Accelerating Instruction Set Emulation using Reconfigurable Hardware and Trace Based Optimization

Andreas Erik Hindborg

Kongens Lyngby 2013
IMM-MSc-2013-96

DTU Compute
Technical University of Denmark
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@compute.dtu.dk
www.compute.dtu.dk IMM-MSc-2013-96

Summary

The speed of instruction set emulation is essential when researching new instruction sets or changes to current instruction sets. It is also essential when executing programs for which the originally targeted platform is no longer available.

Software-based instruction set emulators are used extensively in computer architecture development. They allow researchers to evaluate performance implications of changes to the instruction set without going through the effort of implementing the instruction set in silicon. They also allow software developers to develop and debug software for hardware architectures for which the hardware model is not yet finished. However, there is an inherent limit to the amount of instructions that can be emulated in a given time frame for emulators implemented in software.

This report presents a method to implement instruction set emulation by using a reconfigurable hardware accelerator combined with a conventional processor. A significant part of the proposed system is designed and implemented. The system aims to emulate the Motorola 68000 instruction set. The hardware part of the implemented solution is capable of operating at a clock frequency of 237 MHz.

Preface

This thesis was prepared at DTU Compute at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Computer Science and Engineering.

Lyngby, 30. August, 2013

[illegible]

Andreas Erik Hindborg

Acknowledgements

I would like to thank my supervisor Sven Karlsson for pitching the project idea and providing excellent supervision during the execution of the project. I would also like tho thank DTU Compute for providing excellent coffee during execution of the project. This report would not have been what it is if not for the coffee machine at the first floor of building 322 at DTU. At last, but not least, I would like to thank my girlfriend Terese S. Lund for helping me out with English spelling and grammar in the report.

Contents

CHAPTER 1

Introduction

Throughout the history of computing, compatibility between different computer architectures has been an issue. A program that is designed to run on one type of computer may not run on another type of computer. When a computer system user has bought a computer of a certain type, along with a set of programs that is able to run on the computer, the user becomes tied to that specific computer platform. The programs that the user has bought for the computer will possibly not work on another type of computer. If the user wants to upgrade the computer system, the user must buy a computer of the same type in order to keep using the software that was bought with the old computer system.

Computer manufacturers are interested in selling more computers. Offering faster computers might compel users to upgrade their old computer systems to new models. However, the user might not buy a new computer if the new computer obsoletes all previously acquired software. This forces the computer manufacturer to keep supporting old features of previous computer systems, even though it is a complex task that might even hurt performance of the new computer.

At some point, the manufacturer of a certain type of computer may stop producing that type of computer. This may happen for one of many reasons. It may not be profitable to produce the computer in question, or the company may cease to exist for some reason. The user may be required to eventually upgrade his computer system, and lose the ability to run old software.

1.1 Emulation

Emulation technology makes it possible to run software for one type of computer on another type of computer. By using emulation software, the user might be able to run his old software on a new computer. He might also be able to run software written for one type of modern computer on another type of modern computer. However, emulation software usually incurs some overhead. The emulated software will not run as fast under emulation as it would on the original platform. As research in computer architecture tends to find ways to make computers operate faster all the time, this overhead may be tolerable. Software that ran with some speed on an old computer system, may run fast enough under emulation on a modern computer system, due to advancements in computer architecture. Even though the software may be run with an acceptable speed, any advancements that are able to make the emulation process faster are still interesting. A faster computer is always interesting.

The *Instruction Set Architecture*, ISA, of a system defines the programming model that a program must use to run on that system. If a program is to run on a specific computer system, the program must at least[1] target the ISA of that computer system. An emulator that emulates one ISA on another ISA is called an ISA emulator.

Besides making it possible to run software written for a machine that implements one ISA to run on a machine that implements a different ISA, emulation may be used to port software between systems that use the same ISA but runs different operating systems. Such an emulator is called an operating system emulator. Operating system emulators can be implemented by emulating operating system code along with user program code, or by implementing emulated operating system services as emulator functions.

Besides ISA and operating system, other aspects of computer systems such as I/O devices may be emulated. Emulators that provide full ISA emulation, and I/O device emulation are often referred to as *full system emulators.* .

[1]The program may also be required to target specific Application Programming Interfaces provided by the computer platform.

1.2 Simulation

Simulation is a concept closely related to emulation. Emulation is concerned with making a program targeting one type of computer system execute on a different type of computer system. Simulation, on the other hand, is concerned with investigation of the behavior of a running program or the computer system on which it is running. Computer architecture researchers depend extensively on simulation to investigate and validate architectural features of computer systems. Depending on the type of information a simulation is required to produce, the system or program under investigation may be simulated with more or less accuracy. If the target of the simulation is the underlying silicon hardware, the simulation is required to handle a high level of details. If the simulation is concerned only with memory references made by a program, many details can be left out of the simulation. For instance a DRAM timing model is not necessary in this case, because DRAM timing (usually) has no effect on the memory references made by a program.

Emulation and simulation are both related to the use of high level virtual machines to enhance portability of programs across heterogeneous computing environments.

Emulation technology is often used to emulate ISAs that only exist as software implementations. Such ISAs are referred to as *Virtual Machines*, or VMs. A VM architecture describes a computer architecture that includes, but is not limited to, the ISA. VMs allow programs targeting the VM to run on any platform that has a software program that implements the VM. Some VM specifications describe VMs that operate on binary coded instruction streams as known from real machine architectures [LYBB13]. Others specify the virtual machine architecture on the program source level [vR13]. A VM in the latter group is some times referred to as *language VM.*

1.3 Goal

The goal of this project is to emulate the m68k instruction set by using dynamic binary translation from the m68k ISA to a custom ISA developed as part of the project. An execution engine implementing the custom ISA will be implemented in reconfigurable hardware.

1.4 Overview

The rest of the report is organized as follows. Chapter 2 summarizes research related to the project scope. Chapter 3 provides background on some of the topics that the rest of the report builds on. Chapter 4 describes the goal of the project and summarizes the approach used to reach the goal. Chapter 5 to 7 describes the implementation of the hardware based emulator that was developed as part of the project. Chapter 8 describes a simulation framework that was used to develop and test the emulator. Chapter 9 presents results and findings of the project. Chapter 10 gives an overview of project tasks that would be nice to complete, but which could not be completed within the project time frame. Chapter 11 concludes the project.

As a help to the reader, an index is provided at the back of the report.

CHAPTER 2

Related Work

In this section related work in various disciplines related to the project will be summarized. The references made in this section originate in some of the material that was investigated as part of the project process.

2.1 Interpretation

The most basic way to emulate a foreign ISA is by using a piece of software to interpret the instructions of the program to be emulated one by one. A representation of relevant pieces of architected state[1] is kept in memory. The core of a typical emulator is the instruction dispatch loop. The instruction dispatch loop uses the emulated PC to fetch, decode and execute one instruction at a time. The instruction is typically executed by calling a specific function that implements the semantics of the fetched instruction. The called function updates the emulated state and returns to the instruction dispatch loop where the next instruction is fetched, decoded and executed.

[1]Architected state refers to state that is defined by the ISA. This includes the PC and condition code registers, but not internal micro architectural features such as rename registers or reorder buffers.

The performance of simple interpreters are not impressive, but they can be simple to implement

SimpleScalar is an extendable instruction set simulator [BA97]. It is capable of performing simulation if the IA-32[2] ISA and others, with configurable level of detail. It is widely used in computer architecture research because of its openness and flexibility. Both the IA-32 and ARM implementations in SimpleScalar use hand crafted multi-level table based instruction decoders.

Depending on the complexity of the ISA to be decoded, it can be more or less time consuming to write a fast instruction decoder by hand. Krishna et al. has presented a method to automatically generate fast instruction decoders from an instruction description language named *Rosetta* [KA01]. They show that the generated decoders are capable of achieving performance comparable to hand-coded and hand-optimized instruction decoders, and that their IA-32 decoder is faster than the IA-32 decoder that ships with SimpleScalar.

Simics is an interpretive simulator that translates the foreign instruction stream to an intermediate format before interpreting the intermediate format instructions [MS94]. Simics achieves better performance than normal interpreters by translating the interpreted program into a format that is better suited for interpretation. The intermediate format may hold pre-computed information about the instruction that it represents, such as pre-scaled immediate operands. By caching and reusing translations, the Simics system is able to remove some of the decoding overhead. In addition, multiple translations for an instruction can exist at the same time, to represent the effect of an instruction under different conditions. Simics is developed and sold by Wind River Systems in California, USA.

In a 2008 paper Mihoca et al. show how interpreting emulators implemented in C and C++ using suggested implementation techniques can achieve performance similar to the performance achieved by dynamic recompilation engines on the x86 platform [MS08]. By implementing a trace based cache structure to hold decoded instruction traces for the emulated program, they are able to increase the performance of the Bochs x86 simulator by 20%. The authors argue that their interpreters are more portable than dynamic translation solutions because they rely only on high level programming language constructs, rather than target specific assembly constructs. They find that updating the emulated condition code registers is a significant performance bottleneck for interpreting emulators.

In a 2011 paper Tröger et al. evaluate a fast, portable interpreter implemented

[2]The terms IA-32 and x86 are used interchangeably about the Intel 32 bit architecture as featured in the Intel 80386 and subsequent processors.

using C++, based on the principles described by Mihoca et al. [TM11]. The simulator achieves its portability by being written in standard C++. The authors use specification driven instruction decoder generator to generate a decoder that translates the emulated instructions to an immediate representation, much like the representation used in Simics. The intermediate instructions are cached as traces, and the traces are able to execute atomically via special commit/abort constructs. This allows for precise exceptions to be implemented, even when one emulated instruction is translated into more than one internal instruction. The presented simulator employs *lazy flag evaluation*. This technique allows condition code register values to be evaluated only when required. This is achieved by keeping the result and carry information of an instruction around until it can no longer be used to base condition code values on. The authors are able to show that 15-20 different instructions account for more than 60% of dynamically executed instructions in full system workload benchmarks.

2.2 Binary Translation

Dynamic binary translation[3] is an alternative means to implement emulators and simulators. Instead of calling a function based on the instruction to be interpreted, emulated instructions or groups thereof are translated to native host instructions at execution time. The translation process can be costly, and the translated instructions are cached in order to mitigate the translation cost. Usually binary translation systems start out by interpreting the emulated program. Only when a section of code has been executed multiple times is the translation process started.

The translated instructions can be instrumented in order to collect information about the emulated program at run-time. This information can then be used to optimize the translated sequences of host instructions. This process is often referred to as *dynamic recompilation*. Dynamic binary translation and dynamic recompilation are also known as *just-in-time compilation*, or *JIT compilation*.

Shade is one early attempt at implementation of an instruction set emulator using dynamic binary translation [CK94]. Shade achieved good performance in comparison with interpretation based simulation tools available at the time. Using Shade, the amount of information collected during simulation can be increased at the cost of simulation speed by injecting code into the translated code traces.

[3]The terms *dynamic binary translation* and *binary translation* are used interchangeably throughout this report. I shall not refer to the term *static binary translation* unless explicitly stated.

Dynamo is a dynamic optimizer for the PA-RISC platform [BDB99]. It executes on the PA-RISC platform and takes a native PA-RISC binary as input. Dynamo starts by emulating the input program via interpretation. When a given address has been interpreted a number of times, the interpreter will generate a trace starting at the address. The trace is optimized based on information collected by the interpreter, and the optimized result is emitted to an in-memory trace cache. Next time the address of the instruction that caused the interpreter to start the trace is encountered, control is passed to the optimized trace. Because Dynamo requires a native executable as input, strictly speaking it does not perform dynamic translation because no translation is involved. Instead the functionality of Dynamo is referred to as *dynamic optimization.* By leveraging program behavior information collected at run-time, Dynamo is able to increase the performance of statically optimized programs.

Dynamically Architected Instruction Set from Yorktown, or *Daisy*, is an IBM research project that uses binary translation to execute an existing ISA on a VLIW architecture specifically designed as a target for binary translation [EAGS01]. The architecture targeted by Daisy is features 16 execution units. The emulated architecture is the PowerPC RISC ISA. Like many other systems, Daisy starts execution of the emulated program by interpreting the instructions one by one. When a hot region is detected, the region is translated to native code. The translated segments reside in a memory code cache. Instead of linear traces, Daisy uses instruction trees as the unit of translation. Instruction trees are dynamically executed instruction traces that have one entry point, but multiple exit points. As opposed to linear traces, instruction trees may contain both control paths of a conditional branch instruction. Daisy uses this feature in combination with predicated instruction support in the underlying VLIW architecture. Because the VLIW architecture has so many issue slots with predication capability, it may effectively execute down multiple control flow paths simultaneously.

Transmeta Crusoe was a commercially available binary translation system much like Dynamo[K$^+$00, DGB$^+$03]. The Crusoe chip was a VLIW architecture capable of executing x86 programs via dynamic binary translation and optimization. Key points of the Crusoe system are the ability to aggressively reorder and optimize the translated instructions. Because the IA-32 architecture uses precise exceptions, the dynamic optimizer must take care not to optimize the translations in such a way that the exception behavior of the program is altered. To allow reordering and aggressive optimization of translated traces, the Crusoe chip includes hardware to support commit/rollback semantics of executed instructions. This includes shadowed versions of registers and special memory aliasing detection hardware. The first time an exception occurs when the Crusoe chip is executing a heavily optimized code sequence, the sequence is aborted and the effects of the sequence are canceled. The instruction sequence is then interpreted one instruction at a time. If the exception does not occur under this form of execution, then the exception was a result of the aggressive optimization. The dynamic translator will try to compile the trace using more conservative optimization passes. If the exception does occur under interpretation, it should also be reflected in the emulated program.

2.3 Trace Processors

Trace Processors are a class of processor architectures that use *trace caching*, rather than conventional caching, as part of the micro architecture.

Rotenberg et al. introduce the trace cache in a 1999 paper [RBS99] as a means to achieve a higher fetch bandwidth for superscalar processor architectures. The authors use a branch address and a sequence of branch outcomes to specify the address of a trace in the trace cache. A next trace predictor is used to select the next trace from the trace cache based on branch outcome predictions. In case of a miss in the trace cache, the regular instruction cache is queried. If a predicted trace turns out to be wrong, the trace is repaired with the correct instructions. The trace cache is able to increase SPEC95 performance on a simulated superscalar architecture by 15% to 35% over a regular block based instruction fetch mechanism.

Black et al. present a block based trace cache that assembles multi-block traces by storing a trace as a series of pointers to cached basic blocks [BRS99]. The design is able to utilize storage space more effectively than a conventional trace cache. This results in an increased fetch bandwidth that in turn is able to increase the IPC of a 16-wide superscalar architecture by 7%.

Patel et al. evaluate partial trace matching and inactive issuing of blocks [PFP99]. They show that the ability to fetch a trace from the trace cache that only partially matches the predicted trace can increase performance by 14% over a configuration that requires a perfect match. The ability to speculatively issue blocks to the execution engine, even though they were not predicted, can increase performance by 17% over their baseline.

Patel et al. have shown that converting branch instructions to assertions in traces allow for construction of long atomic traces [PTBC00]. Long atomic traces in the trace cache are good candidates for on-line optimization. The proposed assertions cancel the effect of an entire trace if their conditions do not hold. This effectively converts a trace to an atomic entity. The branches of a trace are only converted to assertions if branch prediction accuracy for the trace is good. Using this technique the authors demonstrate that it is possible to build atomic traces that average over 100 instructions and has a 97% completion probability and that the atomic traces constitute 80% of the dynamically executed instruction stream. These results are achieved using the SPEC2000 benchmark suite.

The trace cache is an excellent target for dynamic optimization, and consequently many proposals that use the trace buffer of trace processors as target for dynamic optimization have been published.

Chou et al. and Fahs et al. both propose the use of a small co-processor that runs optimization algorithms on instructions in the trace construction buffer [CS00, FBC$^+$01b]. Because these systems operate in the retirement stage of the processor, they are not subject to the same critical timing constraints as the rest of the processor. Friendly et al. and Jacobson et al. present a similar solution that uses the fill unit to perform the optimization [FPP98, JS99].

Fahs et al. have published an evaluation of the performance potential of such instruction path coprocessors [FMS$^+$04a]. They find that atomic traces are much better subject for dynamic optimization than traces that commit incrementally.

Slechta et al. have published an investigation of the effects of dynamic optimization of micro operations in the trace cache of a simulated x86 architecture [SCF$^+$03]. They find that they are able to reduce the number of micro-operations by 21% by using a combination of common subexpression elimination, value assertion, store forwarding and redundant load elimination on the cached traces.

