.

$$E_{op} = \int_{t_{op}} vi \, \mathrm{d}t \;\; [J] \tag{6.1}$$

This is given from the technology vendor for each component in different strengths, component meaning gate, flip flop, buffer, etc. This will be explained in chapter 6.0.2, for now it is enough to know that this data is available.
These numbers are used to calculate power used in a memory access, equation 6.3 shows the fundamental idea.

$$E_{ma} = \sum E_{\text{ for all components used in this memory access}} \tag{6.2}$$

ma is memory access, E_{ma} is the energy per memory access. In the flow chart for a cache shown in figure B.1 each decision or process is in hardware

done by use of some of the components. For example the sub part of the flow chart, which is shown in figure 6.1. The way to deside if there is a cache hit or not is by comparing the tag of the current address with the tag saved at the index of the current address.

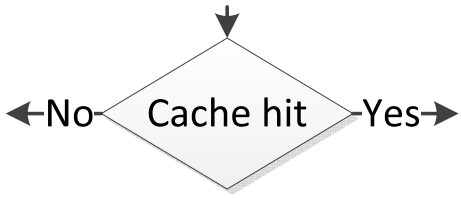


Figure 6.1: Tag compare in the flow chart

The address is 16 bit wide, and in a 4 line cache the index is 2 bit, leaving 14 bit to the tag. To do this compare 14 XOR and 14 OR is used, as shown in figure 6.2.

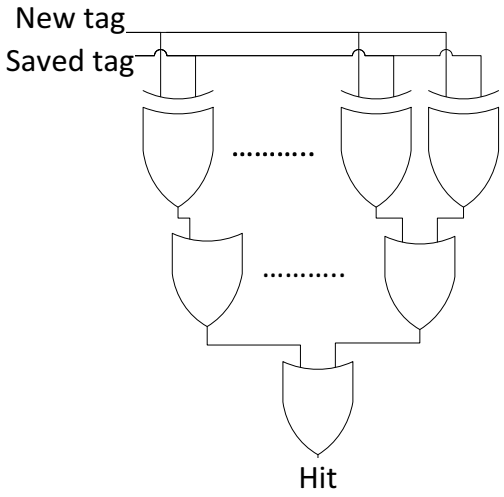


Figure 6.2: Tag compare as gate

$$E_{compare} = a * \sum (14 * E_{XOR} + 14 * E_{OR}) \tag{6.3}$$

Where a is an activity factor. This is of course only a small part of a memory access. For example before doing the compare the saved tag has to be read from the memory array, costing switching activity on the mux, think back to how the memory array is designed, figure 3.5.

In chapter 5 it was presented that the model takes one memory access at a time and then the next memory access. How to calculate the energy for one memory access was just presented, to get the entire energy, all the separate memory accesses energy only needs to be sum op. This is shown in equation 6.4.

$$E_{total} = \sum_{i=0->Trace\ size} E_{ma}(i) \tag{6.4}$$

E_{total} is the energy used for running the entire trace file, $Trace\ size$ is the number of memory access in the chosen trace file and E_{ma} is the energy for the given memory access. This is the number the model has as output.

The tool used to do the power simulation on the VHDL caches return the result as watt, the next part of this section will explain how to compare this number. To get the average energy used per memory access E_{av-ma}, equation 6.5 is used.

$$E_{av-ma} = \frac{E_{total}}{no.of ma} \tag{6.5}$$

With the average energy per memory access and one memory access per clock, equation 6.6 is used for that.

$$P_f = E_{av-ma} * f \ \ [W] \tag{6.6}$$

RAM and ROM

The cost function for the RAM and ROM are the same as for the caches. The difference is that the RAM in just one component and the ROM is two, one for hitting the latch and one when missing it, think of figure 5.2. The total energy used for the RAM is expressed in equation 6.7, and for the ROM in equation 6.8.

$$E_{RAM\ total} = E_{one\ RAM\ access} * no.of\ RAM\ access \tag{6.7}$$

$$\begin{aligned} E_{ROM\ total} = &[E_{ROM\ latch} * no.of\ ROM\ latch\ hit] \\ &+[E_{ROM} * no.of\ ROM\ latch\ miss] \end{aligned} \tag{6.8}$$

This equation is possible because the model, as explained in chapter 5 keeps track of memory accesses.

6.0.2 Data from technology vendor

In chapter 6.0.1 the function for calculation of the energy cost for a single memory access to the cost for a whole trace file was presented. These equations were based on, that the value from equation 6.1 was available.
In listing 6.1 is shown a part of the data sheet from the RAM compiler used by GN ReSound.

Listing 6.1: From RAM compiler data sheet, 2048x32 CM=8 Bank=4

Process condition	Worst		Typical		Best	
Description	RD	WR	RD	WR	RD	WR
Power Dissipation (uW/MHz)	2.652	2.701	3.464	3.498	4.444	4.467

The typical is the data used in the model, $3.464\frac{\mu W}{MHz}$, this is E_{op} and what is needed in the model is E_{pc} or in the terminology of this thesis E_{ma}. Using equation 6.5 result in the energy use per memory access $E_{one\ RAM\ access}$.

$$E_{one\ RAM\ access} = \frac{E_{op}}{no.of ma} \tag{6.9}$$

Here it is important to notice that the data for the RAM is given at the same volts as the hearing aid is runned at.

$$E_t = \int_0^t vi\,\mathrm{d}t \;=\; \frac{1}{2} C_L V_{DD}^2 \tag{6.10}$$

The components used in the caches, flip flop gate, ect. has like the RAM its power dissipation given in $\frac{\mu W}{MHz}$, a example is shown in figure 6.1. While the RAM data was given for 0,72 volt, this data is for 1,2 volts, which means they have to be adjusted. In equation 6.11 is shown that the energy scale is quadratic with the voltage.

Process Technology TSMC CL013G-FSG(HVT)						AND2
AC power						$\frac{\mu W}{MHz}$
Pin	X1	X2	X4	X6	X8	X12
A	0,0059	0,0082	0,0136	0,0192	0,0245	0,0365
B	0,0068	0,0094	0,0162	0,1221	0,0281	0,0431

Table 6.1: Data sheet for AND2

$$E_{at\ 0,72} = \frac{0,72^2}{1,2^2} * E_{at\ 1,2} = 0,36 * E_{at\ 1,2} \tag{6.11}$$

All components are chosen at strength X1 as very few of them have more then 1 or 2 input to drive, only the AND gate used in the model for local clock gating is bigger, strength X4, as they have to drive a local clock.

The ROM is a hand built ROM and there are no data sheets, GN ReSound informed me that it uses 100 mA for reading a new line and 10 mA when reading from the latch. Both at 0,72 volt and 16MHz.

All the constants for energy per clock or access is in listing 6.2.

Listing 6.2: Power constants in the model

```
1  RomReadPower      = 100.0*0.72/(16*1e6);
2  RomLineReadPower  = 10.0 *0.72/(16*1e6);
3  RamReadPower      = 3.460/1e6;
4  RamWritePower     = 3.460/1e6;
5  DFFx1Power[]   = {0.0084*0.36/1e6, 0.0178*0.36/1e6, 0.0101*0.36/1
       e6};
6  MX2x1Power[]   = {0.0116*0.36/1e6, 0.0101*0.36/1e6, 0.0110*0.36/1
       e6};
7  XOR2x1Power[] = {0.0066*0.36/1e6, 0.0128*0.36/1e6};
8  OR2x1Power[]   = {0.007*0.36/1e6, 0.0077*0.36/1e6};
9  AND2x1Power[] = {0.0059*0.36/1e6, 0.0068*0.36/1e6};
10 AND2x4Power[] = {0.0136*0.36/1e6, 0.0162*0.36/1e6};
```

There is one more constant which is used in the cost function and that is the activity factor. This is a statistical number which tells how often en signal changes. GN ReSound has a number from the DSP, this number will be used until the power simulation of the VHDL. $Activityfactor = 5\%$

6.0.3 Cost function implemented in the code

In chapter 6.0.1 an example of what a tag compare would look like was shown. The same example is shown in listing 6.3 as it is implemented in the code. Each place some is done...

Listing 6.3: Power in the code for a tag lookup

```
1  pair<bool,unsigned int> Cache::TagLookup(unsigned int Index,
       unsigned int Tag){
2    pair<bool,unsigned int> hit (false,0x0);
3    for(unsigned int i=0;i<this->set;i++){
4      if( (array[i][Index].Valid) && (array[i][Index].Tag == Tag)
           ){
5        hit.first = true;
6        hit.second = i;
7        this->CacheHit++;
8      }
9    }
10   //Power for compare
11   double power = 0;
12   //So on all mux
13   power += this->TagAddressSize*MX2x1Power[0]*(this->lines-1);
14   //Data ripple out
15   power += MX2x1Power[1]*(this->lines-1);
16   //Compare
17   power += this->TagAddressSize*(XOR2x1Power[0]+XOR2x1Power[1])+
         (this->TagAddressSize*(OR2x1Power[0]+OR2x1Power[1]));
18   //all set
19   power += power*this->set;
20   this->Power += power*ActivityFactor;
21   return hit;
22 }
```

The code include the part where the saved tag is read.

6.1 Result from the cost function

In this section the result will be presented, this result is based on the energy number before calabriting in regard to the data from the power simulation of the VHDL. Which is done in chapter 7.

Lines	Direct	2-way	4-way	8-way	16-way
1	101	96	90	92	120
2	97	88	83	94	147
4	88	80	83	106	211
8	80	78	87	139	336
16	79	81	108	209	575
32	83	99	155	344	1020

Table 6.2: System energy for different configuration of a cache

Table 6.2 shows the energy of a system with different cache configuration, normalized to a system with no cache, being 100. A part of this is shown in figure 6.3.

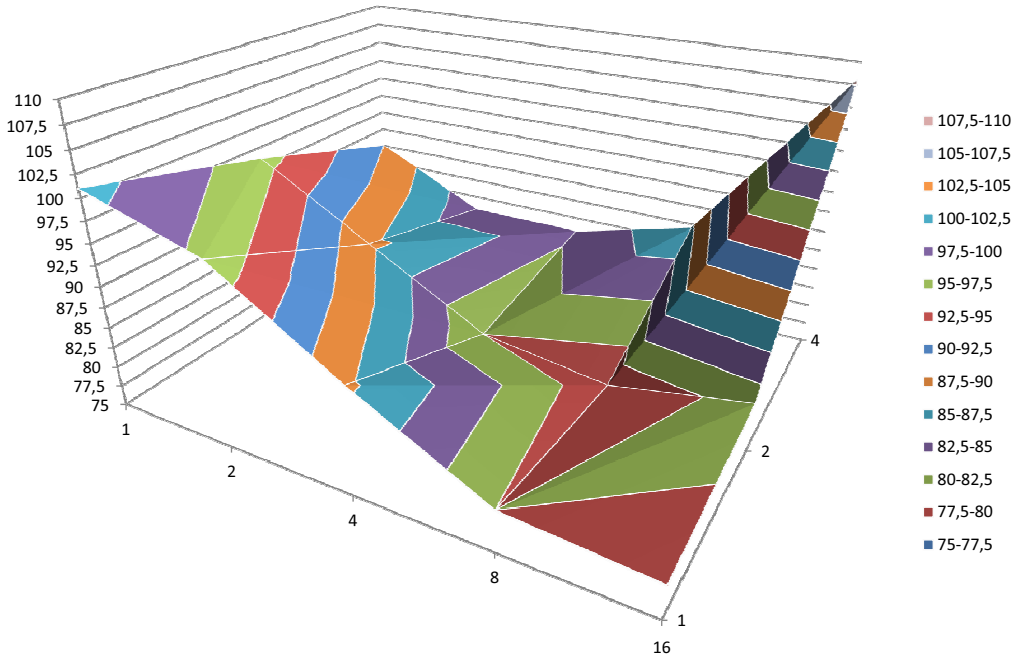


Figure 6.3: Energy as function of cache configuration

As shown, the best solution, given the data for energy before the power simulation of the VHDL, is some where around a cache with 8 or 16 words as a direct mapped or 2-way associative.

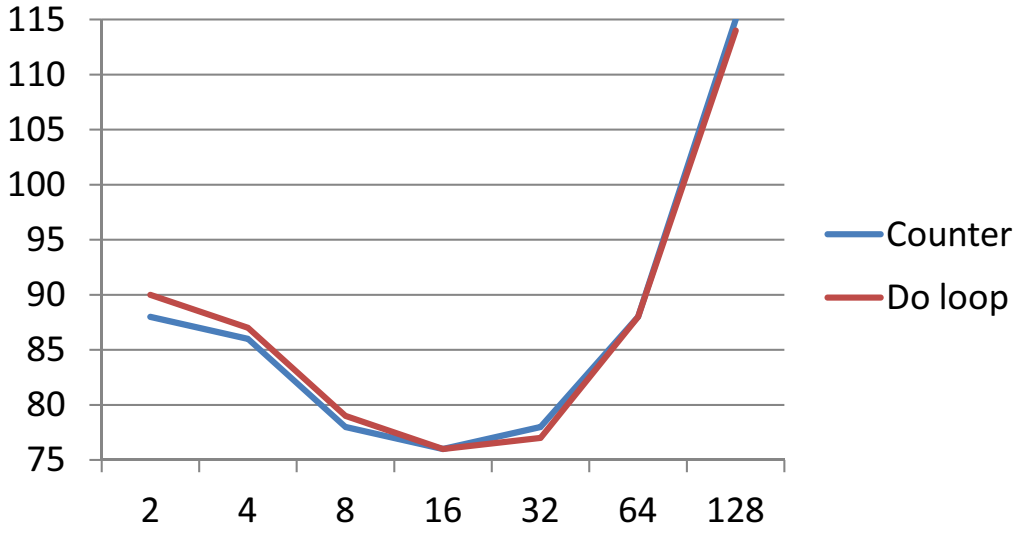


Figure 6.4: Loop cache energy relative to no cache

For a loop cache the best solution is a 16 word in any of the two versions. As they are to a great extent identical, both in design and hit rate, this does not come as a surprise.

Chapter 7

VHDL caches

The model presented in chapter 5 returns a number for the power use of a specific cache. But the author wants to make an examination on the precision of the model and at the same time calibrate it. For this reason a number of caches following the timing requirement of the GN ReSound system has be developed.

The configuration of these caches have been chosen based on what the model returns as a good choice. These caches will then be synthesized and a back annotation power analysis will be made.

7.1 Design of cache

Running the model with different traces, have made the author choose three caches, there are no loop caches among the chosen. The reason being that the author has a better feeling in regard to, how the function for the three chosen cache will be implemented as hardware. And thereby giving a better picture of the precision of the model.

- 4 line direct mapped.
- 8 line direct mapped.
- 4 line 2 way associativity.

Other caches could be used and result in just as good a result. The important thing is to choose caches in the area where the best precision is desired, i.e. around the area with the expected cache, returning the lowest power number. The power is not going to be proportional with the size of the cache, for several reasons, some of them are listed below.

1. The number of transistors in the cache will not double when size double. Control logic pipeline register etc. will not doubles.

2. In a direct mapped cache there are only one tag compare, independent of the number of cache lines.

3. Hit rate, this change with cache size, the rate of change depend on the trace files pattern.

4. Activity is going to change from cache to cache, it is hard to predict as it depends on the pattern of the input. Higher hit rate leads to more activity in the part of the cache where data is read, on the other hand lower hit rate leads to more activity in the part saving new data.

7.1.1 Diagram

The three caches selected above have to be implemented in VHDL, before doing that a diagram has been made from the flow chart shown in chapter 5. Diagram of the caches can be divided in two groups, direct mapped and 2-way associativity.

Direct mapped

The diagram shown in figure C.1 in the appendix is based on the flow chart in appendix figure B.1. As the flow chart is sequential and the diagram is parallel pipeline states, it is not directly translatable. The diagram follows the overall structure shown in figure 3.4.

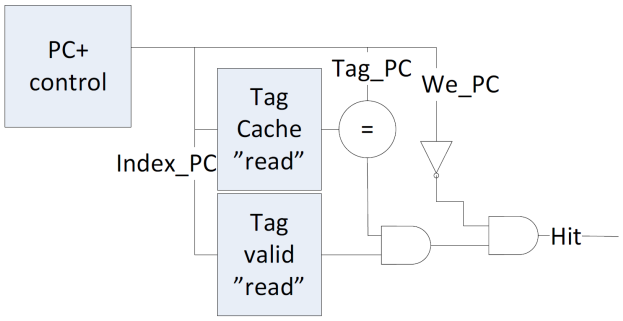
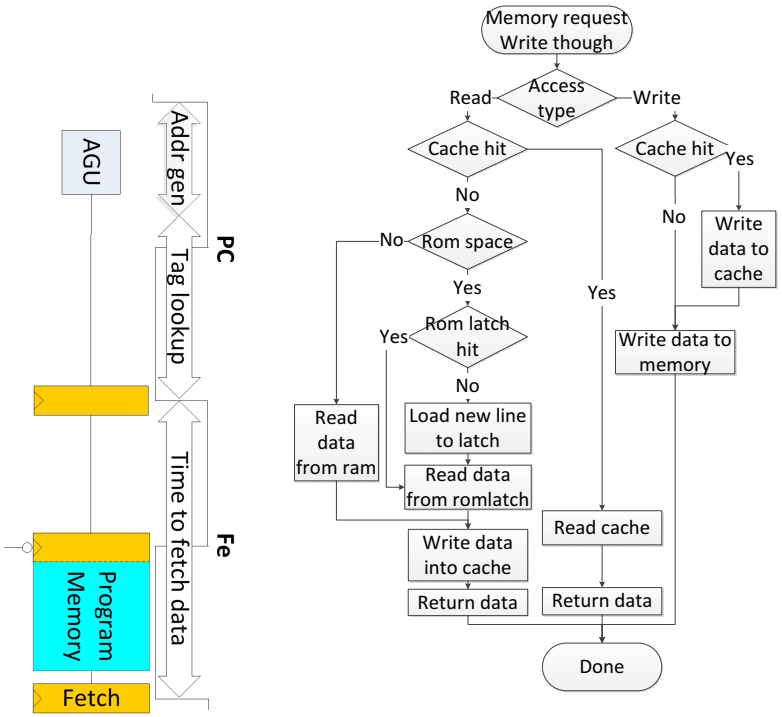


Figure 7.1: Tag compare in diagram

Figure 7.1 shows the tag compare (check for cache hit) in the diagram. In the flow chart in figure B.1 the tag compare is show as a chose named "Cache hit".



(a) From diagram (b) From flowchart

Figure 7.2: Timing diagram vs. flow chart

Figure 7.2 illustrate how the flow chart fit into the timing diagram from chapter 3.3. Dividing the flow chart into state.

- **PC:** From memory request to Cache hit, both included.
- **Fe:** Below cache hit down to return data.
- **The Fetch register:** Return data.

This is a good illustration of how the sequential flow chart and the parallel start divided diagram fit together.

2-way associativity

The flow chart is the same as for a direct mapped cache, as shown in appendix figure B.1. This is not the case for the diagram which can be found in appendix figure C.2, although the overall structure is the same, there are two tag memories and two data memories. This makes for a more complex

control system. For example a hit in tag 0 should lead to a read from data 0 and the mux has to be aligned accordingly. The VHDL codes are not going to be explained in detail. Part of the control is shown in figure 7.3, the mux is

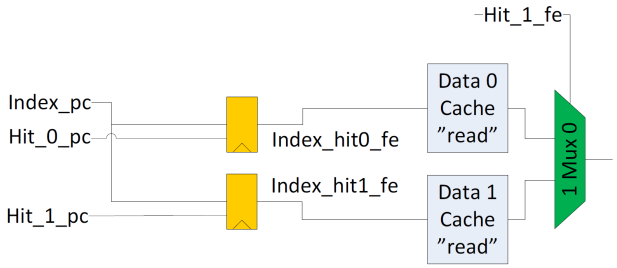


Figure 7.3: Controlling which data array to read from in a 2 way cache

used to choose between data 0 and data 1. the signals Hit_0_pc and Hit_1_pc is use to lower power by not changing control signals on the mux in the data array not used, think back to figure 3.5.

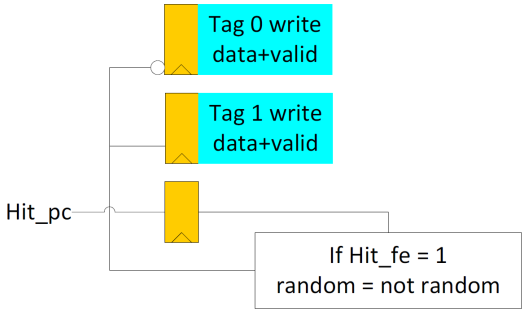


Figure 7.4: Replacement police for 2-way cache

At a miss a decision has to be made, in regard to which way to overwrite. Here the semi random round robin used in the model has been implemented, the function and use of this replacement policy is shown in figure 7.4. When ever there is a hit in any of the ways the value of *random* is inverted The reason for choosing this replacement police is that is gives a good result and that it is easy to implement in VHDL.

7.1.2 Code

The VHDL code was developed with reference to diagram seen in chapter 7.1.1. In this section the 4 line direct mapped cache will be described on a block level. The naming should be some what self explaining, as an example

the signal *addr_mem_fe_o* in the *top_4l_dm* is the address output from the cache to the memory in the Fe state.

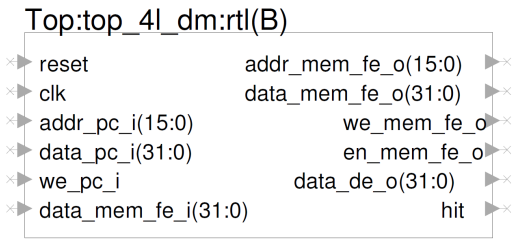


Figure 7.5: VHDL top module

Figure 7.5 shows the top module of the cache, with its input and output. Hit is only to be used by the test-bench, in order to keep track of hits. All the renaming output but data_de_o is input to the memory. The top component instantiate a tag component and a data component. The code for this module is shown in listing C.1

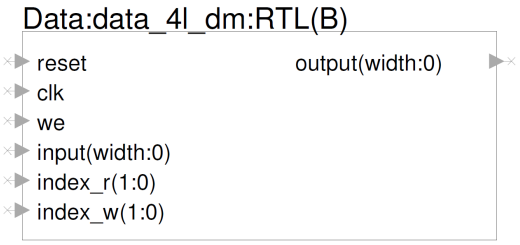


Figure 7.6: VHDL data module

The data component shown in figure 7.6 is a 4x32 bit version of the memory array presented in chapter 3.5. The code can be found in listing C.2.

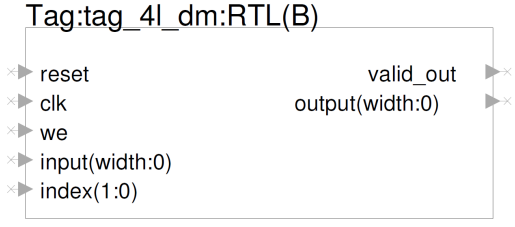


Figure 7.7: VHDL tag module

Beside the fact the size is 4x14 bit instead of 4x32 bit the tag component is the same with the addition of a valid bit for which the code is shown in

listing C.3. Whether the valid bit is necessary can be discussed. It is only used from reset until the cache has been filled the first time.

7.2 Test-bench and verification

In order to test the VHDL code and to make an activity file for the back-annotated power analysis, a test-bench has been designed. As the VHDL is developed to analyse and calibrate the precision of model, then the test-bench needs to take the same input, the trace presented in chapter 4. The format of this is shown in listing 7.1.

Listing 7.1: Input format

```
1 address;data;wr;en
2 00001510;00004FC0;0;1
```

A blockdiagram of the test-bench including the cache is in figure 7.8. The test-bench will read one line at a time, from the file. Each line holds what is needed to decide if it is a read or write memory access, see listing 7.1 line 2.

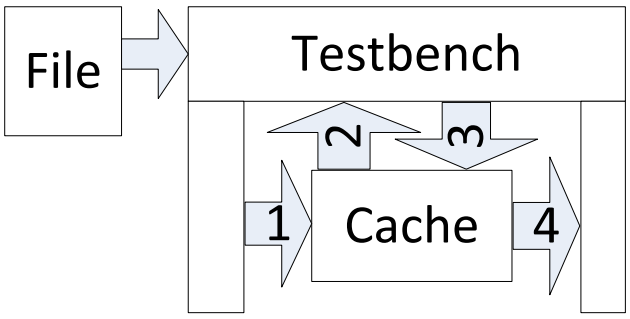


Figure 7.8: VHDL testbench block diagram

This test-bench does not instantiate any memory, since the data needed in all cycles is in the line from the input file, the author chose to let the test-bench act as memory. The numbered arrow is the input and output to and from the cache, the unnumbered is from the input file.