2.4 Exceptions and Speculation in (Dynamic) Optimization

One key problem that arises when performing dynamic optimization on program traces, either in a hardware trace buffer or as part of a recompilation/translation software framework, is that the optimized code must behave identically to the original code. This is especially challenging when precise exceptions[4] are

[4]Precise exceptions are exceptions that are reported on well defined instruction boundaries. This is opposed to imprecise exceptions that may occur anywhere in a region of code if the exception is triggered. When a precise exception is thrown, an exception handler may expect architected state to reflect the state that was active when the instruction that caused the exception was executed.

thrown in an optimized region that includes reordered or speculatively scheduled instructions.

Work by Mahlke et al. presents a concept called *Sentinel Scheduling* [MCH+92]. Sentinel Scheduling allows a static compiler to preserve precise exception behavior when using speculative scheduling. The method divides potentially excepting instructions, PEIs, into a computational part and an excepting part. The excepting part of a PEI can be combined with an instruction that uses the result of the PEI. This instruction is the sentinel for the PEI. The sentinel must remain in the basic block of the PEI originated from. If the speculated PEI causes an exception, it is not raised until the sentinel for the PEI is executed, and thus it does not alter exception behavior of the program.

Nystrom et al. present a software controllable hardware mechanism for dynamic optimization software [NBMH01]. The method, named *Precise Speculation*, use a shadowed register file and special commit instructions and implicit abort instructions. The system effectively allows the optimizer to control the commit points by inserting the special instructions in the optimized code sections. This allows the optimizer to attempt aggressive optimization and reordering of code segments, while still being able to support precise exceptions. If a speculated instruction would raise an exception, a flag in the destination register of the instruction is set. When the next branch instruction or PEI is executed the exception flag is checked. If it is set, the speculative register file is discarded and control is transferred to the original code. In the original code, the exception will occur again, but at the correct location.

A compiler based approach that allows aggressive optimization while preserving exception behavior without requiring hardware support is presented by Gschwind et al. [GA01]. The method relies on the capability of the exception handler to call special repair code when a PEI causes an interrupt in a situation where the processor state is different from what it would have been if no optimization was done. The compiler generates repair code, which is placed in the translation cache. The repair code is called by the exception handler before exception processing is started, in order to recreate the architected state as it would have been, had the exception occurred at the original location.

Chapter 3

Background

This section provides some background information on subjects of which a certain level of knowledge is required in order to understand the rest of the report.

3.1 The Motorola 68000

The Motorola 68000 is an ISA[Mot92] and a series of micro controller devices[Fre93] developed and manufactured by Motorola (now Freescale Semiconductor) from 1979 and on. The Motorola 68000, abbreviated m68k, was used in a wide range of popular computing devices, from enterprise systems to personal computers. The use of the m68k CPU in the Amiga, Macintosh and the Atari ST has made m68k the production of m68k emulators interesting, both commercially (in the past) and community wise (in the present).

The m68k ISA is a Complex Instruction Set Computing ISA, or CISC ISA. A CISC ISA is an ISA that allows complex operations to be encoded in a single instruction, as opposed to a Reduced Instruction Set Computing ISA, or RISC ISA, which allows only simple instructions to be encoded. CISC instructions can typically access multiple operands in memory using advanced addressing modes, and perform advanced operations on the operands, which would require multiple instructions to execute in a RISC ISA. The m68k ISA supports an abundance of addressing modes, the most complex of which allows for double indirect memory addressing. This means that an operand for an instruction can be accessed via a double pointer dereferencing.

The instructions of the m68k ISA use two operand addressing. Under two operand addressing, instructions take a source and a destination operand as input. A typical instruction loads the values of both the source and the destination, and stores the result of the operation on these two values at the destination operand address.

The basic instructions in the m68k ISA is divided into a set of data movement instructions, a set of integer arithmetic instructions, a set of logical instructions and a set of program control instructions. In addition to these, the m68k ISA defines a set of bit manipulation instructions, a set of bit field instructions, a set of BCD instructions, a set of floating point instructions and a class of system, cache, multiprocessor and memory management instructions.

The m68k ISA defines 16 general purpose integer registers, divided into 8 data registers and 8 address registers. The data registers are D0 to D7 and the address registers are A0 to A7. A7 is used as a dedicated stack pointer. The address registers may be used as base addresses for calculating the address of memory resident operands. In addition to the 16 integer registers, the ISA defines a program counter register (PC) and a condition code register (CCR). Many instructions set bits in the condition code register, based on the result of the instructions. The bits in the CCR are:

- Extend bit (X) has special purposes depending on the executed instruction.
- Negative bit (N) is set if the result of a signed operation is negative.
- Zero bit (Z) is set if the result of an operation is zero.
- Overflow bit (V) is set of an arithmetic overflow is detected.
- Carry (C) is set if a carry out or borrow out is generated by an arithmetic operation.

Instructions of the m68k ISA may operate on three different operand sizes, although not all instructions support all operand sizes. The sizes are byte, for 8 bit operation, word, for 16 bit operation and long, for 32 bit operation. When an instruction writes a register in byte or word mode, the most significant bits of the register are untouched.

3.1.1 Addressing Modes

The m68k ISA supports a number of complex addressing modes. Key aspects of the addressing modes are discussed here. For full specifications, please refer to the m68k manual.

Operand addresses can reference either immediate data, memory locations, or one of two register classes. References to operands that reside in memory are referred to as *indirect addressing.* Register references are referred to as *direct addressing.* Memory addresses for indirect addressing are build from address register values, the program counter, and immediate values.

Address Register Indirect Mode is the most simple indirect addressing mode. Let $M(a)$ be the function that returns the value of memory at address a. The following equation describes the addressing mode:

$$V = M(A_n) \tag{3.1}$$

where A_n is an address register.

Using *Address Register Indirect with Displacement Mode* it is possible to add a 16 bit 2's complement immediate to an address register value, in order to form a memory address. The following equation describes the addressing mode:

$$V = M\left(A_n + d_{16}\right) \tag{3.2}$$

where V is the resolved value for the operand, A_n is the address register used as a base and d_{16} is the immediate displacement.

The *Address Register Indirect with Index (8-Bit Displacement) mode* further allows the use of a second register (the index register) multiplied by a scale factor, but only allows for an 8 bit displacement:

$$V = M\left(A_n + d_8 + X_n \cdot s\right) \tag{3.3}$$

Here X_n is the index register which may be an address register or a data register, and s is a scale factor. The scale factor, which is encoded in the instruction, can be either 1, 2, 4 or 8.

The *Address Register Indirect with Index (Base Displacement) Mode* allows for a full 16 bit or 32 bit displacement:

$$V = M\left(A_n + d + X_n \cdot s\right) \tag{3.4}$$

The most complex addressing mode, *Memory Indirect Preindexed Mode* allows for double indirection as show below:

$$V = M\left(M\left(A_n + d + X_n \cdot s\right) + od\right) \tag{3.5}$$

Here a second immediate displacement, *od*, is allowed. Similarly, *Memory Indirect Postindexed Mode* allows for double indirection but with the indexing register added after the first memory reference.

$$V = M\left(M\left(A_n + d\right) + X_n \cdot s + od\right) \tag{3.6}$$

Some of the address register indirect addressing modes allow the PC to be used as a base register instead of an address register. These modes are listed below.

Program Counter with Displacement:

$$V = M\left(PC + d_{16}\right) \tag{3.7}$$

Program Counter Indirect with Index (8-Bit Displacement):

$$V = M\left(PC + d_8 + X_n \cdot s\right) \tag{3.8}$$

Program Counter Indirect with Index (Base Displacement):

$$V = M\left(PC + d + X_n \cdot s\right) \tag{3.9}$$

Program Counter Memory Indirect Preindexed:

$$V = M\left(M\left(PC + d + X_n \cdot s\right) + od\right) \tag{3.10}$$

Program Counter Memory Indirect Postindexed:

$$V = M\left(M\left(PC + d\right) + X_n \cdot s + od\right) \tag{3.11}$$

The m68k ISA supports auto increment and decrement when using simple indirect addressing. The auto decrement mode automatically decrements an address register used in indirect addressing mode before its value is used to construct the memory address. Similarly the auto increment mode increments the value of the address register used in addressing, but after the values is used to construct the memory address. The updated register values are automatically written back to the register file. The value that is added/subtracted to/from a register in auto increment/decrement mode depends on the size field of the operation. Byte operations increment by ±1, word operations by ±2 and long word operations by ±4.

Different instructions have different limitations on what addressing modes are supported.

3.1.2 An Example

In order to get a feel of the m68k ISA, a short C program, a translation by GCC to m68k instructions, and disassembly of the resulting binary are presented. The program in listing 3.1 defines the location of three different integer arrays, a, b and c. In the loop at lines 8, 9 and 10, the first 256 elements of a and b are added and stored in c. At the end, a function is called to print a message.

Listing 3.1: A small C program.

```
 1  #include <stdio.h>
 2
 3  void main() {
 4    int *a = (int*)0x00001000;
 5    int *b = (int*)0x00002000;
 6    int *c = (int*)0x00003000;
 7
 8    for(int i=0; i<0x100; i++) {
 9      c[i] = a[i] + b[i];
10    }
11
12    printf("The␣end\n");
13
14  }
```

The post compilation assembly version of the program is presented in listing 3.2. The first line of the assembly program (line 2) pushes register A2 on the stack. This is achieved by a move from A2 to the stack pointer (A7). The stack pointer is addressed using a pre decrement addressing mode. The stack grows from the top of the address space towards the bottom, so this instruction achieves a move to stack and an update of the stack pointer in one instruction. Because the operation has size long (it has a `.l` postfix), the stack pointer is decremented by 4, which matches a 32 bit word in memory. A2 is restored on line 15 by before the main function returns in line 16. Note how the stack pointer is addressed using post increment mode, to handle the stack pointer update inline. These stack operations illustrate how the advanced addressing modes can be used to achieve more complex operations in one instruction.

Line 3, 4 and 5 initialize the registers used to reference the arrays a, b and c. The instructions move an immediate word into registers A0, A1 and A2.

Lines 7 to 11 of the assembly program implement the loop body. Line 7 moves an element of the array `a` into register D0 while simultaneously advancing the pointer for array `a` using post increment addressing mode. Line 8 loads an element of array `b`, increments the pointer for array `b` and adds the loaded value to register D0. Line 9 writes the result of the addition back to the `c` array in memory while advancing the `c` pointer by 4. Line 10 compares the pointer for array `a` to the address of the last element of array `a`. This operation sets the condition code register based on the result. Note that the comparison is a word operation that only considers the lower 16 bits of register A0. Line 11 transfers control back to line 7 if the result of the comparison at line 10 was not zero.

Lines 12 to 14 handle the function call to `printf`. The instruction on line 12 pushes an address on the stack and decrements the stack pointer using the `pea` instruction. For this instruction, the pushed operand is always a long, and the stack pointer is always decremented by 4. The `jsr` instruction at line 13 decrements the stack pointer, pushes the PC on the stack and transfers control to the address given as the operand. In this case the operand is the address of the `puts` function. Line 14 takes care of popping the argument from the stack that was pushed in line 12.

Listing 3.2: The C program translated to m68k instructions by GCC.

```
 1  main:
 2          move.l %a2,-(%sp)
 3          move.w #12288,%a2
 4          move.w #8192,%a1
 5          move.w #4096,%a0
 6  .L3:
 7          move.l (%a0)+,%d0
 8          add.l (%a1)+,%d0
 9          move.l %d0,(%a2)+
10          cmp.w #5120,%a0
11          jne .L3
12          pea .LC0
13          jsr puts
14          addq.l #4,%sp
15          move.l (%sp)+,%a2
16          rts
17  .LC0:
18          .string "The end"
```

A m68k instruction consists of one 16 bit base word and up to 10 extension words, 16 bits each. The base word encodes the instruction and the number of extension words. The extension words are used for immediate data and additional description of advanced addressing modes.

Listing 3.3 shows the disassembly of the compiled C program. The listing shows three columns. The leftmost column is a memory address. The middle column is the content of the memory at the address specified to the left. The rightmost column is the data decoded as m68k instructions. The leftmost and middle column use hexadecimal numbers. The first line shows that the main function has been placed at address 144 (hexadecimal) in memory. Line 2 shows that the instruction that pushes A2 on the stack is encoded as a single word. The instructions that load the addresses of the three arrays at lines 3, 4 and 5 are encoded using a single extension word. The extension word is used to hold immediate operands for the loads. Since the loads have word size, a single extension word is required. If the operation had been of size long, two extension words would have been required. The `pea` instruction at line 11 uses two extension words to encode a long immediate operand.

Listing 3.3: The compiled object file, disassembled by objdump.

```
1  00000144 <main>:
2       144:      2f0a            movel %a2,%sp@-
3       146:      347c 3000       moveaw #12288,%a2
4       14a:      327c 2000       moveaw #8192,%a1
5       14e:      307c 1000       moveaw #4096,%a0
6       152:      2018            movel %a0@+,%d0
7       154:      d099            addl %a1@+,%d0
8       156:      24c0            movel %d0,%a2@+
9       158:      b0fc 1400       cmpaw #5120,%a0
10      15c:      66f4            bnes 152 <main+0xe>
11      15e:      4879 0000 2c40  pea 2c40 <__EH_FRAME_BEGIN__+0x4>
12      164:      4eb9 0000 0330  jsr 330 <puts>
13      16a:      588f            addql #4,%sp
14      16c:      245f            moveal %sp@+,%a2
15      16e:      4e75            rts
```

3.2 Computer Architecture

This section introduces some concepts that are used in computer architecture.

3.2.1 Caching

Caching refers to the concept of storing data that is accessed often in a location where it can be accessed with a lower latency than it can when stored in

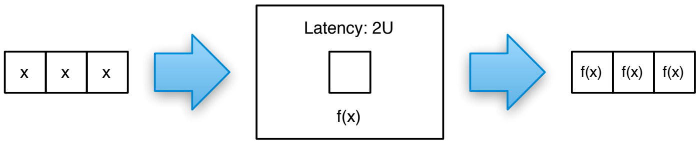

Figure 3.1: Calculating the function $f(x)$ for a stream of inputs x. Each computation takes 2 time units.

the original location [HP12]. Caching is widely used on all levels of computer systems because of the inherent trade-off in storage size and price versus speed, power consumption and price. Large storage buffers typically have a high latency and a limited bandwidth, while the most low latency storage devices are small and expensive. By moving frequently used data from big and slow storage devices to fast but small storage devices, performance can be increased. Whenever a caching scheme is applied, a replacement policy must be defined. The replacement policy defines what action to take when the small and fast storage device is full. One such replacement policy is the *Least Recently Used*, or LRU, policy. This policy dictates that when an item should be placed in the cache but the cache is full, then the item that has been accessed least recently of all replacement candidates in the cache should be removed from the cache.

3.2.2 Pipelining

Pipelining is a technique that allows for performing multiple long latency operations in parallel [PH08]. Figure 3.1 shows a conceptual sketch of a logic circuit that process a stream of input elements. For each input element the logic circuit applies the function $f(x)$, and the computation takes 2 time units. Thus the throughput of the circuit is 1 sample per 2 time units.

Imagine that the logic that calculates $f(x)$ can be separated into two separate circuits, so that the first circuit calculates $f_1(x)$ and the second circuit calculates $f_2(x)$ and that $f(x) = f_2(f_1(x))$. This situation is depicted in figure 3.2. If each of the two new circuit blocks has a lower latency than the original circuit block, then performance of the system is increased. The latency for processing a single sample might be increased, because the total latency of the two circuits may be greater than the latency of the original circuit. Let the latency of the original circuit be l_o, the latency of the $f_1(x)$ circuit be l_1, the latency of the $f_2(x)$ circuit be l_2 and let $l_s = \max(l_1, l_2)$. Then the throughput of the system is given by 1 sample per l_s time unit where $l_s < l_o$. For the depicted system the throughput of the system is 2 sample per 2U time period, which is a 100% improvement

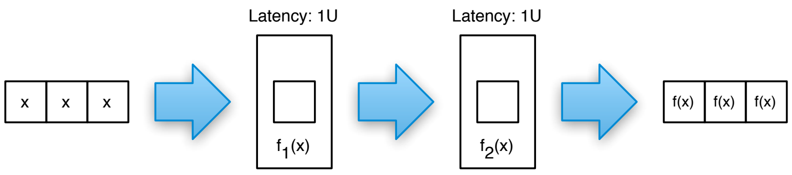

Figure 3.2: The calculation logic has been split into two parts with half the latency each.

over the original system.