- **Arrow 1:** This is the input needed for a memory access, an address, a write signal, a enable and in the case the memory access is a write a data word. With out having the exact time for when the address generation unit is done, the author and a couple of GN ReSounds IC designer agreed that half a clock cycle, is more then enough.

- **Arrow 2:** If the cache has a miss on a read, it needs to access the memory. The memory address, write signal and enable signal is then used by the test-bench to control if the memory access is correct. In case of a write, the data is forwarded as well.
- **Arrow 3:** Here the data is returned in case of a successful memory read.
- **Arrow 4:** If the memory access at arrow 1 was a read then the data is returned here, either from memory or cache depending on hit or miss.

The memory access handled in the test-bench can be seen in listing 7.2, as shown the process keeps track of number of memory read and write.

Listing 7.2: Memory access in test-bench

```
1  mem_access : process(clk)
2    variable read_cnt : integer := 0;
3    variable write_cnt : integer := 0;
4    begin
5      if (falling_edge(clk)) then
6        if addr_fe = addr_mem_fe and en_mem_fe = '1' and start =
             '1' then
7          if we_mem_fe = '0' then
8            read_cnt      := read_cnt + 1;
9            mem_read_cnt  <= read_cnt;
10           data_mem_fe_o <= data_fe;
11         elsif we_mem_fe = '1' then
12           write_cnt     := write_cnt + 1;
13           mem_write_cnt <= write_cnt;
14         end if;
15       end if;
16     end if;
17   end process;
```

Whit a trace file on 500.000 or more lines. And a system where the end state of the device under test (here the cache), does not tell anything about the correctness of the system, there is a need for the test-bench to keep track of this. For this the author has chosen some of the parameters which the model also keeps track of.

- **Trace size:** Number of inputs.
- **Cache hit:** Number of hit in the cache, the signal hit coming from the cache is used here.

- **Mem read:** Number of memory read.
- **Mem write:** Number of memory write.
- **Wrong data:** Number of times the cache have returned a wrong word.

This makes for an easy comparison between the model and the VHDL. By it self it is useful, as this $Tracesize = Cachehit + Memread + Memwrite$ should be true. And of course there should not be any wrong data.

During development much simpler trace files was used, with patterns for which the author easily could predict the behaviour. These will not be presented in this thesis.

Verifying that the code works like intended

During development small predictable input patterns were used, this was as to make debugging of the VHDL easy, the debugging was done in modelsim.

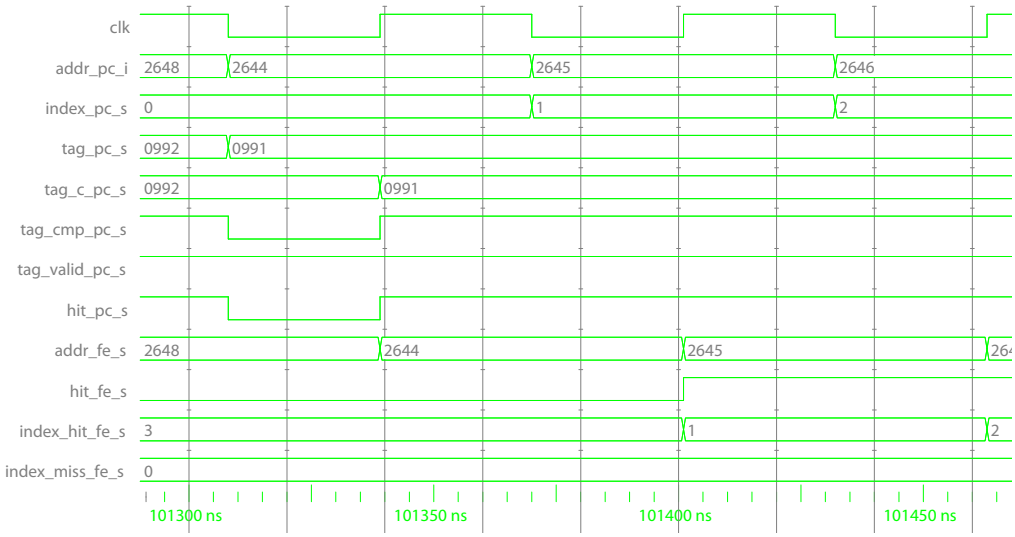


Figure 7.9: Waveform of a miss and a hit

Figure 7.9 shows cache miss followed by a cache hit. Just before time 101310 ns at the falling edge of clk the address is ready. Here the address comes from the test-bench, if it was in the real system it would come from the address generation unit. Address H'2644 has a tag of H'0991 and an index of H'00, at address H'00 in the cache is stored H'0992 which means that this is a cache miss and the index is clocked to $index\_miss\_fe\_s$. At time 101375 ns

the address is H'2645, which is a tag of H'0991 and an index of 1, address 1 in the cache store H'0991 so this is a hit.

As mentioned in last section, the size of the trace file and that verifying though wave is a lengthy task. The test-bench keeps track on some of the key parameters. Listing 7.3 shows a selected part of the output from the model.

Listing 7.3: Selected output from model

```
1 Trace file size: 715816. Unik address: 6744
2 Read:     715730 Write:          86
3
4 4: 1: 1 ....  155857
```

The read in this output relates to the trace file, i.e. without taking into account the cache hit, the number of memory read for a 4 line direct mapped cache is shown in equation 7.1.

$$715730 - 155857 = 559873 \tag{7.1}$$

Listing 7.4: Output from testbench using a 4 line direct mapped cache

```
1 # Reading file
2 # Trace size: 715816
3 # Cache hit: 155857
4 # Mem read: 559873
5 # Mem write: 86
6 # Wrong data: 0
```

The result from a 4 line direct mapped cache when simulated in Modelsim is shown in listing 7.4.

	Modelsim	Model
Trace file	715816	715816
Cache hit	155857	155857
Memory read	559873	559873
Memory write	86	86

Table 7.1: Comparison between model and Modelsim

In table 7.1 the comparison between the simulation of the VHDL in Modelsim and the result from the model is presented. As seen the numbers are

identical, which shows that the VHDL and the model behaves in the same way.

The VHDL was developed in order to analyse the precision of the power estimates from the model. The power numbers for the VHDL was done through power simulation at gate level. As the synthesise flow is integrated in the GN flow, the author got some help.

The VHDL was passed to GN ReSound's back end engineer during the synthesise. After the synthesise the netlist along the sdf[1] file is simulated. In order to compare the number to the ones from the model the same trace file has to be used. From this simulation a VCD[2] dumpfile is created. This file hold the information for each cell in the netlist, which transition it performs at which time.

7.3 Result and sub conclusion

In order to have a starting point to compare the result to, equation 7.2 show the power if only using a RAM and equation 7.3 show what a system with out a cache uses according to the model.

$$P_{RAM} = E_{ma} * f \ = \ 3,464 * 16 \ = \ 55,4\mu W \tag{7.2}$$

The numbers in equation 7.2 and 7.3 are from chapter 6.0.2.

$$P_{System} = \left(\frac{E_{RAM} + E_{ROM}}{Trace\ size}\right) * f \ = \ 31.83\mu W \tag{7.3}$$

$$P_{4L_DM} = \left(\frac{E_{4L_DM}}{Trace\ size}\right) * f \ = \ 2.92\mu W \tag{7.4}$$

7.3.1 Relative comparison

Listing 7.5 show the result from the power simulation. A 4 line direct mapped cache uses $57\mu W$ and equation 7.2 shows that a memory system only consisting of RAM uses $55,4\mu W$. This will be considered in the next section, in this section the number will only be used relative to each other.

[1] Synopsys delay format file
[2] Value Change Dump

Listing 7.5: Numbers from power simulation

```
model            leakage     internal    swcap       total
---------------  ----------  ----------  ----------  ----------
(H) top_4l_2w      1.2 uW      54.5 uW     19.8 uW     75.6 uW
(H) top_4l_dm    700.8 nW      43.7 uW     12.6 uW     57.0 uW
(H) top_8l_dm    969.8 nW      38.9 uW     14.1 uW     54.0 uW
```

In listing 7.5 it can be seen that the static part is a small part of the total power uses, this means the choice not to include it in the model was right. The ratio between the total power use is shown in table 7.10, normalized to the power used by a 4 line direct mapped cache. This is compared to the same number from the model.

	Power simulation	Model
4L_DM	1	1
8L_DM	0,947	1,37
4L_2W	1,33	1,4

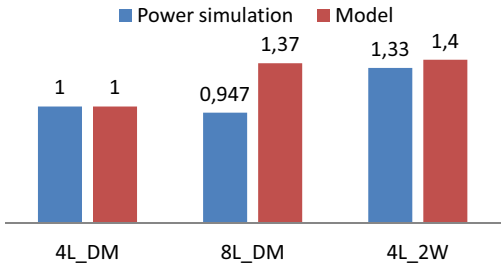


Figure 7.10: Comparison between power simulation and model

As can been seen in the table and graph in figure 7.10, the model and the power simulation does not follow the same trend. In the model an 8 line direct mapped cache uses more power than a 4 line direct, this is shown in figure 7.11.

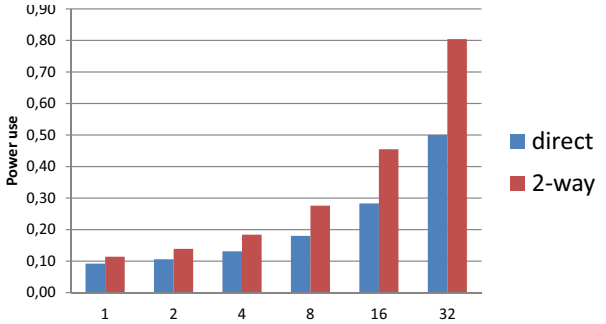


Figure 7.11: Trend for power uses in model

With the limited data from the power simulation, it is hard to conclude anything with certainty. But the data which is available does not follow the same trend, as shown in figure 7.10.

The model use the same activity factor for all caches, this may explain the behavior. In table 7.2 the activity factors from the netlist simulation are shown. Adjusting the power numbers to an activity factor of 5% may help

	Activity factor
4L_DM	0,1181707
8L_DM	0,07682422
4L_2W	0,09580368

Table 7.2: Activity factor from VHDL caches

in explaining what causes the difference to occur.

$$\frac{8L\_DM}{4L\_DM} = \frac{\frac{54}{0{,}1182} * 0,05}{\frac{57}{0{,}1182} * 0,05} = 1,45 \tag{7.5}$$

$$\frac{4L\_2W}{4L\_DM} = \frac{\frac{75}{0{,}096} * 0,05}{\frac{57}{0{,}1182} * 0,05} = 1.62 \tag{7.6}$$

Equation 7.5 and 7.6 show that with same activity factor the power simulation behave the same.

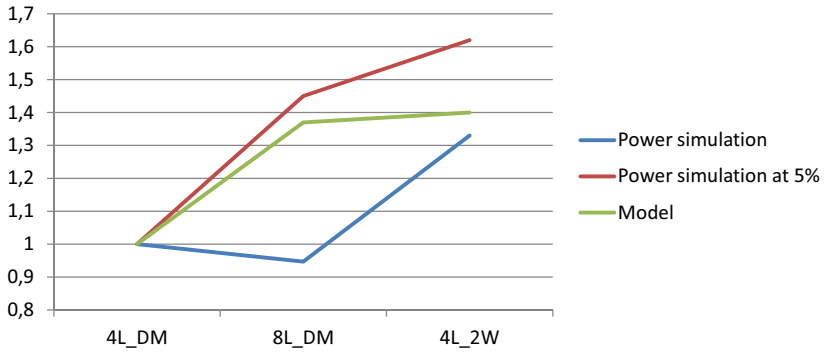


Figure 7.12: Trend for power

Figure 7.12 shows the power from power simulation both with the real activity factor and when converted to a 5% activity factor. The converted graph is a lot closer to the model, which indicates that the activity factor may be what is causing the difference in behaver.

On a side note please note that the difference between an 8 line direct mapped and a 4 line 2-way cache is not the same for the 5% power simulation graph

and the model. This indicates that the cost for having more ways in a cache is too low in the model. Based on the authors feeling, this is because the activity factor in the compare block is higher than in the rest of the cache. The model uses the same activity factor all over, which could lead to the behaviour seen in figure 7.12.

7.3.2 Actually numbers

Shown in table 7.13 are the numbers from the power simulation and from the model/equation.

top_4l_dm	57,0 uW
top_8l_dm	54,0 uW
top_4l_2w	75,6 uW

RAM	55,4 uW
System	31,84 uW
4L_DM	2,92 uW

Figure 7.13: Comparison between power simulation and model/equation

In the power simulation a 4 line direct mapped cache used more power then a RAM did, according to the RAMs data and a simple calculation. And it uses much more then the models 4 line direct mapped cache, nearly 20 times more. In order to clarify why this may be so, it will help to look at the netlist and what components the tool has used. A part of this list is in figure 7.14. A full list for all three caches can be seen in appendix figure C.3 to C.5. Marked with red is some of the components which in a small design like this are questionable.

- **Register:** Thinking of figure C.1 there are very few register, which need to drive more then 1 or 2 input. There are some, like the $index\_hit\_fe$ and $index\_miss\_fe$. This design have 26 X8 and 52 X4

- **CLK-buffers:** Using X40, X24 and X20 clock buffer seam a bit extreme for this design.

- **Latches:** The latches is used for local clock gating, so ones again using 8 X20 seam bit extreme.

In figure 6.1 the data for an AND2 was shown, the different from a X1 to a X12 is $\frac{0,0431}{0,0068} = 6,34$. A factor 6+ when moving from a X1 to a X12, and the synthesis tools choosing components with a high X may explain the extrema power used by these small caches.

Module	afcells	Registers	Buffers	CLK-buffers	Simple	Complex	CLK-gates	Latches
top_4l_dm	1019	274	465	9	47	165	33	16
	'L'	0	14	0	27	122	0	0
	'1'	195	302	0	6	38	0	0
	'2'	1	125	0	6	4	33	0
	'3'	0	5	0	3	0	0	2
	'4'	**52**	7	1	2	1	0	2
	'5'	0	4	0	0	0	0	0
	'6'	0	2	0	0	0	0	1
	'8'	**26**	0	0	3	0	0	1
	'10'	0	4	0	0	0	0	0
	'12'	0	0	0	0	0	0	**2**
	'14'	0	0	0	0	0	0	0
	'16'	0	**1**	0	0	0	0	0
	'18'	0	0	0	0	0	0	0
	'20'	0	**1**	**4**	0	0	0	**8**
	'24'	0	0	**3**	0	0	0	0
	'32'	0	0	0	0	0	0	0
	'40'	0	0	**1**	0	0	0	0

Figure 7.14: Components in a 4 line direct mapped cache

At the moment of writing this, the guy how helped with the power simulation have a open support case with the EDA[3] tool vendor. He has not be able to get the tool to stop using what seam like oversized components.

[3]Electronic design automation

Chapter 8

Calibrating the model and the final result

In this chapter the result from the model build in 5 and the result from the power simulation of the VHDL from chapter 7, was supposed to be used to calibrate the model making it more precise. After the calibration the end result from the model would be presented.

As shown and explained in chapter 7 the result from the power simulation is for now not useable.

8.1 Calibration of the model

As the result from the power simulation was useless there real are nothing to calibrate with. The only thing which can be calibrate is the cost for doing a tag compare, as it turned out it is not high enough. This seam reasonable, an activity factor of 5% is to low.
In figure 8.1 is shown the gate used to do a compare, the input will change each clock cycle. But new tag arrives before saved tag, meaning that the XOR gate can change output as a function of new tag and the saved tag from last clock cycle. When the saved tag from this clock cycle then arrives the output may change again. This behaver will ripple though the OR gates. With a tag of 12 bit and tag being two sequential addresses, it will still result in two toggle on 6 output port.

The result from the power simulation show that the activity factor for the caches varies enough to make the result from a model using a fixed activity factor to inaccurate. There for no calibrating of the model will be done.

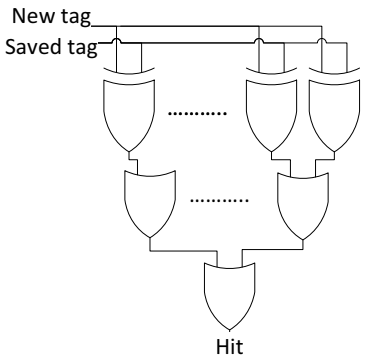


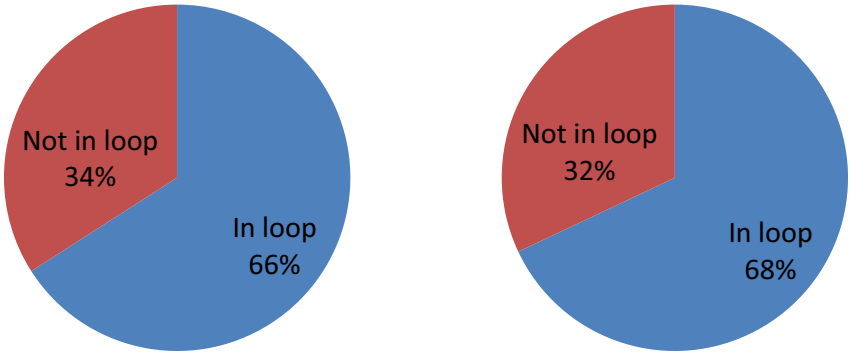
Figure 8.1: Gate to do a compare

8.2 Result

There are still result worth showing. All result presented is from a model which have not been calibrated. This mean that they will show a trend, but that the absolute number probably are not correct.

8.2.1 Stream vs. not stream

In chapter 4 was stated that two different trace file was created, one where the hearing aid received wireless stream audio and one where it did not. The non streaming trace file have not bee used until now.



(a) Streaming (b) Not streaming

Figure 8.2: Instruction divided into loop and not loop

Figure 8.2 show the division of instruction in those in loop and those not in

loop. As can be seen, there are a bit more instruction in loops, in the not streaming trace file. This make sense since the streaming trace file have a lot big interrupts, fetching data from the wireless interface.

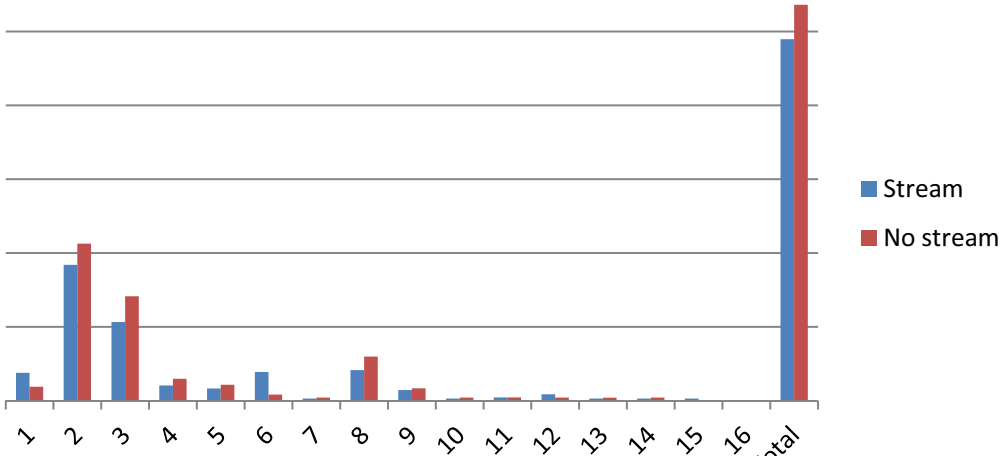


Figure 8.3: Relative amount of loops in different sizes, with and with out streaming

Figure 8.3 show the same trend as figure 8.2. That when the hearing aid handle wireless communication, the relative amount of loops is lower.

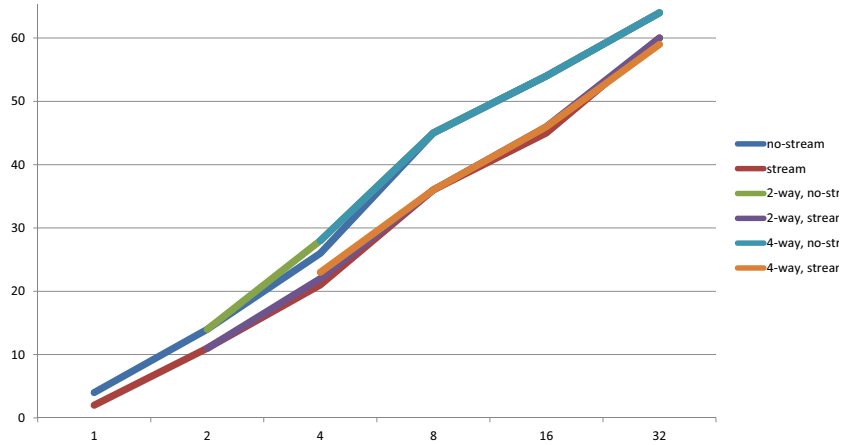


Figure 8.4: Hit rate as function of cache size and streaming being on or off

In figure 8.4 can be seen two thing, first - that the hit rate do not vary much as function a numbers of ways, long as the total cache remain the same. Second - the non streaming trace file result in higher hit rate all over. As

the non streaming trace file contain relative more loops, it is only logic that it should give a higher hit rate.

8.2.2 Presentation of the result

In this section all data is from the streaming trace file.

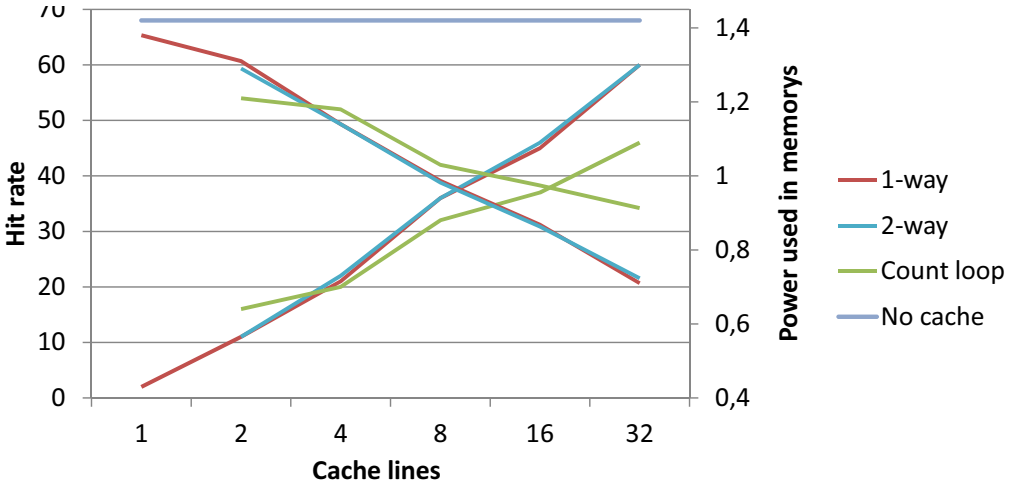


Figure 8.5: Hit rate (rising graphs) and power saved (falling graphs) as function of cache size

Figure 8.5 show both hit rate and the power saved as function of cache size. As in figure 8.4 it is shown that hit rate do not change much as function of ways. Neither do the the power, but the model have not be calibrated, and in chapter 7 was stated the the power cost for doing a compare is to low in the model. This mean that the falling blue line (power used in memory with a 2-way cache) should be higher all over.

Looking at the Counter loop graph, it can be seen that both the hit rate and the power is less steep, as the loop caches only caches loop.