Pipelining is extensively used in modern computer architecture design. A good example is the MIPS 5-stage pipeline that includes fetch, decode, execute, memory and write back stages [PH08]. The fetch stage reads an instruction from the instruction cache. The decode stage generates control signals for subsequent stages and reads operands from the register file. The execute stage performs some arithmetic operations. The memory stage access the system memory such as caches backed by DRAM. The write back stage writes the register files.

3.2.3 Branch Prediction

Branch Prediction, or BP, is the process of predicting the outcome of a conditional branch instruction. *Branch Resolution* is the process of verifying that the predicted outcome of a branch is correct. In the m68k ISA, the outcome of a conditional branch is decided by the state of the CCR. The CCR is potentially updated by the instruction that precedes the branch instruction in program order. If the processor implementation is pipelined, which almost all processor implementations are today, the branch cannot be resolved before the preceding instruction has moved past the stage in the pipeline where the CCR is updated. This leaves a potential gap where the processor does not know what instruction to execute next. If the processor is allowed to guess or predict an outcome of the branch, it might be able to spend the waiting time on something useful. However, if the processor makes a wrong guess, then it must be able to undo the effects of the instructions that were issued down the wrong path. Instructions that follow a branch that was predicted or guessed are said to be issued speculatively.

There are many ways to predict the outcome of a branch. Some methods are based on previous outcomes of the branch (local history) while others are based on the history of a number of previously executed branches (global history). The most effective schemes use a combination of local and global history [McF93].

Predicting the outcome of a branch is no good if the branch is predicted to be taken, and the target address cannot be computed in a timely manner. *Branch Target Buffering*, or BTB, is the concept of remembering (or caching) the target of a branch or jump instruction. Normally the target of a branch or jump instruction is calculated somewhere down the pipeline. When BTB is in use, the result of the calculation is saved in a local memory element. Next time the same branch instruction is executed, the target is immediately known early in the pipeline. This can be used in combination with BP to quickly start fetching instructions from the predicted target. Like with BP, the execution of instructions from cached targets might be speculative, and the processor must be able to revert changes done by the speculative instructions.

3.2.4 Store Buffering

A *Store Buffer* is a buffer structure that is used to buffer store operations in a processor [POV03, PH08]. In an in-order pipeline, a store buffer may be used for two purposes. Firstly, it may mask the latency of the cache by buffering a store operation and thus allow execution to continue without waiting for the store to completely retire. Secondly, it allows for store-to-load forwarding [POV03]. With store-to-load forwarding, store operations do not have to reach the cache before a load from the same address can be issued. In a pipeline where loads are issued in the first part of pipeline and stores are executed in the last part of the pipeline, a store that appears logically earlier in program order than a load, may reach the data cache after the load. If the address of the load and store coincide, the value of the store must be forwarded to the load. Otherwise the load may retrieve the wrong value from the cache.

3.2.5 Very Long Instruction Word Processors

Very Long Instruction Word Processors, or VLIW processors, are processors that execute instructions that contain multiple operations [PH08]. A VLIW instruction may for instance have three arithmetic operation slots, a branch slot and a memory slot. This allows the compiler to find *Instruction Level Parallelism*, ILP, in the instruction stream, and encode the ILP directly in the instruction stream. Finding ILP in an instruction stream amounts to finding instructions in a sequential instruction stream that is able to execute simultaneously. Two instructions are able to execute simultaneously if there is no data dependence between the two instructions. One of the advantages of VLIW processors is that much if the complexity associated with extracting ILP in an instruction stream is moved from the processor at the hardware level to the compiler at the software level.

3.3 Field Programmable Gate Arrays

A *Field Programmable Gate Array*, or FPGA, a programmable device that allows digital logic circuits to be implemented by programming connections between a number of digital gates. The gates are implemented as small look-up-table (LUT) structures that are essentially small SRAM memory cells [BV00]. A LUT with four inputs and one output can implement any logical function of four inputs. The LUTs are wired together in a programmable interconnect. In this way multiple four input LUTs can be combined to produce functions of higher numbers of inputs.

The LUTs are usually combined with registers and multiplexers to form *Configurable Logical Blocks*, or CLBs. With the registers in the CLBs, it is possible to create pipelined logical circuits.

FPGAs may also include special dedicated logic to perform certain tasks. Dedicated resources, also sometimes known as hardened resources, are macro cells that fulfill specific purposes. They are either implemented as single ASIC macro cells, or by combining multiple less specialized ASIC macro cells. The advantage of such hardened features, as opposed to the same features implemented in regular FPGA fabric, are that they are able to operate on a higher clock frequency, they consume less power, and they use less space. For instance, caches, branch target buffers, and some times register files, are often best implemented using dedicated *block RAM* memory primitives. Many modern FPGAs also include transceivers for high speed serial communication and DSP blocks for implementation of high performance DSP applications [Alt13].

A single FPGA devices model is usually delivered in different speed grades. The speed grade of a device determines how fast the device may operate. FPGA manufacturers test the FPGA chips as they come off the production line. Chips that are able to operate at higher frequencies are given a lower speed grade and sold at a higher price. The variations in the frequency capability of the chips is a result of imperfections in the production method.

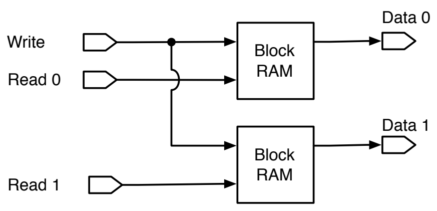

Figure 3.3: Duplicating ram blocks to create more read ports.

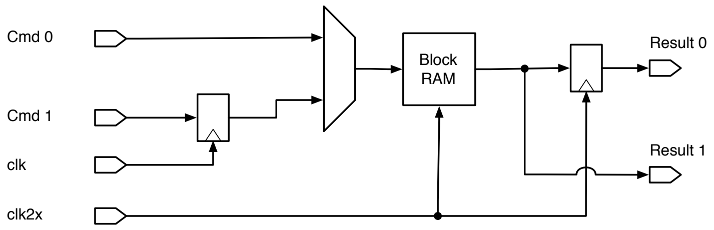

Figure 3.4: Creating more logical write ports by running the block ram macro cell at double frequency.

3.3.1 Dedicated Memory Resources

Block RAM primitives available in FPGA devices typically have no more than two read/write ports[1] [Xil12, Alt13]. While it is possible to create more read or write ports by combining multiple block RAMs in clever ways, the resulting circuit always operates at a lower frequency and/or uses more resources [LS10]. Depending on the available resources and the performance goal of the application, it may or may not be acceptable to apply these combination techniques.

It is possible to create structures with more read ports by creating multiple copies of a block ram. This is achieved by sending write commands to multiple blocks simultaneously as depicted in figure 3.3. This is an acceptable procedure as long as extra memory resources can be spared for the purpose.

It is more difficult to create more write ports. By running the block ram macro cell on a clock twice as fast as the base clock, two write commands can be executed in each base clock cycle. Figure 3.4 depicts the concept, which is referred to as *double pumping.*

[1]It was not possible to locate an FPGA vendor that includes SRAM blocks with more than two write ports on FPGA devices.

However, this may limit the base clock frequency. The base clock frequency can be no faster than half frequency of the fast clock signal. If long paths already limit the base frequency to half that of the maximum allowable fast clock frequency, there is no problem. But if this is not the case, then clock speed is sacrificed for the extra write ports. Implementing a double pumped ram structure in a FPGA can be difficult, because the synthesis tools have to be set up properly to correctly verify timing constraints for the signal paths that cross clock domains.

A second way of producing multiple write ports is the *Live Value Table*, or LVT, technique [LS10]. With LVT, n banks of replicated block RAMs with 1 write port and m read ports are combined to produce a solution with n write ports and m read ports. A write to the cluster of replicated banks only write one of the banks, and thus only one of the banks contain the written value. By using a small table to remember which bank that holds the most recent value for a given address, it is possible to select the correct read result when reading that address.

3.4 Peephole Optimization

A peephole optimizer is an optimization algorithm that works on a linear stream of instructions [TC11]. The optimizer performs pattern based substitution in a sliding window over the instruction stream. If an instruction sequence that matches a known pattern is found, it is substituted by another pattern that uses values from the original match. When all possible substitutions have been performed, the sliding window is advanced. The purpose of using a sliding window is to limit the pattern matching and substitution to a small area, instead of searching the entire linear instruction sequence at once.

Peephole optimization can for instance be used to remove some classes of redundant copy operations, or to lower an intermediate format instruction stream to target specific instructions.

Chapter 4

System Design

This section presents the goal of the project and gives a high level overview of the project components. The section will introduce the concept of foreign ISA emulation backed by FPGA acceleration.

4.1 Concept

The core goal of the project is to emulate the legacy m68k ISA [Mot92]. The emulation is to be achieved by using binary translation from m68k ISA to an internal FPGA friendly VLIW ISA. An execution engine supporting the VLIW ISA is to be implemented as an in-order statically scheduled pipeline in FPGA fabric.

The execution engine is in its own right a fully capable CPU with separate instruction and data caches that interface a local bus system. However, to simplify the design and ease the bootstrapping process, the execution engine is operated as slave co-processor controlled by a generic host CPU via a local bus system. By delegating some tasks to a generic host CPU, the system can be operational before the FPGA execution engine is stable. If the entire functionality of the system was to be offered by the FPGA execution engine alone, the functionality of the FPGA pipeline would have to be quite complete before practical benchmarks could be executed.

One of the major challenges in the project is to design and implement the execution engine in such a way that it will achieve adequate performance when synthesized for FPGA fabric. Designing hardware models that achieve good performance when synthesized for FPGA fabric can be a challenge. The nature of the FPGA fabric makes some constructs efficient while other constructs become inefficient. A large part of the work associated with the project was spent on FPGA implementation.

4.2 Design Overview

The proposed emulation system is depicted in figure 4.1. It consists of an execution engine implemented in FPGA fabric, a host processor and a shared Dynamic RAM, DRAM. The host system is a conventional commercially available system featuring a common CPU. The FPGA may be connected to the host CPU via PCI Express or some other local bus. The DRAM may be connected to the FPGA or to the host CPU memory controller, but for best performance, the FPGA should have a low latency path to the DRAM.

The FPGA resident execution engine is capable of executing m68k instructions by dynamically translating the instructions from m68k ISA to the internal ISA. It can also execute programs specified using the internal ISA directly, bypassing the m68k translation stage.

The FPGA execution engine is connected to main memory via separate instruction and data caches. A *Hot Spot Detector*, HSD, is monitoring the branches executed by the execution engine. When the HSD finds a set of branches that qualifies as a hot spot, it dumps the addresses of these branches to main memory and notifies the host CPU.

The host CPU acts as the orchestrating entity. A high level sequence chart depicting the actions of the host CPU is given in figure 4.2. The host CPU is responsible for booting and initializing the system. It initially loads a m68k program into the shared DRAM. Then it uses the control interface to instruct the FPGA execution engine to start executing the program. Finally the host CPU waits for the FPGA execution engine to detect a hot region in the program.

When the HSD detects a hot program region, it notifies the host CPU, for instance via interrupt. The host CPU extracts the hot region and builds a

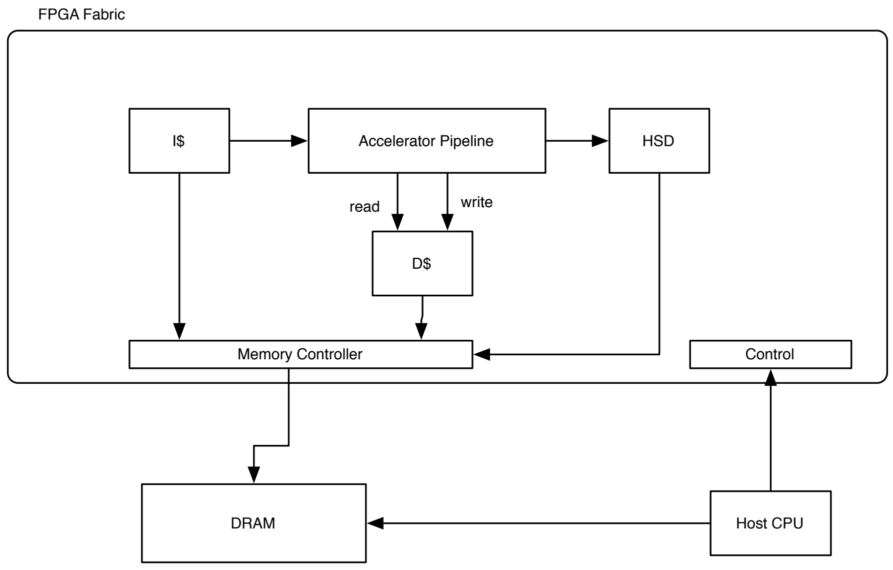

Figure 4.1: Conceptual overview of the accelerator system.

CFG of the code in the region. Then it tries to optimize the hot region. The host CPU may use profiling information collected by the HSD to drive the optimization process. When the region is optimized it is written back to memory. Original code is never overwritten. The host CPU notifies the execution engine of the existence of the optimized code and the possible entry points. When the accelerator executes a branch at one of the possible entry points, it jumps to the optimized code instead of the original code. The execution engine executes the optimized code until it reaches a branch that exits the optimized region. At this point, the execution engine will again translate m68k instructions on the fly, until it encounters a branch to a translated region.

Figure 4.3 depicts a control flow graph in which the HSD has determined that the path consisting of blocks 2-3-4-2 is hot. The optimizer has emitted optimized code for these blocks. The dashed edges represent entry and exit points to and from the optimized region. Because the hardware based dynamic translator emits poorly optimized code, the most basic compilation steps could bring performance enhancements to the hot regions.

Many different optimization techniques may be used when generating code for the hot region. Techniques such as code straightening and loop unrolling could be utilized. The optimizer may even emit code that executed speculative, if

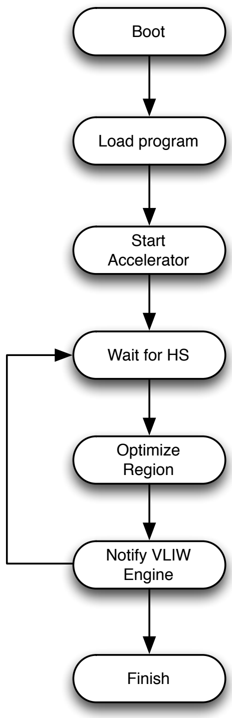

Figure 4.2: Host CPU actions.

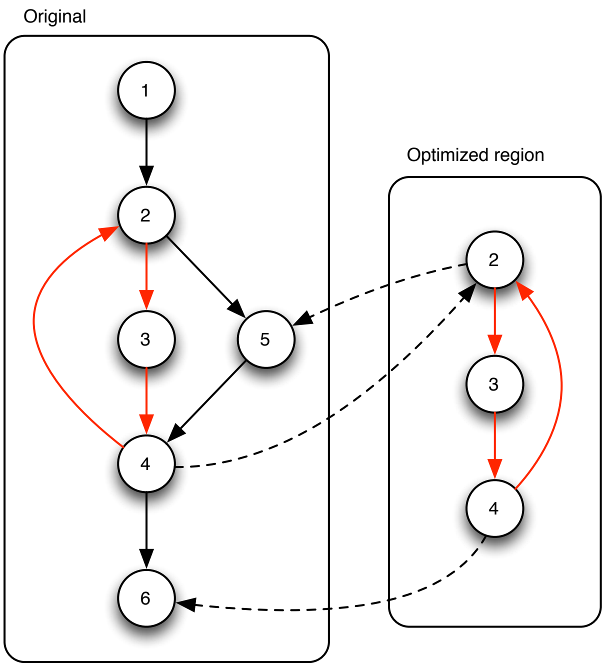

Figure 4.3: Optimizing a hot path.

the execution engine implements proper transaction semantics via commit/roll-back primitives. Alternatively cleanup code can be inserted to undo effects of speculative code.

4.3 Target Platform

For evaluation purposes the Xilinx Zynq 7020 platform is targeted as implementation platform. The Zynq family of devices contains contain two ARM Cortex A9 cores tightly coupled to on-chip FPGA fabric. The ARM cores and the FPGA fabric share an on-chip DRAM controller [Xil13b]. Targeting such a platform, the entire set of system components can be implemented on a single chip.