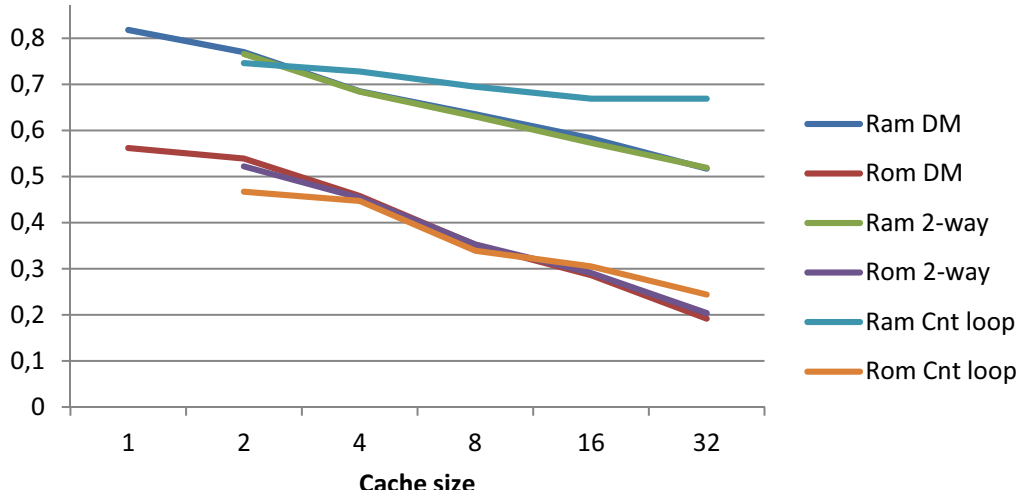


Figure 8.6: RAM and ROM power saved as function of cache size

The main thing to see in figure 8.6 is that the power saved for a loop cache in RAM access is less steep, then the rest as cache sizes grow. This lead to that the the main part of all loops are in ROM.

Figure 8.7, 8.8 and 8.9 show how the power of a system change as the activity factor change.

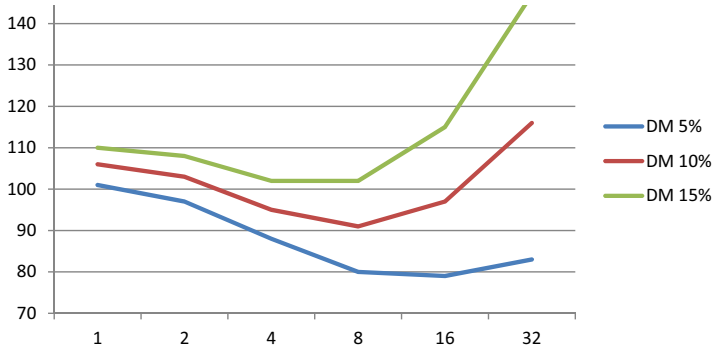


Figure 8.7: Direct mapped cache power as function of size and activity factor, power is in % of system with out cache

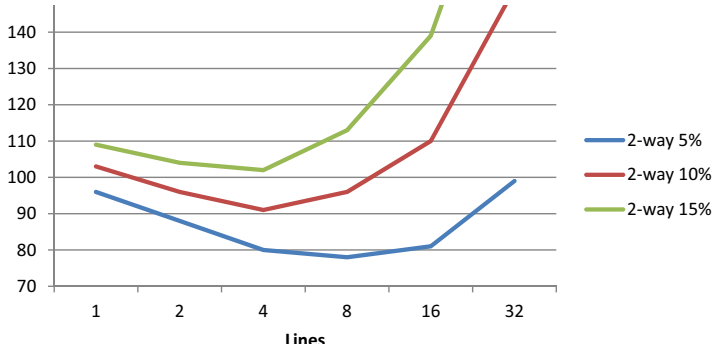


Figure 8.8: 2-way cache power as function of size and activity factor, power is in % of system with out cache

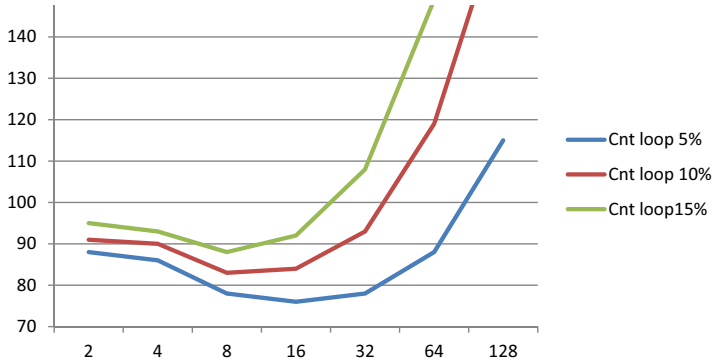


Figure 8.9: Loop cache, with counter reset, power as function of size and activity factor, power is in % of system with out cache

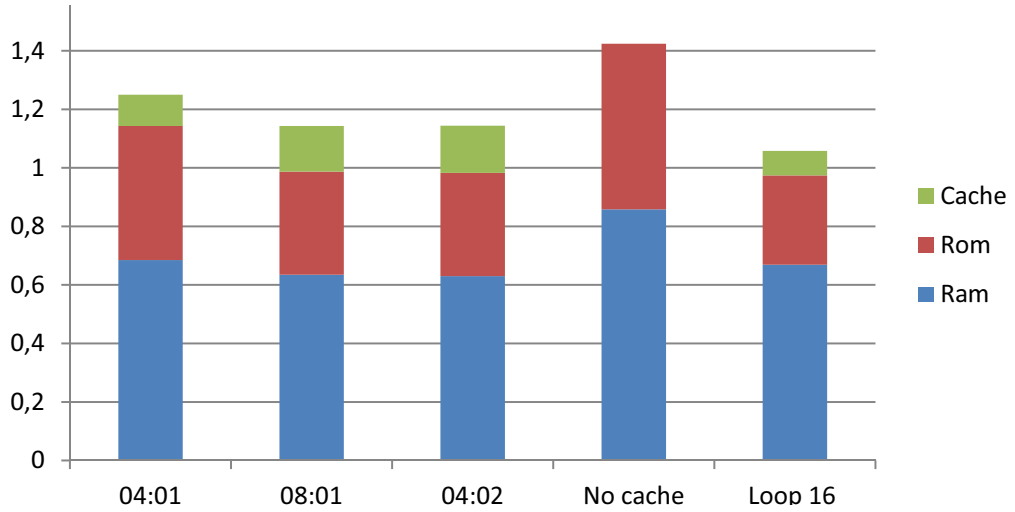


Figure 8.10: Division of power, result from model

In figure 8.10 the power for the three caches chosen for the VHDL implementation and a loop cache is shown. The power is from the model with an activity factor of 5% and no calibration. The green is the caches power and as it can been seen it can more then double before reaching same level as the system with out a cache.

Chapter 9

Future work

This chapter will presented some of the idea the author have, but do not have time to explore. Some of the wisdom in regard to thing which may have worked better if done differently and which way to go if any one wish to continue this work.

The chapter is diveded into two part, one for the model and one for the cache most suitable for GN ReSound. Both part is based on that the power simulation ends up giver a better result.

9.1 Model

9.1.1 Models structure

The model is build taking one memory access at a time. It would have make the task of designing the power cost function less challenging, if the model was designed taking two step. Step referring to the pipeline state and the registers. Down side to this is that the complexity of the model will grow.

9.1.2 Model with no cache power

Taking into account that the activity factor change a lot from cache to cache. It may be a better solution to build a model, which only show what will be saved in power by using a specific cache. A long with this some statistic about the trace file. A designer can use that to make a qualified guess on the cache, the test a few caches vis power simulation.

9.1.3 Model with activity factor

The author started doing this, but the complexity of the model exploded. Then again a model doing activity factor power calculation ect. is more or less the EDA tools used for power simulation. All though only for one design

9.2 Cache for GN ReSound

9.2.1 Data memories

If trace files was created for the data memories, these could be used in the model, just like the ones from the program memory. There by it would be possible to get an idea, regarding power saving using caches on the data memories.

9.2.2 Loop cache in vhdl

More the half the instruction fetch is in a loop, this make it interesting. As the only referents to its power use is the model, it could be good to see what a power simulation would return for a loop cache.

9.2.3 Loop cache change of flow

The version in this thesis uses a address compare scream to validate if there are a change of flow. This being costly in regard to power a better way may be to use information from the AGU. The author have not looked in the code, so the following is a guess. The AGU much know is the address it is generating is a +1 or a jump. Forwarding this knowledge to the loop cache and there is a signal slowing change of flow. When returning the AGU will fetch the return address from the return register, where is was saved when jumping. Signalling to the cache when the AGU is fetching from this register will tell that the cache is on again.

9.2.4 Loop cache taking loop larger then its size

The design chosen in this thesis, discard all loop greater then its size, resulting in missing potential cache hit. It would be interesting to try a design caching only part of a loop, when it is to big to fix inside the cache, and then fetching the rest of the loop from main memory. The upside from this approach is that even with a 8 word cache, there will be cache hit from a 12 word loop.

Down side is the extra logic need to check if the address from the AGU is in the cache or part of what did not fix in the cache.

9.2.5 Loop cache and cache

This part is based on the authors feel, a loop cache like the one suggested in the section above. This cache will be off when the DSP is in halt, at this time a small cache is used. This cache need not be big, a one or two word cache should be sufficient. It is only mend to cache the nops surrounding the one instruction interrupt, example of this was mentioned in chapter 4.

Chapter 10

Conclusion

During the course of this thesis, a software model, for finding which cache (type and size) in a processor memory results in the lowest total use of power, has been presented.

In today's hearing aid, there is a requirement for advanced algorithm, for example perceptual noise reduction and automatic gain control. Along these algorithms, alot of today's hearings aids have a wireless connections and at GN ReSound all algorithms and the wireless control is handled by the processor. This puts pressure on the processor and the memory system.

A background on different caches and other ways of saving power in a memory system has been presented. Following this, an introduction to the current processor and memory system in a GN ReSound hearing aid has been given. At the same time, some of the design rules at GN resound has been stated, as these have influence on the choices made.

There are many studies made in regard to caches. The majority concerns getting as high hit rate as possible at the expense of complexity and power. A minor part of these studies concern power saving in caches, making the cache use less power by using clever ways of reading the RAM used as memory in the cache. An even smaller part of these studies concerns using caches to lower system power and these all use RAM as memories.

Building a software model would not make sense unless an input resembling

the behaviour of the processor systems memory accesses would be available. As this was not the case this thesis work included making such an input.

The majority of work gone into this thesis has, been put in to designing and implementing the model, with the time split nearly equally between the functional part and the power cost function. First - the functional part of the model was presented, and it was shown how the memory system behaviour when having a cache added. The model handles difference cache size and type and the most important discovery made was presented. Second - the idea and implementation of the power cost function was presented and the result from this shown.

In order to determine the precision of the model and to calibrate it, a number of caches were described in VHDL. These cache were chosen based on the results from the model. After synthesization, power simulation was used to find the power these caches use. The same input as the model uses was used here. Due to issues with the tools the actual power numbers from this power simulation was not usable. This was caused by what seemed like a bug in the tool and was reported to the tool vendor. The issue being that the tool for some part of the design used components way too big, causing the power to be higher than what it in reality should be. Doing this, however, resulted in a disturbing discovery, that the activity factor varies a lot more than experted from cache to cache.

The power cost function in the model builds on a common activity factor for all caches, and with the before mentioned observation leaves the cache power part of the model inaccurate. If the power simulation could have resulted in some more accurate numbers, the model could be calibrated and then even with a constant activity factor it would be useable. In this case, the results would not be fully accurate but they would show a trend and with the statistics from the input this would help a designer choose the right cache.

Even with the issue in regard to power simulation, the work done in this thesis can be used. With the result from this thesis and a bit more work, suggestions in chapter 9, it will be possible to determine if a cache can lower the total power consumption. This, however, requires that the synthesis and power simulation works properly.

In this thesis no work has been done with regard to the data memories,

again suggestions in chapter 9.

After spending several months working on this thesis, looking at trace files, running and debugging the model with different settings numerous times, the author feels he can come with a qualified guess, as to the cache giving the lowest power consumption. This is a loop cache like the one suggested in chapter 9.2. This cache is using the signals from the address generation unit to look for change in the flow and it can cache part of a loop bigger than its size. Next to it, a small direct mapped cache should be placed, placed which would only be on when the processor is in halt.

Chapter 11

References

1. The webpage *http : //www.gnresound.com*

2. Bruce Jacob, Spencer W. Ng, David T. Wang, Memory Systems Cache, DRAM, Disk. 2008 Morgan Kaufmann. ISBN: 978-0-12-379751-3

3. John L. Hennessy, David A. Patterson, Computer Architecture A Quantitative Approach 1-4ed. 2007 Morgan Kaufmann. ISBN 978-0-12-370490-0

4. John L. Hennessy, David A. Patterson, Computer Organization and Design: The Hardware/Software Interface 3-4ed. 2007 Morgan Kaufmann. ISBN 978-0-12-374493-7

5. Peter J. Ashenden, The Designers Guide to VHDL 3ed, 2008 Morgan Kaufmann. ISBN 978-0-12-088785-9

6. Jens Sparsø, 2004. Digtal Design and Computer Organization, Technical University of Denmark [online via DTU internal webpage]

7. Alberto Nannarelli, Advanced Digital Design Techniques, Technical University of Denmark [online via DTU internal webpage]

8. Alberto Nannarelli, Flemming Stassen, Design of IC's, Technical University of Denmark [online via DTU internal webpage]

9. The Filter Cache: An Energy Efficient Memory Structure. Johnson Kin, Munish Gupta and William H. Mangione-Smith. The Department of Electrical Engineering, UCLA Electrical Engineering. 1997

10. Tiny Instruction Caches For Low Power Embedded Systems. ANN GORDON-ROSS, SUSAN COTTERELL AND FRANK VAHID. Department of Computer Science and Engineering University of California, Riverside. 2002.

11. Low-Cost Embedded Program Loop Caching - Revisited. Lea Hwang Lee , Bill Moyer*, John Arends*. Advanced Computer Architecture Lab, Department of Electrical Engineering and Computer Science, University of Michigan. 1999.

12. L. H. Lee, B. Moyer, J. Arends, Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops, Proc. Int'l. Symp. on Low Power Electronics and Design, 1999.

13. Algorithm and architecture of a 1V low power hearing instrument DSP. Finn Møller Nikolai Bisgaard, John Melanson. 1999 international symposium on Low power electronics and design. ISBN:1-58113-133-X