The Zynq platform is chosen over a PCIe equipped FPGA attached to the PCIe bus of a PC, because the author has prior experience with the Zynq platform. The author believes it is easier to set up communication between the ARM cores

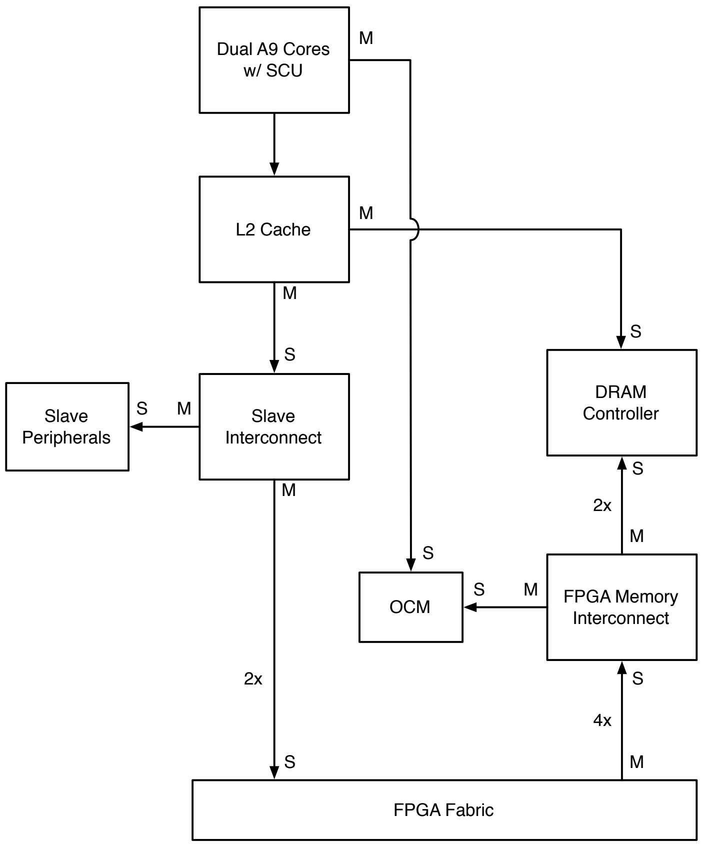

Figure 4.4: The Zynq 7020 on-chip interconnect. (M) indicates a master while (S) indicates a Slave.

of the Zynq device and the FPGA fabric in the Zynq device than it is to set up communication between the CPU of a PC and a FPGA attached to the PCIe bus.

Relevant parts of the Zynq 7020 on-chip interconnect is depicted in figure 4.4. The chip uses the AXI 3 bus standard for most on-chip busses [ARM11]. The FPGA fabric has 4 high performance 64 bit AXI master ports facing the shared DRAM controller. These ports go through the FPGA Memory Interconnect which provides routing towards the DRAM controller or a fast on-chip SRAM. The FPGA Memory Interconnect has 2 64 bit AXI master ports towards the DRAM Controller.

The ARM cores have a single 64 bit AXI master port connected directly to the DRAM controller from the L2 cache controller. It also has access to two 64 bit general purpose AXI slave ports in the FPGA fabric.

The control interface of the execution engine is attached to one of the general purpose AXI slave interfaces of the FPGA fabric. The instruction cache, data cache, and hot spot detector are given an AXI master port each for accessing system DRAM. These three components could share a single master port, but that would require arbitration to be implemented in the FPGA fabric. The system might as well utilize the arbitration features provided in the hardened logic part of the chip [Xil13c].

Chapter 5

Execution Engine

This section presents the design of the FPGA resident execution engine.

5.1 Initial Design

As opposed to some earlier work in binary translation targeting custom execution engines [K+00, EAGS01], the chosen translation target matches the m68k ISA relatively close. This decision was taken in order to make the translation process easier. Translating from the m68k CISC ISA to an internal VLIW RISC architecture unarguably entails more work than translating to an ISA that more closely matches the source ISA. The internal VLIW ISA is not as complex as the m68k CISC ISA, but it is capable of performing many of the complex addressing modes of the m68k CISC ISA.

Implementing an execution engine can be a very time consuming task. In order to increase the chance of arriving at a complete implementation within the project time frame, simplicity and ease of implementation has been key goals in design and implementation of the execution engine.

5.1.1 Design Considerations

One key design choice made in the early design phase is that register files and caches must be limited to two read/write ports. This choice was made in order to (1) keep the required implementation resources at a reasonable level, (2) to cut down implementation time, and (3) allow for the highest possible operating frequency. As mentioned in section 3.3.1, multi-ported memory structures increase implementation complexity, require additional resources in the design and may limit clock frequency.

One consequence of this decision is that there is a limitation on the number of instructions targeting the same register file that can execute in parallel. At most one write to the register file can be executed in each cycle.

As mentioned in section 3.1, the m68k ISA uses two register classes, address registers and data registers. Most operations can use any register class as a source, while address operations use address registers for destination and data operations use data registers for destination.

In order to take advantage of this feature of the m68k ISA, two separate block RAM primitives are used to implement two register banks, one for each register class. Using this scheme, it is possible to write both a data register and an address register in each cycle.

Because of the limited number of register file write ports, some addressing modes cannot be executed as a single instruction. All addressing modes that require multiple memory accesses cannot execute as a single instruction, because this would require two cache accesses in a single cycle. For the same reason, instructions that collect both the source operand and the destination operand from memory must execute as two instructions.

Auto increment/decrement addressing modes may also require two cycles. If either two registers are implicitly updated, or if one register is implicitly updated and the explicit destination of the operation goes to the same register bank as the implicit update.

Table 5.1 shows a statistical analysis of a simulator trace of an m68k application[1]. The simulator produced a trace of all executed instructions. The table shows summations over the instructions in the trace. The column labeled *Instruction* give the name of the instruction for which the row holds statistics. The column labeled *Count* shows the number of times the instruction appear

[1]The data that is the base of this analysis was provided by Sven Karlsson.

Instruction	Count	Percentage	PRef Count	PRef Count %
move	12098436	40.14	640237	5.29
Bcc	4946851	16.41	0	0.00
add	3725176	12.36	0	0.00
cmp	3307080	10.97	0	0.00
and	1514948	5.03	0	0.00
lea	683988	2.27	0	0.00
sub	621173	2.06	0	0.00
tst	551148	1.83	0	0.00
DBcc	503853	1.67	0	0.00
rts	432016	1.43	0	0.00
mvem	336232	1.12	336232	100.00
Total $>$ 1%	**28720901**	**95.28**	**976469**	**0.03**

Table 5.1: Statistics for the a trace of the CuBase program. PRef refers to problematic addressing modes. The total number of executed instructions is over $30 \cdot 10^6$.

in the trace. The column labeled *Percentage* shows the fraction of the total instruction execution count that the instruction accounts for. The column labeled *PRef Count* shows the number of times the instruction appear in the trace with an addressing mode that cannot be handled by the VLIW engine in a single cycle. The column labeled *PRef Count %* gives PRref Count divided by Count for the row. The analysis shows that 11 instructions account for 95.28% of the dynamic instruction stream. Of these 95.28%, 0.03% use addressing modes that require more than one instruction to execute on the internal VLIW engine. Note that only the most significant instructions was analyzed. The addressing mode of the remaining 4.72% was not analyzed. In addition, the analyzer was not able to decode 1.6% of the instructions, as listed in the table. This leaves 5.88% of the instructions out of the analysis, which is a potential error margin.

The `movem` instruction is a special move instruction that moves up to 16 long words (32 bits) to or from the register banks. Such an instruction can never be implemented as one internal VLIW instruction. Instructions such as `movem` could be implemented as procedures resident in system ROM. The instruction would then be translated to a call to the address of the ROM code that implements the instruction.

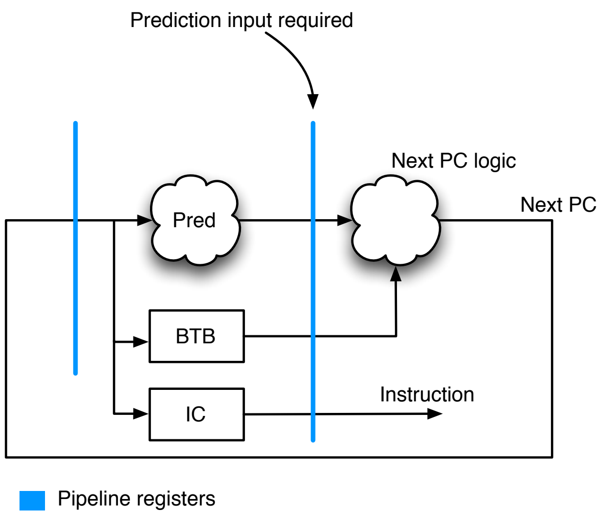

Figure 5.1: Branch prediction latency visualization.

5.1.2 Branch Prediction

It was decided to use a simple bimodal branch prediction scheme for branch prediction. This decision was taken in order to (1) cut down implementation time, and (2) because the most efficient prediction schemes require multilevel memory access.

A multilevel memory access during instruction cache access do not go well with the intended design. The latency from the point where the next PC is registered to the point where the branch predictor output is required is one cycle as seen in figure 5.1. Thus, the chained memory look-up cannot use block ram structures that include output registers, if the block ram cells are to run at the same clock as the rest of the pipeline.

5.1.3 Execution Modes

The execution engine has two modes of operation, direct mode and optimized mode. In direct mode the execution engine is translating m68k instructions to internal VLIW instructions on-the-fly. The translation is performed in hardware by the instruction cache controller. When an instruction cache miss is signaled by the execution engine in direct mode, the instruction cache controller fetches m68k instructions from main memory, translates them to internal instructions and injects them into the instruction cache.

The internal VLIW instruction format encodes control information that controls the data paths in the pipeline. In this sense, the execution engine does not decode the instructions much. Encoded in the instructions are source and destination selection, CCR control, branch control and ALU control. The instruction format is automatically generated from a description file. Methods to access the fields of the instructions are automatically generated as C and VHDL. The size of an instruction is currently around 100 bits. The instructions are laid out as 128 bit fields in memory. In order to optimize memory performance, the 128 bit wide fields must be aligned to 16 byte boundaries. This allows for two instructions per cache line.

One m68k instruction may be translated to a sequence of more than one internal VLIW instruction. Therefore, when executing in direct mode, the m68k PC is extended with two bits at the least significant end. This allows a m68k instruction to be expanded to 4 internal VLIW instructions. As most instructions (at least 95.25% for the benchmark presented above) do not expand to more than one internal VLIW instruction, operating in this mode wastes a lot of instruction cache space. In addition to this, because m68k instructions consist of a variable number of 16 bit words, not every memory location holds a valid m68k instruction. Locations in the data cache corresponding to such locations in main memory must be marked as containing no valid instructions. Figure 5.2 depicts the mapping concept.

It may be possible for the execution to cache the length of the translation of each instruction. This can help improve performance by allowing the execution engine to skip the empty slots. This concept shall be referred to as *length prediction*. Length prediction was not implemented, but is set aside as a possible future improvement.

In optimized mode the execution engine executes internal VLIW instructions that were translated by the host CPU after a hot spot detection. Figure 5.3 shows how the host CPU optimizer might pack the instructions for better instruction cache utilization.

The hardware based on-the-fly translator is unable to perform such a compression of the instruction stream, because this requires a more global view of the control flow of the program being executed. A branch to an address must be rewritten in order to arrive at the address for the correct internal VLIW instruction. To do this, the translator has to maintain information describing how instruction addresses was remapped. This is not feasible for a hardware

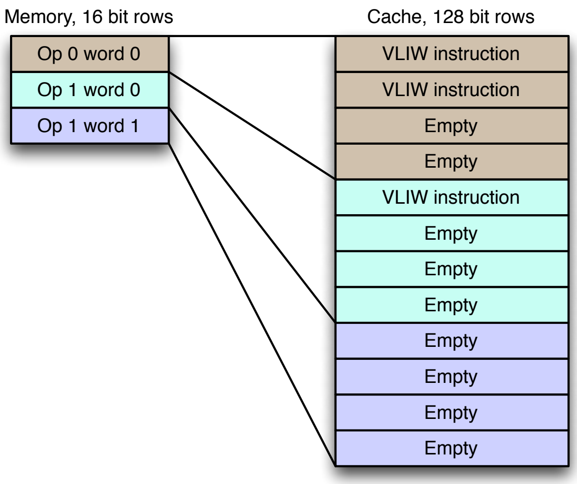

Figure 5.2: On-the-fly mapping of m68k instruction memory locations to instruction cache locations.

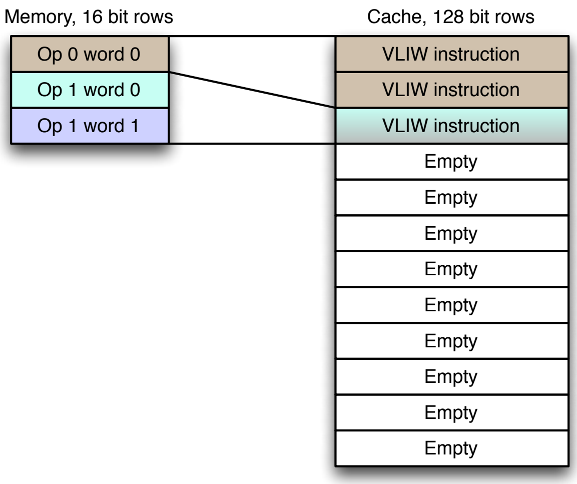

Figure 5.3: Optimized mapping of m68k instruction memory locations to instruction cache locations.

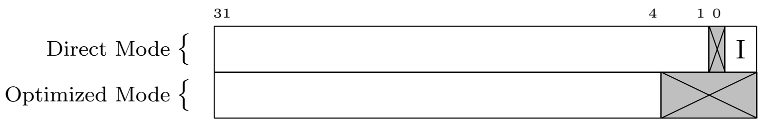

Figure 5.4: PC in relation to memory locations in direct and optimized execution modes.

translator.

Because the on-the-fly hardware translator has these limitations, it is important the hot spots are detected quickly, so that the cache can be properly utilized.

The operating mode of the execution engine is important when the execution engine resolves branch targets or uses PC relative addressing modes. Refer to figure 5.4. The figure shows how the direct mode PC and the optimized mode PC is related to main memory addresses. When the execution engine is operating in direct mode, the PC is effectively 33 bits long. The two lower bit are used to address sub-instructions that originates from the same m68k instruction. These bits are not used in relating the PC to a memory location. Bit 0 in the figure is not part of the PC because m68k instructions must be 2 byte aligned. Therefore, bit 0 of the main memory address is always 0 when addressing main memory instruction locations in direct mode. Bits 31 down to 1 form the most significant bits of the main memory address in direct mode.

When the execution engine is operating in optimized mode it is executing internal VLIW instructions directly from memory. As the instructions must be alined to 16 byte boundaries, bits 3 down to 0 of the internal PC are always 0 when referencing instruction locations in main memory in optimized mode.

When the execution engine operates in direct mode, the instruction cache is indexed using bit $1 + \log_2(i) - 2$ down to bit 1 concatenated with the two bits I, where i is the instruction cache size. In optimized mode, the instruction cache is indexed using bits $4 + \log_2(i)$ down to bit 4. In direct mode, branch targets can only be resolved to cache locations that correspond to 2 byte aligned memory locations. That is, cache locations for which the two bits I and bit 0 are 0. These are the memory locations that can hold valid m68k instructions. When the execution engine operates in optimized mode, branch targets can be resolved to any cache location. That is, any PC where bits $4 + \log_2(i)$ down to 0 are all 0. These observations must be taken into account when implementing the branch resolution logic.

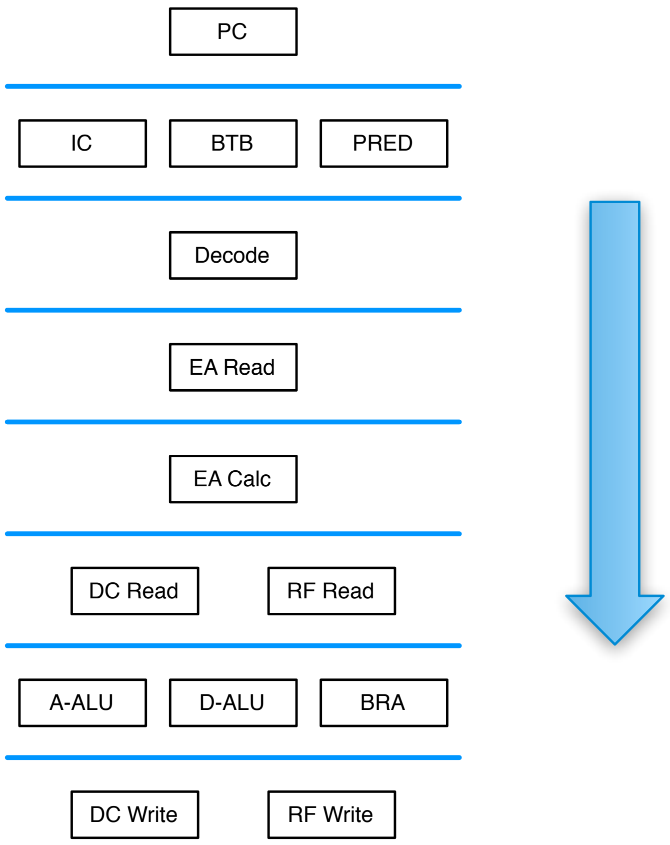

Figure 5.5: Pipeline layout overview

5.1.4 Pipeline layout

This section gives an overview of the execution engine implementation. The execution engine is implemented as a statically scheduled in-order pipeline. An overview of the pipeline layout is depicted in figure 5.5. The blue lines symbolize pipeline registers.

The execution engine pipeline is divided into the following logical components with accompanying short names:

- Program Counter, PC.

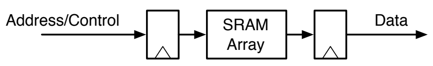

Figure 5.6: Block ram macro cell

- Instruction Cache, IC.
- Decode, DEC.
- Effective Address Read, EAR.
- Effective Address Calculation, EAC.
- Operand Read, OPR.
- Arithmetic Logic Unit, ALU.
- Write Back, WB.

The PC stage calculates the next address to feed the instruction cache. In normal straight line code, this corresponds to incrementing the previous PC. The PC update requires different logic depending on the current execution mode.

To operate block ram macrocells at full speed, the SRAM array inputs and outputs must be fully registered as depicted in figure 5.6. In most if not all FPGA architectures, these interface registers are build into the block ram macrocell. The output register is usually optional, and data can be routed arround the output register. However, using this option will add a significant delay to paths that depend on block ram output. Thus, the pipeline registers arround the IC stage correspond to the input/output registers of a block ram macrocell. The IC is queried along with branch target buffer and branch predictor memories.

The DEC stage is responsible for precomputing control signals for use in subsequent pipeline stages. It is also repsonsible for performing IC hit detection, BTB hit detection and evaluating branch prediction memories. Based on output from the branch target buffer and the branch predictor memory, the decode stage generates control signals that determine if the next address to enter IC should be a cached branch target or simply an incremented version of the PC. Decisions related to the decision of the next PC that are made at this stage are speculative. Branch resolution occurs later in the pipeline, and corrective measures must be taken in case of a wrong prediciton.

Because of the intricate addressing modes supported for memory operands, a section of pipeline stages has to be devoted to address calculation. The EAR stage reads from register files the appropriate values required to construct the effective address. Like the instruction cache, the register file interfaces are guarded by interface registers, and a read takes effectively two cycles.

The EAC stage calculates the effective address using the values read in the EAR stage. The calculated address is fed into data cache read ports of the next pipeline stage.

The data cache memory is read in parallel with the register files in the OPR stage to feed the ALU stage with operands. The ALU stage performs some operation on the operands and optionally updates the condition code register. The pipeline provides one ALU that supports data register targets and one ALU that supports address register targets. Because the branch unit has a data dependency on the condition code produced by the ALU unit, the branch unit cannot be placed earlier than the ALU units.

The ALU stage feeds the writeback stages. The ALU result may be written directly to the the data cache, or to one of the register files.

As mentioned earlier, the DC memory offers only two read/write capable ports. The cache controller uses one port, which leaves the execution engine with only one port. This means that the DC read and DC write stages have to share a single port to the DC memory. This removes potential complexity related to the ordering of loads and stores that would arise if the execution engine had access to two DC memory ports. However, it also limits the performance of the pipeline by introducing a structural hazard around DC memory access.

5.1.5 Effective Address Calculation

The effective address calculation pipeline stage is depicted in figure 5.7. It creates a memory address from a base address, an immediate field and an index register. The base address may be an address register, the program counter, a special TMP register (the use of which is elaborated later in the text), or 0. A 32 bit immediate field is added to the base address to create two variants of the base address.

The index register may be either an address or data register in 8 or 16 bit sign extended version or 32 bit version. The index value is shifted 0, 1, 2 or 4 times to the left. It is also possible to use zero as the index value.

The effective address calculation circuit produces two vales; the base value prior to the 32 bit immediate add, and the sum of the base value, the 32 bit immediate

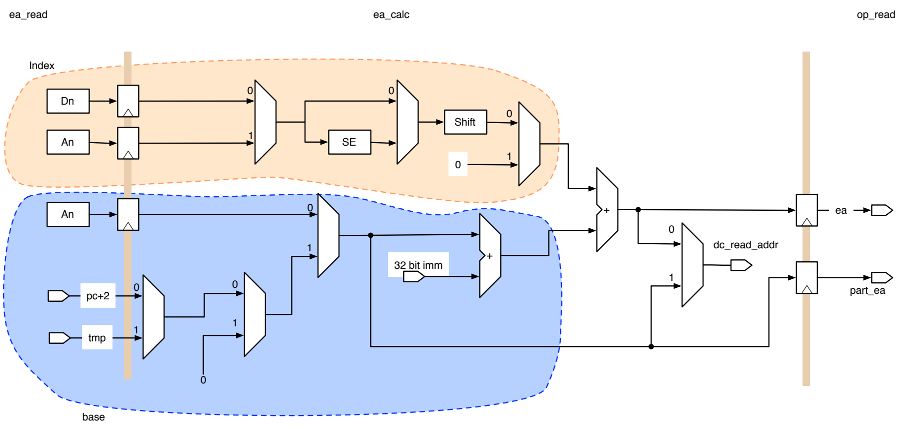

Figure 5.7: The effective address calculation circuit.

and the index value. The former value is referred to as the *partial effective address* while the later is refereed to as the *effective address*.

Using this configuration it is possible to calculate most m68k ISA addressing modes. Address register direct mode is achieved by placing a zero in the 32 bit immediate and suppressing the index register using the last MUX on the index path.

Address Indirect with Predecrement is implemented by placing -1, -2 or -4 in the 32 bit immediate field, depending on the operand size. Address Indirect with Postincrement is similarly implemented by placing +1, +2 or +4 in the 32 bit immediate field. Note that both the unaltered address register value and the incremented value is available as outputs from the circuit. Thus the address register can be updated with the altered value in the write back stage.

Address Indirect with Displacement is implemented by placing the displacement in the 32 bit immediate field. Address Indirect with Index and Displacement mode is implemented by allowing the index value to be added to the base value.

In a similar fashion, the PC relative addressing modes can be implemented by using the PC input as the base value. Absolute addressing modes are implemented by suppressing the base source and using only the 32 bit immediate to form the base address.

However, double indirect access modes require two data cache accesses. To implement these, the TMP register is used to hold the result of the first data

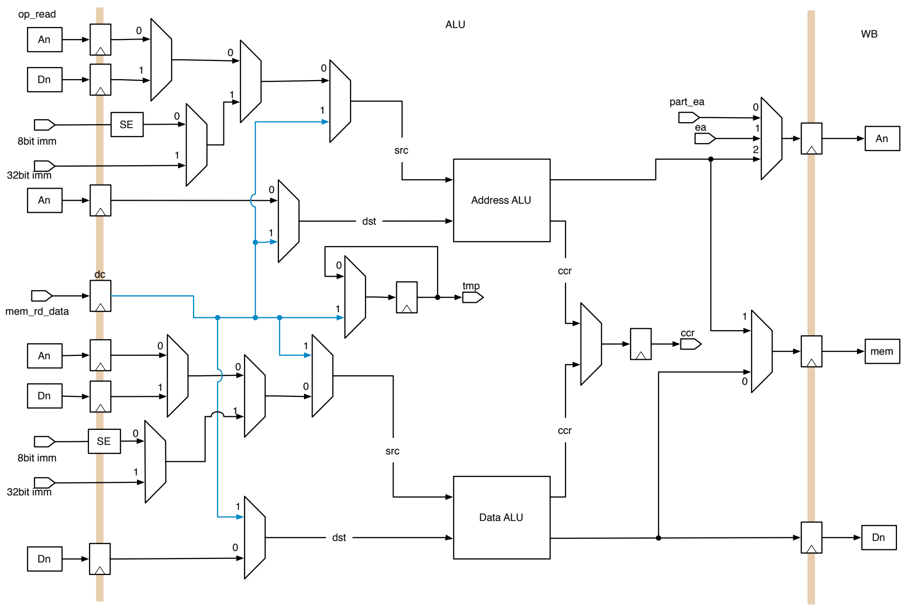

Figure 5.8: The ALU and surrounding pipline stages.

access. This value of the TMP register can then be used as the base for a subsequent effective address calculation.

5.1.6 ALU

A detailed view of the ALU and surrounding pipeline stages are depicted in figure 5.8. The stage receives values from the OPR stage, selects the correct operands, feeds the ALUs and configures the input for the Write Back stage.

The source for the Address ALU may be an address register, a data register, a 32 bit immediate, a sign extended 8 bit immediate or a memory operand provided by the data cache read port. The 8bit immediate and the source register index may share the same instruction word bits, because neither can be used simultaneously. The destination operand of the Address ALU can be an address register or a memory value.

The Data ALU operand selection circuit has a similar structure, except that the destination is a data register or a memory location.

The write back source selection circuitry placed after the ALUs decide what values are served to the write back stage. The Address Register File may receive the Effective Address, the Partial Effective Address or the Address ALU result. This allows automatically incremented values to be written back to the register file, in the same cycle as the result of the operation is written to memory.

The Data Register File can only receive values from the Data ALU. The data cache write port may receive a value from either of the ALUs.

A special temporary register is provided in order to ease implementation of double indirect memory addressing modes, as described in section 3.1.1.

The ALUs also produce the condition code bits, to be stored in the condition code register. In order to optimize throughput, both ALUs must be able to update the CCR. Even though most operations that use an address register destination do not update the CCR, the address ALU can also be used for instructions that use a memory destination, and these instructions may need update the CCR. Thus it is necessary to implement a path for both ALUs to update the CCR.

5.1.7 Branch Unit

The branch unit is placed in parallel with the ALU stage. The branch decision logic is depicted in figure 5.9. The next-pc logic is depicted in figure 5.10.

The next PC is predicted in the DC stage. The prediction relies on a Branch Target Buffer, BTB, and a simple bimodal Branch Predictor, BP. The BTB and the BP are indexed using the next PC from the PC stage. If the branch prediction logic predicts a branch to be taken, and the target address is resident in the BTB, then the next PC is changed to this target.

Because of the two cycle delay of the instruction cache, one instruction after the branch is read from the instruction cache when a branch is predicted taken. To be able to utilize this instruction, the architecture uses one delay slot.

Wrong predictions are detected by the branch unit in the ALU stage (figure 5.9). The branch unit logic calculates the correct target PC and compares it to the predicted PC. The CCR is also checked against the condition code field of the instruction. If any of these two checks fail, the next PC is changed to reflect the correct target, the BTB memory is updated and the pipeline is flushed. Flushing is achieved by injecting a NOP instruction into all relevant pipeline stages. This can be implemented by using the set/reset logic of the register.

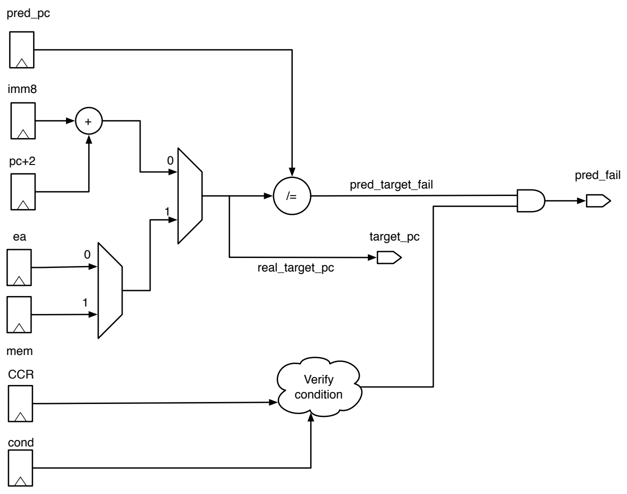

Figure 5.9: The branch decision logic.

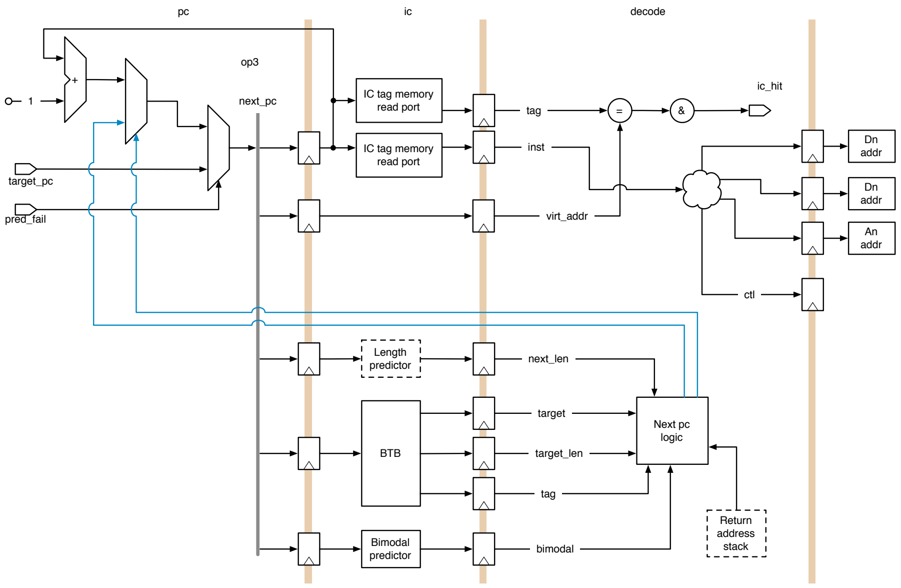

Figure 5.10: Next PC logic. The elements stroke with dashes were not implemented.

Because the branch resolution is completed before any architected state is updated, correct execution is ensured. The penalty for a wrong prediction consists of cycles to re-fill the pipeline from PC to ALU.

The branch unit has to be placed in parallel with the ALU, because the ALU computes the CCR bits. Thus, a branch cannot be resolved before the previous instruction has passed the ALU stage.

5.2 Forwarding and Hazards

There are two places in the pipeline where values are consumed and produced – the effective address calculation stage and the ALU stage. Each of these are candidates for forwarding and stalling. The next sections elaborate the conditions that must be effective to trigger forwarding or stalling.

5.2.1 Effective Address Calculation

The register values used by the EAC stage may be produced by the ALU stage or the EAC stage itself. More specifically, the address register values may be produced by either the EAC stage or the ALU stage, while the data register value can only be produced by the ALU stage.

If a value required by the EAC stage is produced by the ALU stage, there must be a number of instructions between the instruction in the EAC stage that use the ALU produced value and the instruction in the ALU stage that produces the value. If this is not the case, the pipeline must be stalled until the value becomes ready.

If the EAC or ALU stages have produced values required by the EAC stage, but not yet written the values back to the register file, then the value must be forwarded to the EAC stage in, order to achieve optimal pipeline throughput.

Moreover, if the value is to be read from the register file, the value must be resident in the register file, before the read address is clocked into the register file read port interface register. This is due to block ram read/write synchronization semantics. That is, writing a location of a block ram from one port and reading the same location from another port on the same clock edge yields either old data or invalid data, depending on block ram configuration. In addition, some

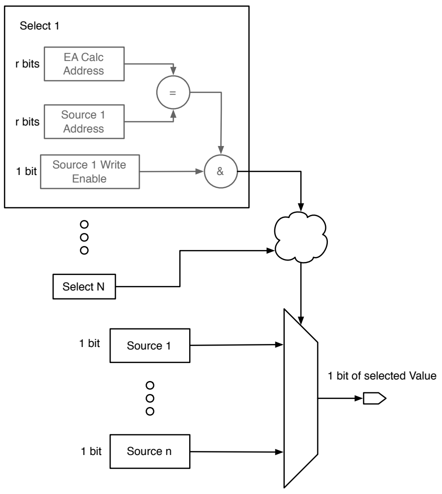

Figure 5.11: Selecting the correct source among forwarding candidates and original source.

block ram macro cells are able to operate on a higher frequency if reading from and writing to the same location simultaneously is not required [Xil12].

There are a number of possible stages that may act as an EAC forwarding source. The address registers may be sourced from EAC, OPR, ALU and WB (WB because of block ram synchronization). The data registers may be sourced only from ALU and WB.

The correct source for a forwarding destination is found by:

1. Deciding if any of the forwarding sources produce a value for the required register.
2. Selecting between these values the value produced by the most recently issued instruction.

Relevant stages of the pipeline carry control bits that specifies what destinations the instruction currently occupying the stage will produce values for. Figure 5.11 depicts the use of these values to select a source for a forwarding target. The box named "Select 1" contains the selection logic for comparing the source of one stage with the destination of another stage. The logic includes a comparator to detect if the register indexes of the two stages match. If the index match, and the stage that might produce the value writes the register defined by the index which was used in the comparison, the output of the circuit is high. If the width of the register indexes is r, then each select box has $2r + 1$ inputs. If n pipeline stages may produce values that might be a forwarding source, n of these decision circuits are required. The output of the decision circuits are used to select between the actual values from each of the pipeline stages. As there are n sources to select from (source 1 to source n in the figure), n bits if input to the selection logic that implements the multiplexer is required. The total number of additional inputs is the sum of the preceding inputs:

$$i_e = (2 \cdot r + 1 + 1) \cdot n \tag{5.1}$$

where r is the register index width and n is the number of sources. The expression shows that the number of sources and the width of the register addresses both affect the number of inputs. The number of inputs in turn limits the maximum operating frequency of the circuit. The consequence is that if too many extra pipeline registers are added to the pipeline, it will become unfeasible to allow forwarding from all of them. Similarly, if the number of registers become too high, the number inputs to the forwarding detection logic may become so large that they limit operating frequency.

Stall decisions require similar logic, but one bit less input per stall source, as the value is not yet produced.

5.2.2 ALU

Logically, values consumed by the ALU stage can only be sourced from the register files or the data cache. In practice, values must be forwarded from the ALU itself, and from the WB stage. As with the EAC stage forwarding, the WB stage must be considered due to the semantics of the register banks.

A construct similar to the one depicted in figure 5.11 is used to determine the correct source to supply the values for the ALU stage.

5.3 Performance Considerations

Preliminary performance evaluation of the described pipeline layout was carried out by building a VHDL model of the pipeline. The model was synthesized using the Xilinx Vivado suite targeting the Zynq XC7020-1CLG484C device. The first model to be synthesized included additional pipeline registers separating

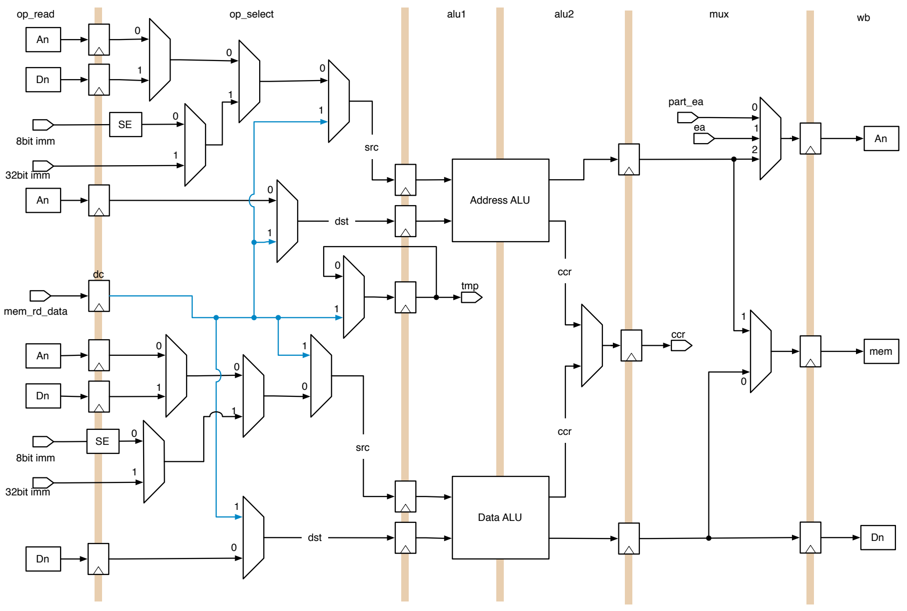

Figure 5.12: The ALU stage after inserting additional pipeline stages.

the ALU stage into OPS, ALU1, ALU2 and MUX as described below. These registers were placed based on earlier experience with targeting FPGAs. This initial design achieved a clock frequency of 80MHz. Using the Xilinx tools to drive the optimization process, additional pipeline registers were inserted into the pipeline.

The ALU stage was divided into four separate stages. Figure 5.12 depicts the location of the additional pipeline stages. The selection of the operands that feed the ALUs is given a separate stage named Operand Select, OPR. A pipeline register is placed inside the ALUs in order to prevent long carry chains in combination with CCR update in limiting the frequency. This creates the two stages ALU1 and ALU2. The selection of values for the WB stage is also given a separate pipeline stage named MUX.

The EAC stage also requires additional pipelining. The extensive forwarding to the base register, combined with the two 32 bit adders became a critical path. The stage is divided into two separate stages EAC1 and EAC2 as seen in figure 5.13.

With most of the critical paths handled by inserting additional pipeline stages, the logic that decides the next PC becomes the critical path. The branch res-

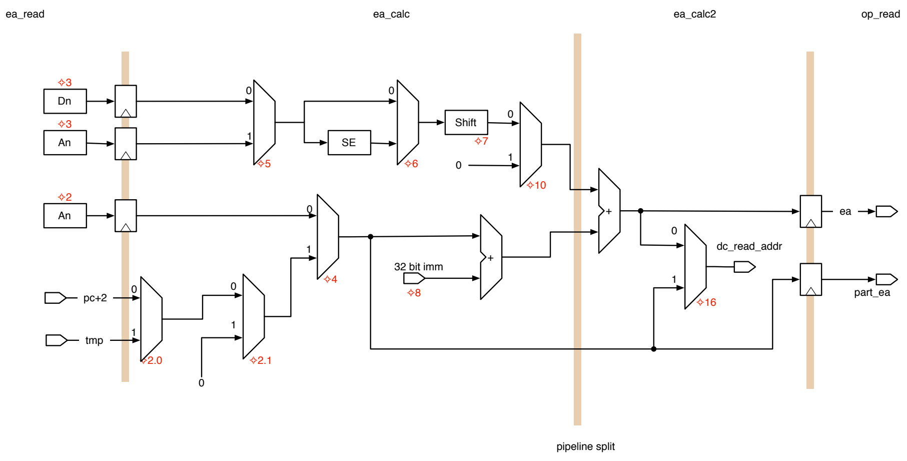

Figure 5.13: Pipelining of the EAC stage.

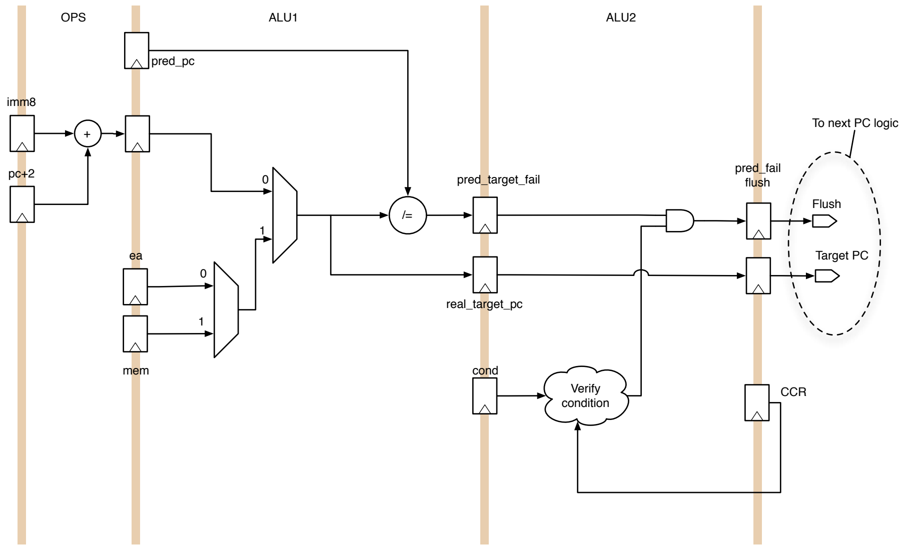

Figure 5.14: Pipeline registers inserted into the branch resolution logic.

olution logic is capable of flushing pipeline stages from PC to ALU2 in case of a wrong branch prediction. This causes logic paths from the branch resolution logic to the inputs of all the pipeline registers that require flushing to limit the operation frequency. By storing the flush decision in a register and acting on the registered value in the next clock cycle, this path is effectively eliminated. This is achieved by registering the flush decision at the ALU2 pipeline register level as depicted in figure 5.14.

The verification of the condition code cannot be moved earlier than ALU2, because the condition code that the decision logic must act on is produced by a previous instruction in the ALU2 stage. However, matching the condition code, and verifying that speculated target PC was correct became a critical path. The speculated target PC verification is moved to the ALU1 stage to kill this path. Further, calculation of the 8 bit PC relative address is moved back to the OPS stage.

5.3.1 Pipelining the Forwarding Logic

The consequence of adding additional pipeline registers to the pipeline is that a number of additional forwarding paths and stall conditions are created. An instruction entering EAC1 must now also stall if it depends on a value produced by EAC2 that has not yet been latched in the EAC2 output registers. Similarly, an instruction in EAC1 must be stalled if it depends on a value that is produced by an instruction in ALU2 if that instruction has not yet reached ALU2.

Forwarding of the address index register value or the base register value for the effective address calculation circuit now has no less than 9 sources, in addition to the register file. In order to avoid the selection of the correct source to become a frequency limiting path, the forwarding itself is pipelined. Figure 5.15 presents a conceptual overview of the pipelined forwarding logic for an address register source in the EAC1 stage.

In the DEC stage of the pipeline, 5 sources are monitored for forwarding conditions. Instructions in EAC2, OPR, OPS and ALU1 may have produced an address register value when they passed EAC2. The instruction in ALU2 may have produced an address register value via either EAC2 or ALU2. The forwarding sources are effectively checked for forwarding conditions before the values are required for forwarding. This moves some of the pressure away from the pipeline stage that is the destination of the forwarding to some of the earlier stages.

In the EAR stage, the 5 sources checked in DEC are condensed down to 1

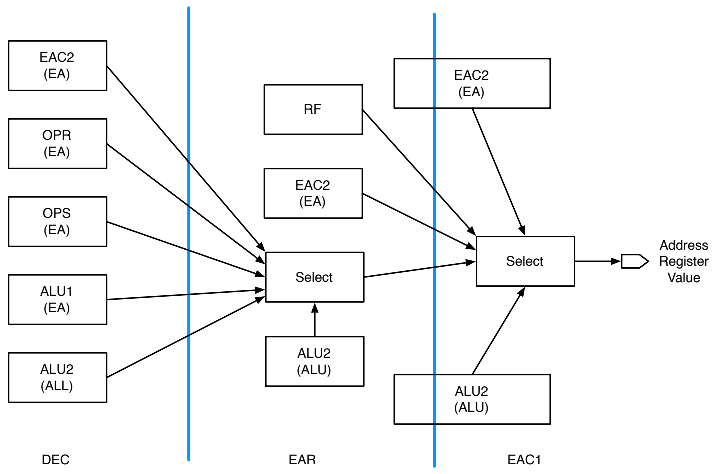

Figure 5.15: Conceptual overview of the pipelined forwarding logic for the address register source of the EAC stage. The vertical blue lines represent pipeline registers.

source using a priority selection scheme. In addition, the instruction that has moved into ALU2 may have produced a value that was not available when the instruction was in the ALU1 stage. If it is required for forwarding, it is captured at this point. The instruction that has moved into EAC2 is checked and the result of the check is propagated to the EAC1 stage. Partial checks for the instructions that will be in EAC2 and ALU2 in the next clock cycle are also performed in the EAR stage.

In the EAC1 stage, the final source for the operand is determined. The condensed value from EAR, the EAC2 check from EAR and the EAC2 and ALU2 checks for EAC1 are reduced to a single value. If no forwarding is required, the register file is used as a source.

Pipelining the forwarding logic in this way presents a challenge in relation to pipeline stalls. A decision made in one stage to capture a value for forwarding in a subsequent stage may capture the wrong value if not implemented carefully. This can happen if the first part of the pipeline (including the stage in which the capture decision is made) stalls just after the capture decision has been made, but the later stages of the pipeline continue to execute. When the stall condition is resolved and the instruction in capture decision stage can move forward, it will capture the forwarded value as decided by the logic in later stage. But this decision was made when the last part of the pipeline contained

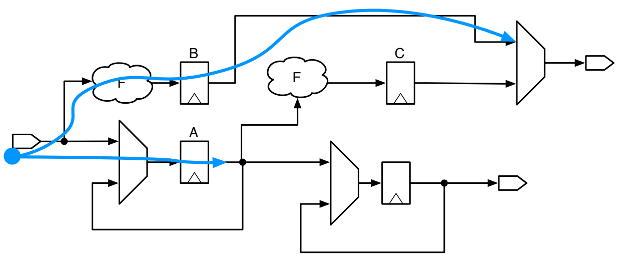

Figure 5.16: Normal operation of gated and ungated registers.

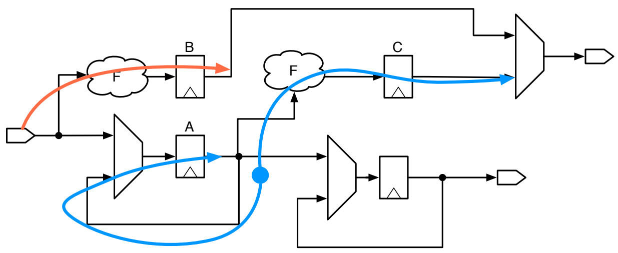

Figure 5.17: Operation of gated and ungated registers during a stall.

other instructions. These instructions may have left the pipeline, or be in a different location when the stall condition is resolved.

In order to overcome this problem, the pipeline registers that capture the output of the forwarding logic are never stalled. Thus, if the first section of the pipeline is stalled for some reason, while the last section of the pipeline continues to execute, the forwarding selection pipeline registers are continuously updated to reflect the actual state of the pipeline.

However, having some registers that are never stalled (ungated registers) in combination with registers that are stalled (gated registers) gives rise to further complications. When the pipeline is operating in normal mode, an ungated register that captures the output of some function F uses the same input as a regular gated register at the same pipeline level. This situation is depicted in figure 5.16. The output of the ungated register B is the result of function F with the same input as is present on the output of gated register A.

Figure 5.17 shows the data flow when the pipeline is stalled. The output of register B now holds the value of F based on a different input than the one available at the output of register A. Using the value at the output of register B

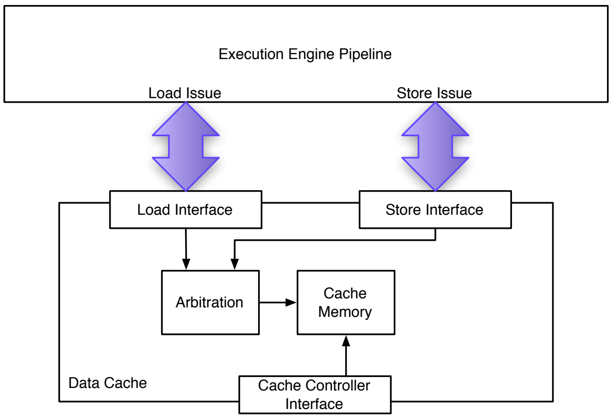

Figure 5.18: Overview of the integration of the data cache and the execution engine pipeline.

as a representation of the function F with the input at the output of register A would be wrong. Instead the output of register C is used to provide the value. As an alternative to the depicted solution, the input to F might be multiplexed instead of multiplexing the registered output. Depending on the criticality of the paths from the inputs to F and into the register versus the paths from the register and on, the placement of the multiplexer can be moved.

Thus, a function that uses the output of an ungated register as input must consider twice as many inputs as if the input was provided unregistered. If the function F depends on many input signals and produce few output signals, the use of ungated registers may still lower the number of inputs to a function that use the result of F.

5.4 Data Cache

A conceptual overview of the Data Cache is given in figure 5.18. Because one port of the data cache must always be available to the cache controller, the pipeline DC read and DC write stages must share a single port to the cache memory. This complicates implementation and presents additional hazards to the pipeline.

The data cache is arranged as an array of 32 bit locations. A cache line holds 8 32 bit words to form a 32 byte cache line. This line size was chosen to match

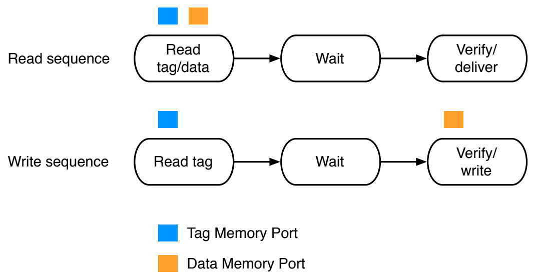

Figure 5.19: Data cache load and store sequencing and resource utilization.

the maximal burst length of DDR3 memory [Jed12] on a 32 bit wide bus as featured on the on the ZedBoard [AVN13].

In order to efficiently process stores and service dependent loads quickly, the data cache uses a store buffer. This allows the pipeline to continue execution, even when a store misses in the cache memory. However, store-to-load forwarding and alias detection[2] were not implemented due to lack of implementation time. For the execution engine pipeline presented earlier, the consequence of this choice is that all stores in the execution engine pipeline and in the store buffer must be processed before loads in the pipeline.

Because of delays associated with block ram macro cells, a read of the data cache memory has a latency of two cycles. One cycle to clock in the read address, and one cycle to clock out the data. Therefore, the data cache must to handle multiple operations at once to be effective. However, switching from servicing stores to servicing loads requires some consideration.

To perform a load, the tag and data memories are read in parallel. When the result is ready, the tag is verified, and if the tag indicates a cache hit, the data is handed off to the pipeline. If the tag indicates a miss, the data cache notifies the external data cache controller.

To perform a store, the tag memory is read first. When the tag value is ready, the tag is checked to determine if the store is a cache hit. If the store is a cache hit, the store data can be written to the data cache memory. Once the data has been clocked into the data memory port, the data cache can forget about it. The read and write sequence is depicted in figure 5.19.

Figure 5.20 shows a sequence of load/store operations. Two reads, two writes

[2]Two memory operations that access the same data in main memory are said to be aliasing.

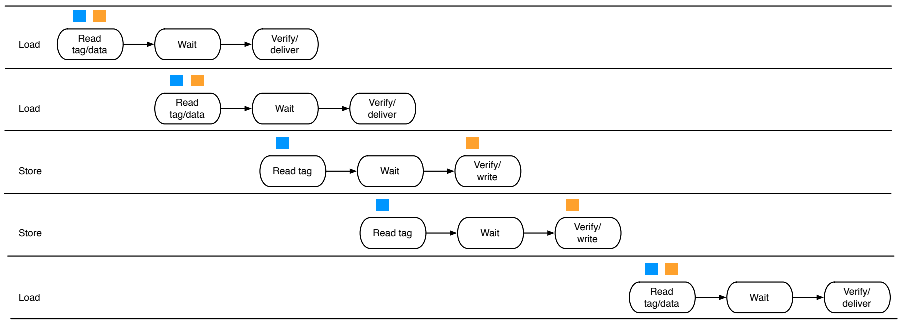

Figure 5.20: Load and store pipelineing in the data cache.

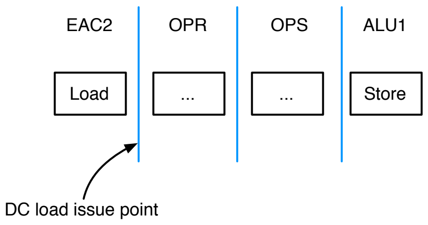

Figure 5.21: A load following a store in program order. The load cannot be issued to the data cache before the store has been issued, unless the two operations can be guaranteed not to be aliased.

and a read are performed in a pipelined manner. The figure shows how reads can be executed in a pipelined manner. Stores can also be executed pipelined after reads, and stores can be executed pipelined after stores. However, as the last operation shows, loads cannot be executed pipelined after stores. This is both because store operation requires use of the data memory port at the end of its execution, and because of the adopted stores-before-loads policy. The control logic must wait for the last store to complete before issuing the load. If the load arrives earlier than it can be issued, the control logic must notify the execution engine pipeline so it can be stalled.

Because of the missing alias detection logic, the situation of stores issued after loads depicted in figure 5.20 cannot legally occur. Figure 5.21 shows the execution engine with a load about to issue and a store in the ALU1 stage. If the load operation is aliased with the store operation, the load must not be issued to the DC before the store has been issued. Alternatively, the value of the store in ALU1 may be forwarded to the load once the store value has been produced. Then the load may skip the DC entirely.

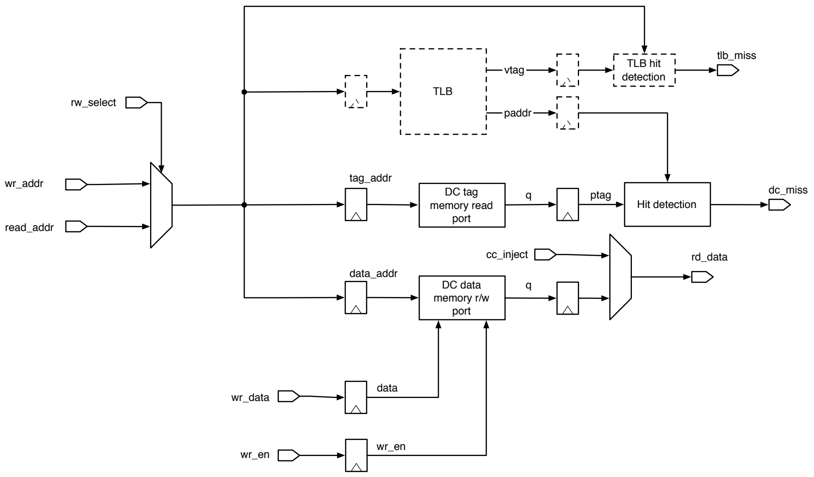

Figure 5.22: Data cache core hardware. The TLB was not implemented.

The data path logic that facilitates the load/store pipelining is depicted in figure 5.22. The address for the memory cells may be routed from the read address or the write address. Note the cc_inject label. When a cache miss is detected, the data cache logic is stalled until the cache controller has resolved the miss. When the data cache resumes operation, it will source the required value from the cc_inject signal provided by the cache controller, instead of from the cache memory. This eliminates the need to restart the operation, and to manage restart of all the load/store operations that were in flight.

In order to handle load/store operation issue and in order to handle unaligned loads and stores, an additional layer of logic is placed between the data cache pipeline and the execution engine pipeline. The data cache has a 32 bit interface to the cache memory and supports 1 byte, 2 byte and 4 byte operand sizes in a byte addressable address space. Unaligned load/store operations that do not cross an aligned 4 byte boundary are handled as a single operation. For these operations, the data is shifted into place before write or after read. Individual byte wide write enable signals of the block RAM macro cell are used to write only the necessary bytes of the 4 byte word. Load and store operations that do cross a 4 byte boundary require two accesses to the cache memory. Such operations are split into two separate operations by the control logic. For writes, nothing more is required. For reads, the data for each of the operations of the split operation must be combined before it is handed off to the execution engine. The sequencing for unaligned reads are depicted in figure 5.23.

The execution engine pipeline interfaces the data cache via the interface depicted

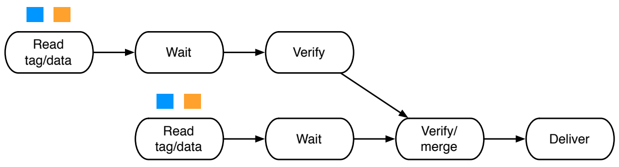

Figure 5.23: Unaligned read that crosses a 4 byte boundary

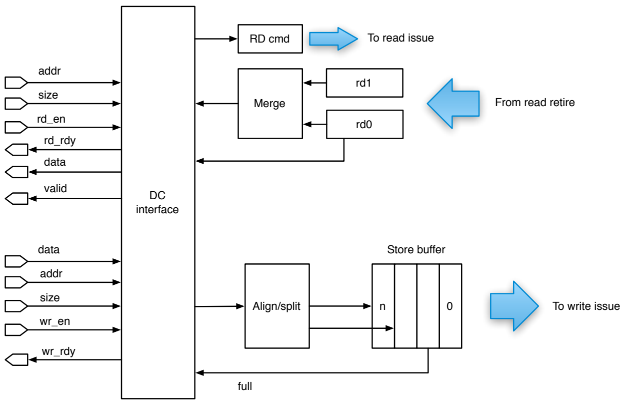

Figure 5.24: Data Cache interface.

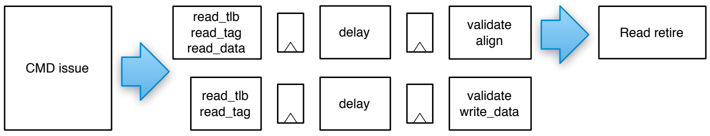

Figure 5.25: Data cache command issue for reads and writes.

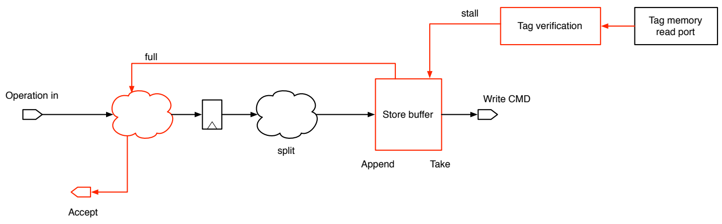

Figure 5.26: Store command issue logic critical path.

in figure 5.24. Stores are accepted if there is at least two free slots in the store buffer. Two slots must be available in order to handle unaligned stores. If the store is unaligned, the store is split in two before being placed in the store buffer.

If the store buffer is empty, the interface will accept a load operation. The result of the load operation will be available on the interface after at least two cycles as depicted in figure 5.25. With store-to-load forwarding, the load would be required to stall. If the address of the load was present in the store buffer, the load could be service from the store buffer. Otherwise the load would be serviced immediately from the data cache, before waiting stores in the store buffer.

Experimental synthesis shows that one path in the data cache is particularly critical for performance. This is the path that originates in the tag validation logic, produces the data cache pipeline stall signal, goes through the store buffer control logic and on to the interface control logic that determines if an operation can be accepted. This path is depicted in figure 5.26. The tag is compared to the address of the load/store operation after leaving the tag memory output registers. If the tag does not match, the stall signal is raised. This stall signal in turn determines whether an operation is removed from the store buffer to be issued to the pipeline. This affects the number of available entries in the store buffer that is reported to the interface logic, and whether the store buffer is able to accept new entries. The store buffer fill level determines whether new store operations can be accepted from the execution engine. If no more operations can be accepted by the data cache, the execution engine pipeline stall logic is

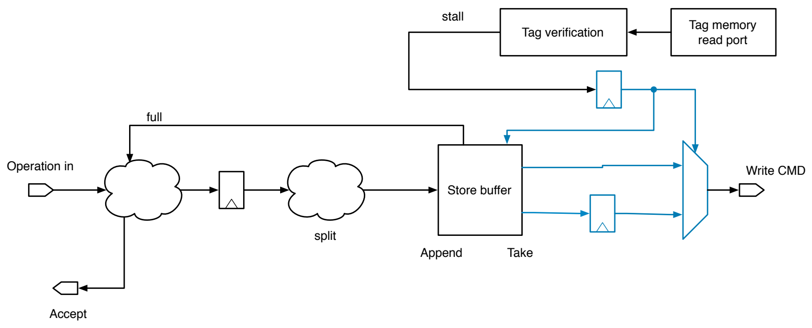

Figure 5.27: Optimized store command issue logic.

activated. This logic already has a high propagation delay, and the whole path ends up limiting the operating frequency of the entire design severely.

To compensate for this effect, the stall logic of the data cache is pipelined as seen in figure 5.27. To break the path, the stall signal is registered immediately after tag validation. This breaks the critical path, but also has the effect that the store buffer sees the stall signal one cycle too late. An additional register is appended to the store buffer to compensate for this effect. The register holding the stall signal is used to select between the value from the extra register and the actual store buffer output. Deciding whether a new operation can be accepted now amounts to checking the availability of the last store buffer slot and the registered stall signal. If the registered stall signal is low, the store buffer will always issue one operation on the next rising edge, and one free slot is enough. If the registered stall signal is high, the store buffer will not issue any operation in the next cycle, and two free slots is required in order to accept an operation (two slots are required to accept unaligned stores).

5.5 Data Cache Controller

The data cache controller takes advantage of the fact that the AXI protocol allows full duplex communication. The controller is constructed as three interleaved state machines. Figure 5.28 presents the state diagram for the data cache controller. The state machines are marked A, B and C. State machine A is responsible for fetching data from memory, B is responsible for writing back dirty lines to memory, and C is responsible for reading and writing the cache memory. The black arrows represent state changes. The blue arrows represent dependencies.

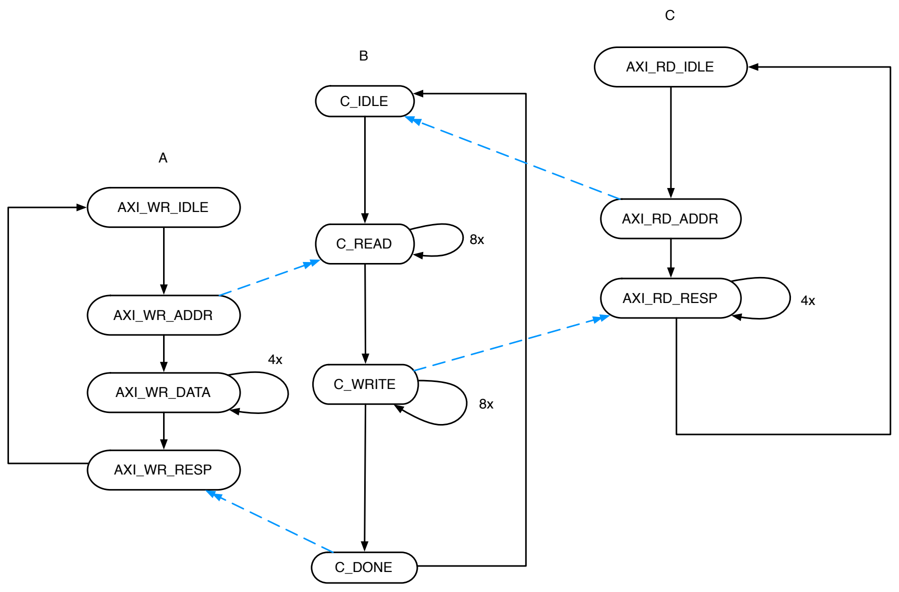

Figure 5.28: State diagram for the data cache controller logic.

Even though the execution engine was designed with multiprocessor operation in mind, no effort has been made to make the cache controller able to operate in such an environment.

5.5.1 Short note on the AXI Protocol

The AXI protocol is a flexible master/slave micro controller bus standard developed by ARM Ltd. AXI is widely used in ASIC and SoC parts and for FPGA IP.

An AXI read transaction is a two phase process. The first phase is the address phase. In this phase, the address is driven on the address bus by the master, and the slave drives accept or error signals on the response bus. The master also drives signals that describe the size, width and type of the transaction. The data cache controller uses 4 burst 64 bit wide transfers in order to transfer an entire 32 byte cache line in one transaction.

The second phase is the response phase. In the response phase, the slave drives the read data bus and the read response bus, and the master drives the accept signals. The response phase may contain many transfers to implement a burst transfer.

The AXI write transaction is similar to the AXI read transaction, but requires one more phase. The address phase is very similar to the AXI read transaction address phase. In the second phase, data flows from the master to the slave, as opposed to the second phase of the AXI read transaction. The third phase is a response phase in which the slave indicates whether the transaction was successfully completed.

For more information on the AXI protocol, refer to the AMBA, AXI and ACE specification [ARM11].

5.5.2 Handling a Cache Miss

When a cache miss is signaled by the data cache logic, state machine B changes state to the c_read state where it moves a cache line from the data cache memory to an internal buffer. As machine B moves out of c_idle, state machine C is allowed to enter the axi_rd_addr state, as shown by the blue dashed arrow in the figure. When state machine C enters axi_rd_addr, it initiates a read of the cache line in main memory that contains the word that caused the cache miss. State machine C is allowed to proceed to the next state where it reads the response from the AXI bus read data channel and places the response in an internal buffer. Once the response has been read into the internal buffer, state machine C is finished.

State machine B is allowed to enter the c_write when state machine C leaves the axi_rd_resp state, as indicated by the figure. In the c_write state, the internal buffer filled with data read by state machine C is written back to the data cache memory.

State machine A waits for state machine B to leave the c_read state before entering the axi_wr_addr state. In axi_wr_addr, an AXI write transaction is issued in order to write the dirty cache line back to main memory.

State machine B must wait for state machine A to finish, before it signals the execution engine that the cache miss has been resolved.

If the cache line that must be evicted is not dirty, state machine A is kept idle, and state machine B skips the c_read state.

5.6 Instruction Cache

The instruction cache is organized as an array of 128 bit instruction words with two words per 32 byte cache line. Because the instruction cache is a read only memory for the execution engine, implementation is straight forward.

As opposed to the data cache, the instruction cache tag is evaluated by the execution engine itself. When a miss is detected no-op instructions are injected into the pipeline until the cache controller has resolved the miss. No other logic is changed. For instance, the program counter continues to progress during a cache miss.

When the cache controller has resolved the miss, an artificial branch instruction is executed to transfer control back to the instruction that caused the miss. The logic to perform this action is the same logic that is used during a reset operation. This approach is chosen in favor of stalling the front end of the pipeline, because the stall logic must be inserted in all critical paths.

5.7 Instruction Cache Controller

The Instruction Cache Controller, ICC, is responsible for resolving cache misses in the instruction cache. When operating in optimized mode, the ICC services a cache miss by fetching the relevant cache line from main memory via an AXI master interface. The line is placed in the cache memory, the execution engine is notified that the miss is handled.

When operating in direct mode, the ICC services a cache miss by translating m68k instructions to internal VLIW instructions. The miss address identifies the address in memory where the m68k instruction to be translated is resident. To avoid problems with m68k instructions that span two cache lines, two consecutive cache lines are always fetched.

The m68k instruction is decoded by a state machine over two cycles. The first cycle examines the first 16 bit word of the m68k instruction. Based on this examination, a sequence of VLIW instructions is emitted in the next cycle.

When the instruction has been decoded and the VLIW instructions written back to the cache memory, the execution engine is notified and execution continues.

Chapter 6

Hot Spot Detection

The hot spot detection mechanism employed in this project is adapted from work by Merten et al. [MTG$^+$99, MTN$^+$00, MTB$^+$01]. During the early stages of the project I planned to use a trace based method for detecting hot spots. I was entrigued by research carried out by Fahs et al. that seeks to use the trace cahce as a dynamic optimization target [FBC$^+$01a, FMS$^+$04b]. However, trace based methods described in literature bring a lot of complexity to the hardware. The trace has to be constructed by hardware, and it has to be placed in a hardware buffer. The trace buffer space is wasted if the trace is shorter than the buffer entry size. Also, a trace captures only a single path through a hot spot. Even though this path may contain loops that are effectively unrolled by the trace, a trace can still only capture one path.

In contrast, the method described by Merten et al. captures information, not about traces, about heavily executed branch instructions. This information is placed in a cache like structure that can be accessed by the user of the information. It is a simpler and cleaner method which is easier to implement.

The next sections describe the implementation of the HSD mechanism.

6.1 Concept

The HSD detects hot branch instructions by monitoring the execution frequency of the branches executed by the execution engine over a closed time interval (the sampling window). The hot branch instructions are sets of branches, that are active for at least some amount of time, and that constitutes at least a predefined percentage of all executed branches in that same period of time. If such a group of branches can be located, the HSD emits an event that indicates that a hot spot was found. The HSD relies on the execution engine to emit an event each time it retires a branch instruction. The event must inform the HSD unit of the address of the branch instruction, and whether the branch was taken or not.

Let the amount of time that the set of branches must be active in order to be considered a hot spot w_s, and the minimum execution percentage of these branches x_t. The amount of time is not measured in time, but rather in a number of retired branches.

The HSD operates with two sampling windows whose length is measured in retired branches. The Candidate Window, CW, is used to decide if a set of branches are candidates for a hot spot. The Sample Window, SW, is used to decide if a set of candidate branches constitues a hot spot. The length of the SW is w_s, the length of the CW is w_c and $w_s = s \cdot w_c$ where $s = 2^n, n \in \mathbb{N}$. To reiterate, in order to be reported as part of a hot spot, a branch must first become a candidate by achieving a certain execution percentage in the CW, and then the set of candidate branches must hold a second execution percentage over the SW.

When the HSD is activated, it first looks for a set of candidate branches. Candidate branches are branch instructions that have an execution count of at least e_c in some CW. If the CW is w_c branches long, the execution percentage that a branch must have to become a candidate branch during some CW is:

$$x_c = \frac{e_c}{w_c} \tag{6.1}$$

When a set of candidate branches has been identified, the HSD measures how often the candidate branches are executed, and for how long they are active. If the candidate set is active over the entire SW of length w_s branches and constitutes at least x_t percentage of all executed branches, the set of candidate branches is declared to be a hot spot. This decision is carried out by maintaining a saturating counter of n bits that is initialized to $(2^n) - 1$. This shall refer to this counter as the Hot Spot Counter, HSC. Each time a candidate branch is retired, the counter is decremented by D. Each time a non candidate branch is retired, the counter is incremented by I. If the counter reaches zero before the SW ends, the set of candidate branches are determined to be a hot spot.

The values of I and D are chosen so that if the candidate branches have an execution percentage larger than x_t, the counter will begin to move down. If this is the case for a longer period of time, the counter will reach zero.

If x is the execution percentage of a set of candidate branches, the HSC will decrease when the increment value I multiplied by the non candidate execution percentage is less than the candidate execution percentage times the decrement value D:

$$I(1-x) < xD \tag{6.2}$$

Thus, if we want the HSD to decrement for some x, then I and D must satisfy:

$$\frac{I}{D+I} < x \tag{6.3}$$

and therefore I and D can be set to achieve an execution percentage threshold x_t of hot branches as follows:

$$x_t = \frac{I}{D+I} \tag{6.4}$$

Let N be the minimum number of branches to be executed before a hot spot is detected and let x be the execution frequency of a set of candidate branches. Then $D' = NDx$ is the total amount by which the HSC is decremented by, due to candidate branches in a window of N branches. Likewise $I' = NI(1-x)$ is the total amount that the HSC is increment by due to non candidate branches in a window of N branches. The total value by which the HSC is decremented in a window of N branches must then be $D'' = D' - I'$. If we want the minimum number of branches to be executed before a hot spot is detected to be N, the total value by which the HSC is decremented in a window of N branches must satisfy:

$$(2^n) - 1 < D'' = D' - I' = NDx - NI(1-x) \tag{6.5}$$

where n is the number of bits in the HSC. Solving for N yeilds:

$$N > \frac{(2^n) - 1}{(D+I)(x - x_t)} \tag{6.6}$$

From equation 6.6 we see that the minimum number of branches that must be executed before a region is classified as a hot spot decreases if the actual execution percentage, x, of the candidate set is larger than the threshold x_t. N is a not a fixed amount. A hot spot is detected faster if the actual candidate branch execution percentage is higher than the threshold execution percentage. If all branches executed are candidate branches, such as in a tight loop that runs for a long time, then

$$N > \frac{(2^n) - 1}{D} \tag{6.7}$$

Thus, the least detection time can be set by selecting D and n according to 6.7.

6.2 Architecture

The HSD captures branch information in an internal block RAM structure. Ideally, the HSD would capture information on all branches retired by the execution engine in the sampling window. However, due to resource constraints it is not feasible to log information on all retired branches. To conserve resource utilization, the HSD employs a hashing policy as known from cache structures. The lower order bits of a branch address is used to form an index into a memory array, while the upper bits form a tag that is stored in the memory row. Figure 6.1 shows the layout of a row in the memory array.

When the HSD encounters a branch, it uses the lower bits of the branch instruction address to index the internal memory array. If the valid flag of the entry is 0, the entry is unused. It is allocated by setting the valid flag to 1. The tag field is set to the upper bits of the branch instruction address and the two counters are initialized to zero. If the entry is in use, the tag is matched against the upper part of the branch instruction address. If a match is detected, the execute counter is incremented. If the execution engine signaled that the branch was taken, the taken counter is also incremented. If the counters are incremented to the point of overflow, their values are kept instead of being incremented. Even though the values do not represent actual execution frequency, it is still clear that the branch is hot, and by stalling the counters, the branch bias is still available to clients of the HSD. If bit b_c of the execute count field is changed from 0 to 1 by an update, the candidate field of the entry is set. The updated values are stored back in the memory array.

If a branch maps to a slot that is occupied by another branch, the incoming branch is simply discarded. If the discarded branch is in fact a hot branch, sampling precision is lost. Precision can be increased by changing the memory array to an n-way set associative structure.

If no hot spot is found before the SW has passed, the detection memory is reset and the process is restarted.

Tag	Taken Count	Exec Count	Cand	Valid

Figure 6.1: HSD memory row layout.

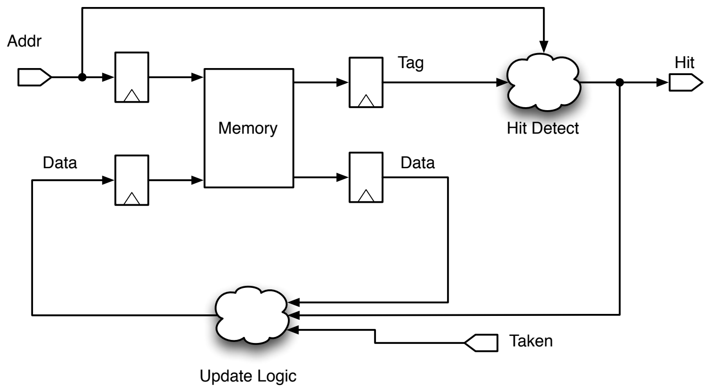

Figure 6.2: HSD memory access logic.

Figure 6.2 depicts the data path hardware constructed to implement the described functionality.

The logic that implements the HSC is depicted in figure 6.3. When a candidate branch is retired the value *D* is subtracted from the HSC. When an non candidate branch is retired, the value *I* is added to the the HSC. When no branch is retired, the HDC keeps its value. When the counter reaches zero, the Detect! flag is raised.

When the CW counter wraps, all entries not marked as candidate branches are expunged from the memory array and the CW counter is reset. This process repeats up to *s* times. If no hot spot is found in *s* iterations, the memory array is cleared and the process can start over.

The state diagram for the control logic of the HSD is depicted in figure 6.4. The HSD starts in the idle state. When the HSD is activated, it transitions to the reset state. The reset state initializes internal state and clears the memory array. The HSD then transitions to the wait_br state where it waits for the execution engine to signal retirement of a branch. If the CW counter expires, the HSD transitions to the rf_read state. The rf_read, rf_wait and rf_write states refreshes the memory array by invalidating all entries that has not reached candidate status. If the execution engine signals that a branch is retired while the HSD is in the wait_br state, the HSD presents the address to the memory

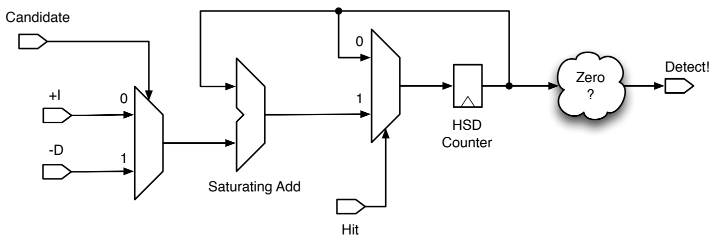

Figure 6.3: HSD counter operation.

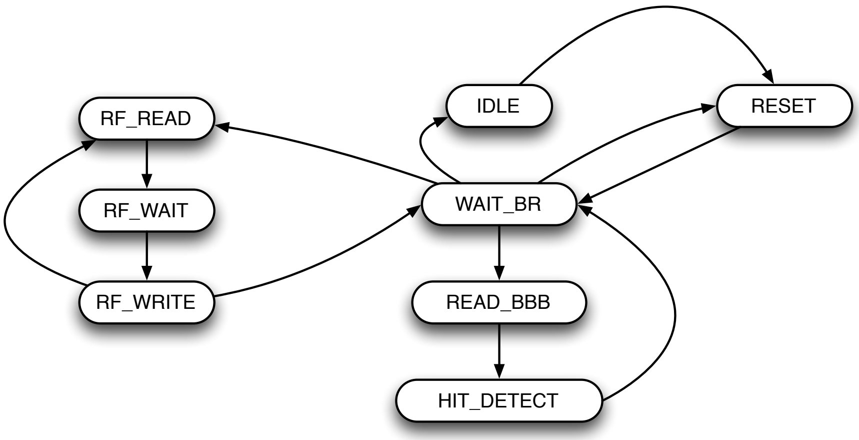

Figure 6.4: HSD state transition diagram.

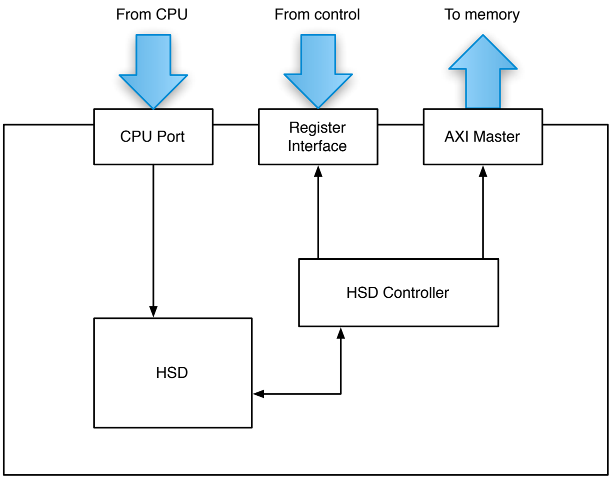

Figure 6.5: HSD infrastructure components.

array and transitions the the read_bbb state. The read_bbb state is a wait state for the read of the internal block ram. In the hit_detect state the tag of the selected memory row is checked against the incoming branch address. If the branch was a hit, or the row was empty, approproate modifications are made to the memory cell before it is written back. Otherwise, no write is performed before the HSD returns to the wait_br state.

The initial implementation of the HSD is not pipelined, and therefore the system may not register control flow events that occur back-to-back. It is possible to solve the problem by pipelining the state machine and using a dual-ported block RAM to implement the memory array.

6.3 Infrastructure

The HSD core is managed by a HSD Controller as seen in figure 6.5. The system is controlled by a register based interface. Execution engine branch events enter the HSD core via a separate interface that connects directly to the execution engine. The HSD Controller is responsible for starting and stopping the core, and for dumping the detected hot spot information to memory. The hot spot information is transferred to main memory via an AXI master interface. A memory mapped register based interface is available to the host CPU to control the HSD. Through this interface the CPU is able start, stop and reset the HSD and to set the address where the profiling information is dumped.

6.4 Monitoring

When a hot spot has been detected and entry points to the optimized code has been installed in the execution engine, it is important to know if execution in fact continues in the detected hot spot. If the path of control strays away from the hot spot, it might be a good idea to restart the hot spot detection procedure to see if another area of code is hot. In the context of this project, it may be feasible to simply restart the HSD and see if it detects a hot spot that is different from the optimized regions. There is no cost associated with this process, because the host CPU is freely available to perform this task. However, in a setup where there is only one CPU, i.e. no extra host CPU, it makes sense to interrupt the CPU only if execution actually moves away from the detected hot spot.

Meten et al. describes a way to implement this feature using a structure very similar to the branch detection feature. A table similar to the HSD memory is filled with branch addresses of branches in the detected hot spot. A counter similar to the HSC is decremented by D each time a non hot spot branch is executed and incremented by I each time a hot spot branch is executed. If the counter reaches zero, an interrupt is raised. The counter will only reach zero if enough non hot spot branches are executed. The calculations to determine design parameters I, D and counter sizes described in section 6.1 can be used in a similar way to determine the parameters for the monitoring component.

Because the host CPU is available at all time to process the output of the HSD, the monitoring feature is not strictly necessary in the proposed setup. The feature was set aside as future work, and implementation time was spent elsewhere.

CHAPTER 7

Host CPU Software

This section describes the software that runs on the host CPU. The software includes initialization code and hot spot optimization code.

7.1 m68k Instruction Decoder

In order to decode m68k instructions, the host CPU software relies on a modified version of the decoder found in the