A. Appendix to chapter 4

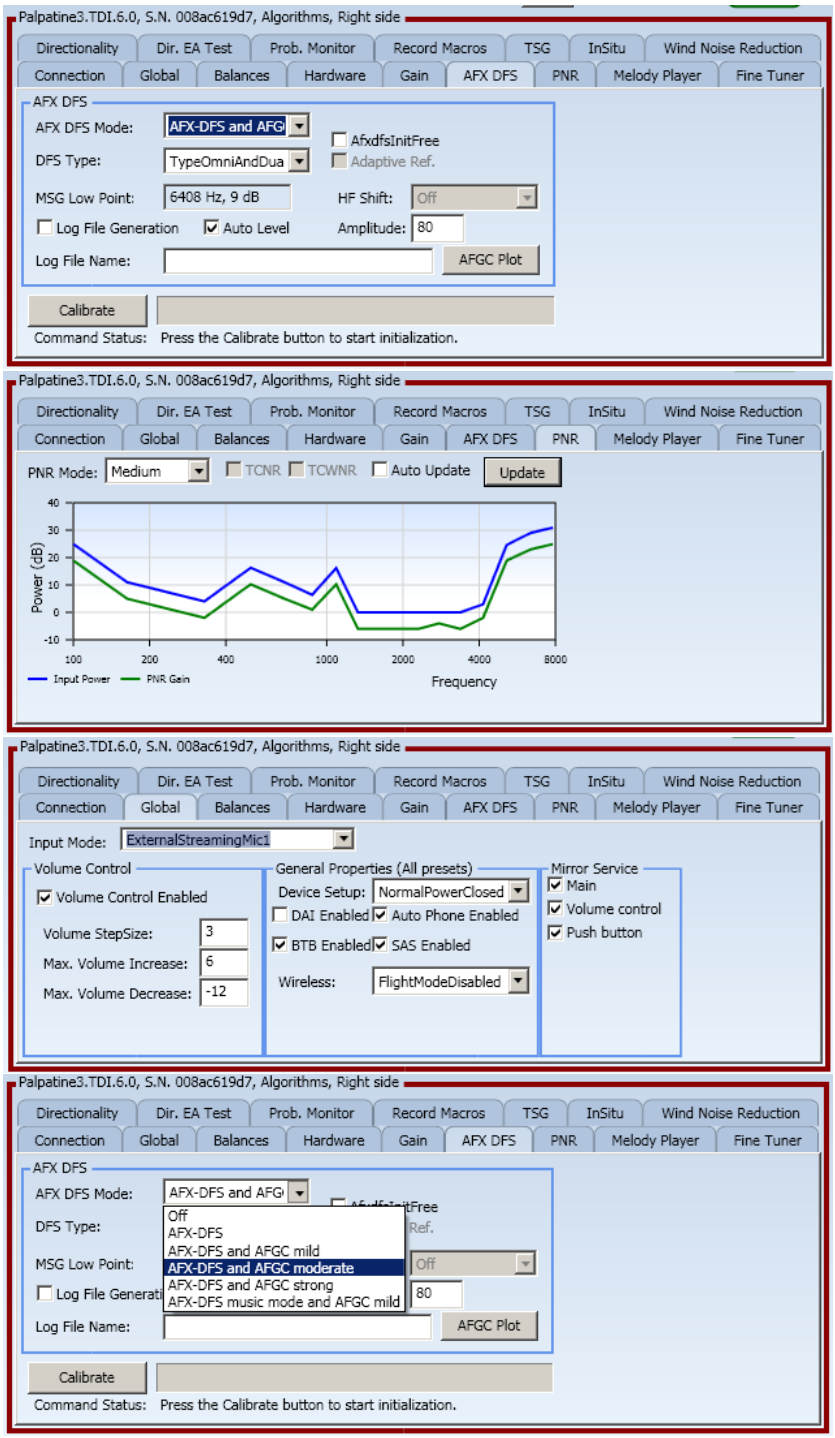


Figure A.1: Set-up of algorithm

B. Appendix to chapter 5

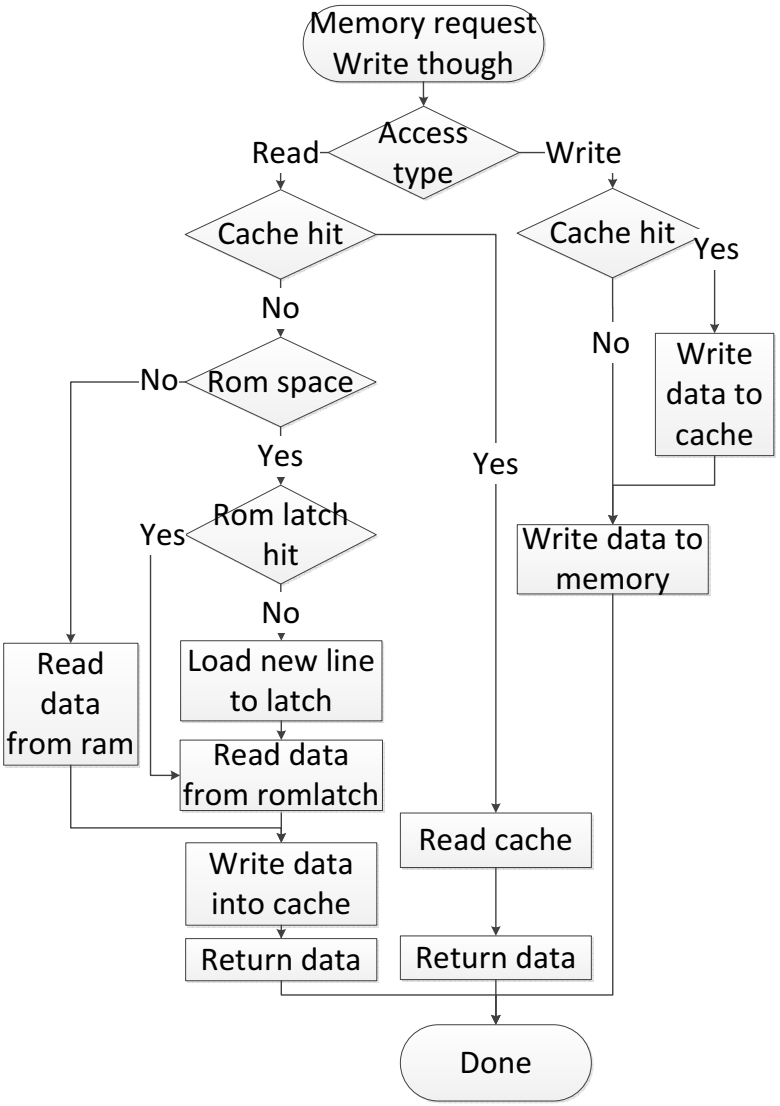


Figure B.1: Flow chart for a write though cache

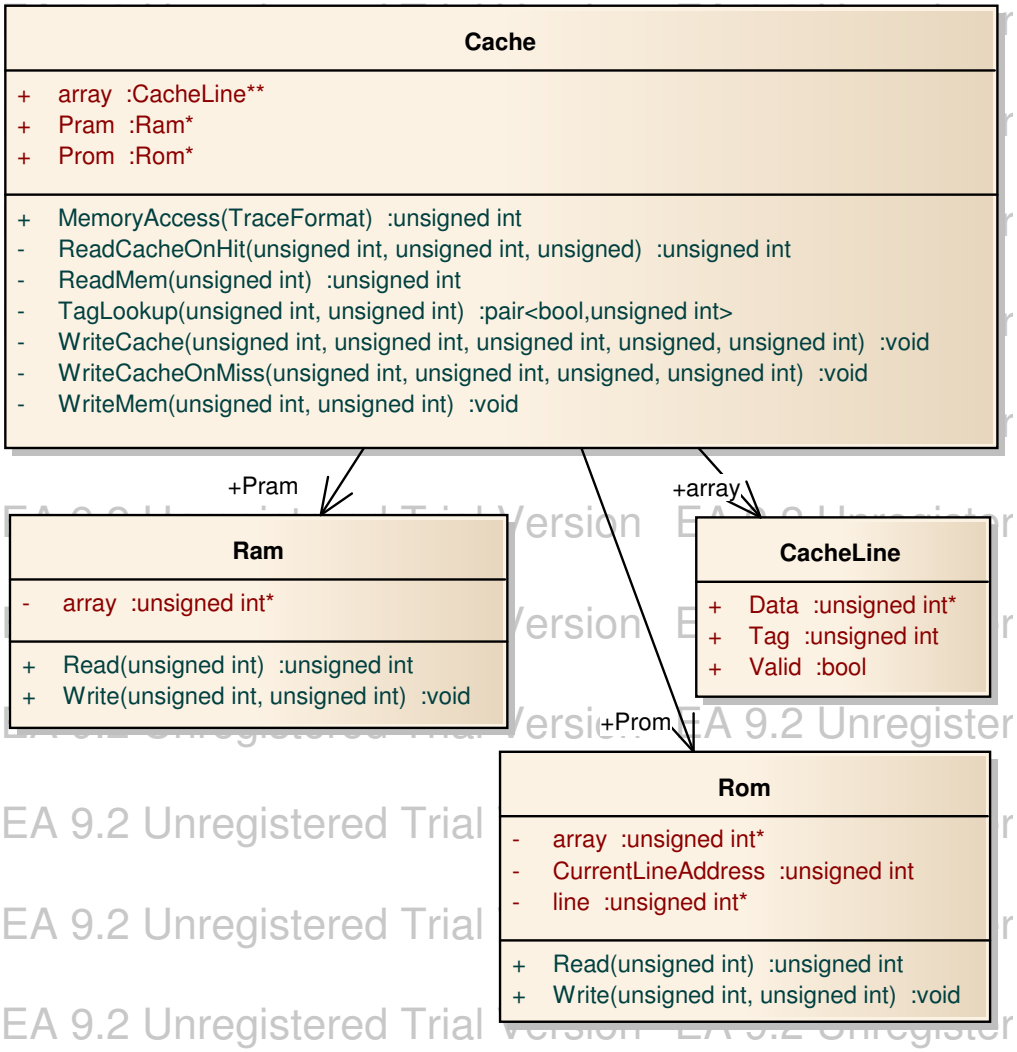


Figure B.2: Simplified class diagram for the class Cache

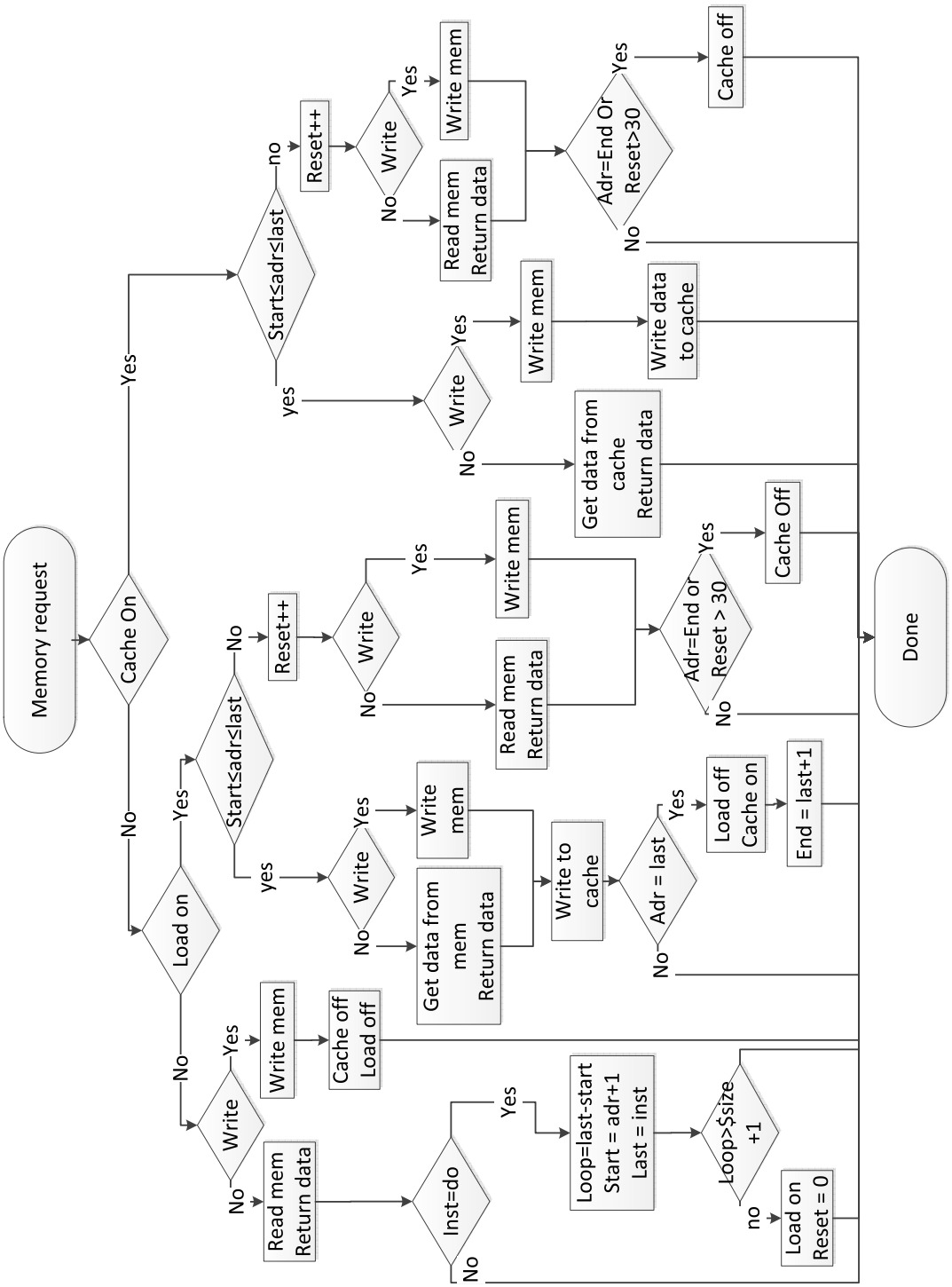


Figure B.3: Flow chart for a loop cache with a count reset

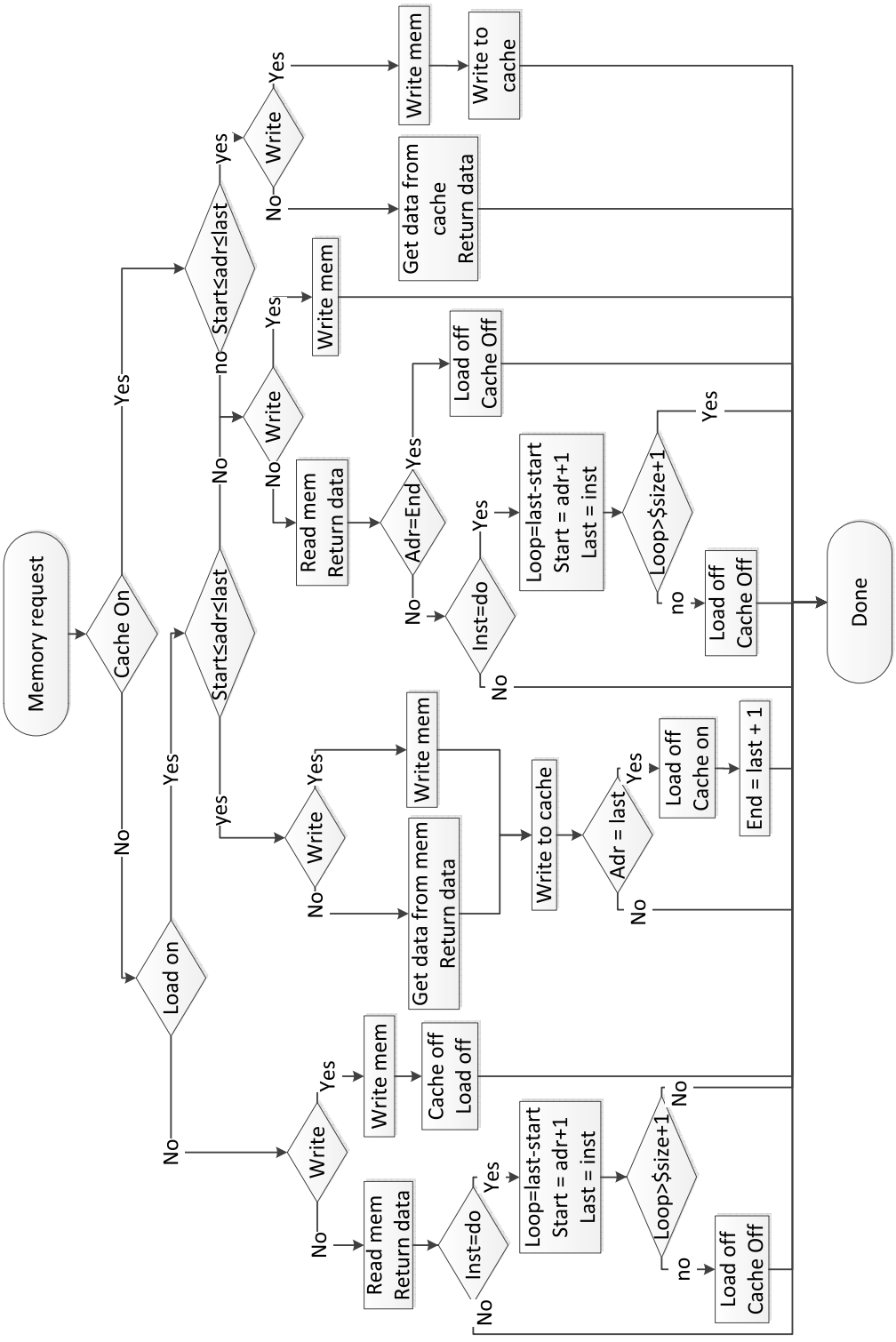


Figure B.4: Flow chart for a loop cache with a instruction dependent reset

C. Appendix to chapter 6

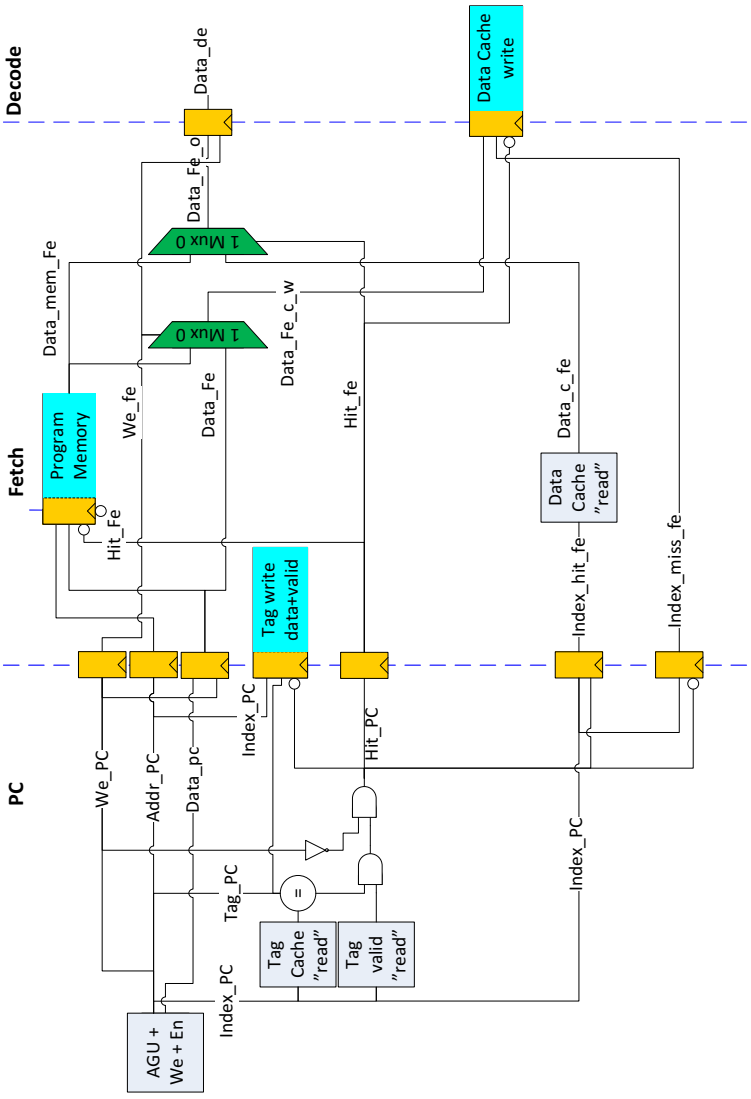


Figure C.1: Block diagram of a direct mapped cache

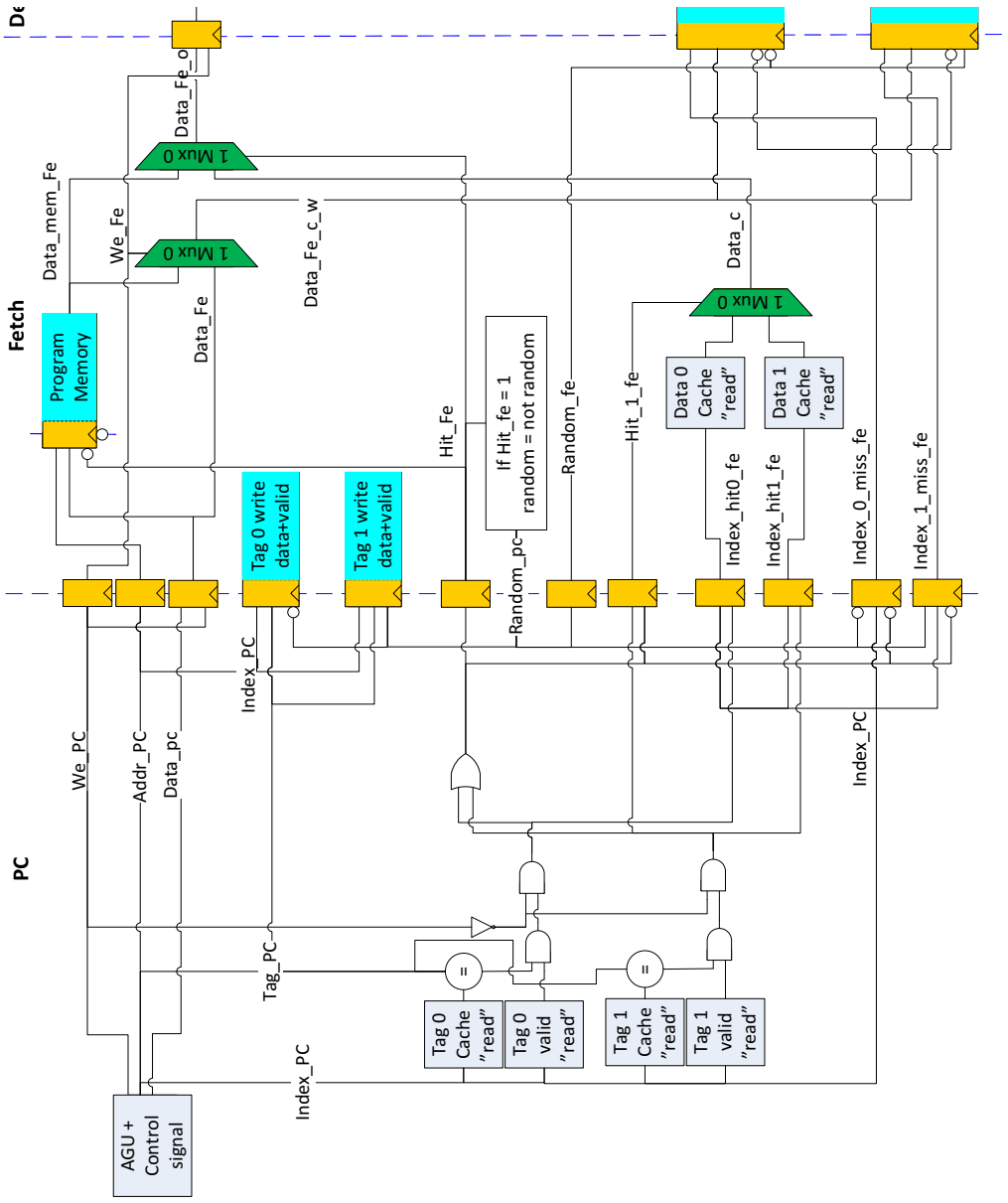


Figure C.2: Block diagram of a 2 way associative cache

Listing C.1: VHDL code for top component in a 4 line direct mapped cache

```
component tag_4l_dm
    generic ( width    : integer := 13 );
    port  ( reset      : in std_logic;
            clk        : in std_logic;
            we         : in std_logic;
            input      : in std_logic_vector(width downto 0);
            valid_out  : out std_logic;
            output     : out std_logic_vector(width downto 0);
            index      : in std_logic_vector(1 downto 0)
          );
  end component;
  component data_4l_dm
    generic ( width    : integer := 31 );
    port  ( reset      : in std_logic;
            clk        : in std_logic;
            we         : in std_logic;
            input      : in std_logic_vector(width downto 0);
            output     : out std_logic_vector(width downto 0);
            index_r    : in std_logic_vector(1 downto 0);
            index_w    : in std_logic_vector(1 downto 0)
          );
  end component;
  begin
    tag_cache_4l_dm : tag_4l_dm generic map ( width => 13 )
    port map  (
              reset      => reset,
              clk        => clk,
              we         => hit_pc_s,
              input      => tag_pc_s,
              valid_out  => tag_valid_pc_s,
              output     => tag_c_pc_s,
              index      => index_pc_s
              );
    data_cache_4l_dm : data_4l_dm generic map ( width => 31 )
    port map  (
              reset      => reset,
              clk        => clk,
              we         => hit_fe_s,
              input      => data_c_w_fe_s,
              output     => data_c_fe_s,
              index_r    => index_hit_fe_s,
              index_w    => index_miss_fe_s
              );
-------------------------------------------------------------------
--PC state
-------------------------------------------------------------------
index_pc_s      <= addr_pc_i(1 downto 0);
```

```
48 tag_pc_s        <= addr_pc_i(15 downto 2);
49 data_pc_s       <= data_pc_i;
50 tag_cmp_pc_s    <= '1' when tag_pc_s = tag_c_pc_s else '0';
51 hit_pc_s        <= tag_cmp_pc_s and tag_valid_pc_s and not
       we_pc_i;
52 miss_pc_s       <= not hit_pc_s;
53 ------------------------------------------------------------
54 --Fetch state
55 ------------------------------------------------------------
56 pc_fe_reg : process(clk,reset)
57 begin
58   if reset = '0' then
59     data_fe_s       <= (others => '0');
60     addr_fe_s       <= (others => '0');
61     index_hit_fe_s  <= (others => '0');
62     index_miss_fe_s <= (others => '0');
63     we_fe_s         <= '0';
64     hit_fe_s        <= '0';
65   elsif rising_edge(clk) then
66     addr_fe_s       <= addr_pc_i;
67     we_fe_s         <= we_pc_i;
68     if we_pc_i = '1' then
69       data_fe_s     <= data_pc_s;
70     end if;
71     hit_fe_s        <= hit_pc_s;
72     if hit_pc_s = '1' then
73       index_hit_fe_s  <= index_pc_s;
74     end if;
75     if hit_pc_s = '0' then
76       index_miss_fe_s <= index_pc_s;
77     end if;
78   end if;
79 end process;
80     addr_mem_fe_o   <= addr_fe_s;
81     data_mem_fe_o   <= data_fe_s;
82     en_mem_fe_o     <= not hit_fe_s;
83     we_mem_fe_o     <= we_fe_s;
84     data_fe_o_s     <= data_mem_fe_i when hit_fe_s = '0' else
          data_c_fe_s;
85     data_c_w_fe_s   <= data_mem_fe_i when we_fe_s = '0' else
          data_fe_s;
86     hit             <= hit_fe_s;
87 ------------------------------------------------------------
88 --de state
89 ------------------------------------------------------------
90 fe_de_reg : process(clk,reset)
91 begin
92   if reset = '0' then
93     data_de_o     <= (others => '0');
```

```
94    elsif rising_edge(clk) then
95      if we_fe_s = '0' then
96        data_de_o   <= data_fe_o_s;
97      end if;
98    end if;
99  end process;
100 end architecture RTL;
```

Listing C.2: VHDL code for data array

```
1  entity data_4l_dm is
2  generic ( width : integer := 31 );
3  port  (   reset       : in std_logic;
4            clk         : in std_logic;
5            we          : in std_logic;
6            input       : in std_logic_vector(width downto 0);
7            output      : out std_logic_vector(width downto 0);
8            index_read  : in std_logic_vector(1 downto 0);
9            index_write : in std_logic_vector(1 downto 0)
10       );
11 end data_4l_dm;
12 architecture RTL of data_4l_dm is
13   type mem is array(0 to 3) of std_logic_vector(31 downto 0);
14   signal ff:mem;
15   signal en_ff  : std_logic_vector(3 downto 0);
16 begin
17   en_ff(0)   <= not index_write(1)  and not index_write(0)  and
         not we;
18   en_ff(1)   <= not index_write(1)  and index_write(0)      and
         not we;
19   en_ff(2)   <= index_write(1)      and not index_write(0)  and
         not we;
20   en_ff(3)   <= index_write(1)      and index_write(0)      and
         not we;
21   process(clk,reset,index_w)
22   begin
23     if reset = '0' then
24       ff <= (others => (others => '0'));
25     elsif rising_edge(clk) then
26       if en_ff(to_integer(unsigned(index_write))) = '1' then
27         ff(to_integer(unsigned(index_write))) <= input;
28       end if;
29     end if;
30   end process;
31   output <=  ff(to_integer(unsigned(index_read)));
32 end architecture RTL;
```

Listing C.3: VHDL code for valid bit in Tag

```
ff_valid : process(clk,reset)
begin
  if reset = '0' then
    valid_reg <= (others => '0');
  elsif (rising_edge(clk)) then
    if we = '0' and en = '1' then
      valid_reg(to_integer(unsigned(index))) <= '1';
    end if;
  end if;
end process;
output <=  valid_reg(to_integer(unsigned(index)));
```

Module	Submodules Le	afcells	Registers	Buffers	CLK-buffers	Simple	Complex	Adders	CLK-gates	Latches	Physicals	Pads	Macros
top_4l_dm	0	1019	274	465	9	47	165	0	33	16	10	0	0
	Drive:	'L'	0	14	0	27	122	0	0	0	0	0	0
	Drive:	'1'	195	302	0	6	38	0	0	0	0	0	0
	Drive:	'2'	1	125	0	6	4	0	33	0	0	0	0
	Drive:	'3'	0	5	0	3	0	0	0	2	0	0	0
	Drive:	'4'	52	7	1	2	1	0	0	2	0	0	0
	Drive:	'5'	0	4	0	0	0	0	0	0	0	0	0
	Drive:	'6'	0	2	0	0	0	0	0	1	0	0	0
	Drive:	'8'	26	0	0	3	0	0	0	1	0	0	0
	Drive:	'10'	0	4	0	0	0	0	0	0	0	0	0
	Drive:	'12'	0	0	0	0	0	0	0	2	0	0	0
	Drive:	'14'	0	0	0	0	0	0	0	0	0	0	0
	Drive:	'16'	0	1	0	0	0	0	0	0	0	0	0
	Drive:	'18'	0	0	0	0	0	0	0	0	0	0	0
	Drive:	'20'	0	1	4	0	0	0	0	8	0	0	0
	Drive:	'24'	0	0	3	0	0	0	0	0	0	0	0
	Drive:	'32'	0	0	0	0	0	0	0	0	0	0	0
	Drive:	'40'	0	0	1	0	0	0	0	0	0	0	0

Figure C.3: Components used for a 4 line direct mapped cache

Module	Submodules Le	afcells	Registers	Buffers	CLK-buffers	Simple	Complex	Adders	CLK-gates	Latches	Physicals	Pads	Macros
top_8l_dm	0	1201	456	373	13	104	191	0	32	23	9	0	0
	Drive:	'L'	0	16	0	73	173	0	0	0	0	0	0
	Drive:	'1'	375	193	0	15	11	0	0	0	0	0	0
	Drive:	'2'	4	136	0	6	6	0	32	1	0	0	0
	Drive:	'3'	0	4	0	1	0	0	0	0	0	0	0
	Drive:	'4'	47	5	0	1	1	0	0	0	0	0	0
	Drive:	'5'	0	5	0	0	0	0	0	0	0	0	0
	Drive:	'6'	0	0	0	4	0	0	0	2	0	0	0
	Drive:	'8'	30	1	1	4	0	0	0	6	0	0	0
	Drive:	'10'	0	9	0	0	0	0	0	0	0	0	0
	Drive:	'12'	0	0	1	0	0	0	0	6	0	0	0
	Drive:	'14'	0	4	0	0	0	0	0	0	0	0	0
	Drive:	'16'	0	0	0	0	0	0	0	2	0	0	0
	Drive:	'18'	0	0	0	0	0	0	0	0	0	0	0
	Drive:	'20'	0	0	1	0	0	0	0	6	0	0	0
	Drive:	'24'	0	0	3	0	0	0	0	0	0	0	0
	Drive:	'32'	0	0	5	0	0	0	0	0	0	0	0
	Drive:	'40'	0	0	2	0	0	0	0	0	0	0	0

Figure C.4: Components used for a 8 line direct mapped cache

Module	Submodules Le	afcells	Registers	Buffers	CLK-buffers	Simple	Complex	Adders	CLK-gates	Latches	Physicals	Pads	Macros
top_4l_2w	0	1532	470	548	14	141	269	0	33	38	19	0	0
	Drive:	'1'	0	24	0	102	198	0	0	0	0	0	0
	Drive:	'1'	422	235	0	10	37	0	0	0	0	0	0
	Drive:	'2'	45	253	0	18	10	0	33	1	0	0	0
	Drive:	'3'	0	7	0	0	0	0	0	2	0	0	0
	Drive:	'4'	3	9	0	2	24	0	0	3	0	0	0
	Drive:	'5'	0	0	0	0	0	0	0	0	0	0	0
	Drive:	'6'	0	1	1	1	0	0	0	1	0	0	0
	Drive:	'8'	0	4	0	2	0	0	0	3	0	0	0
	Drive:	'10'	0	12	0	0	0	0	0	0	0	0	0
	Drive:	'12'	0	0	1	6	0	0	0	5	0	0	0
	Drive:	'14'	0	0	0	0	0	0	0	0	0	0	0
	Drive:	'16'	0	0	0	0	0	0	0	2	0	0	0
	Drive:	'18'	0	0	0	0	0	0	0	0	0	0	0
	Drive:	'20'	0	3	4	0	0	0	0	21	0	0	0
	Drive:	'24'	0	0	4	0	0	0	0	0	0	0	0
	Drive:	'32'	0	0	3	0	0	0	0	0	0	0	0
	Drive:	'40'	0	0	1	0	0	0	0	0	0	0	0

Figure C.5: Components used for a 4 line 2-way cache

Technical University of Denmark
Department of Informatics and Mathematical Modeling
Building 321
DK-2800 Kongens Lyngby
Denmark
Phone +45 45253351
Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk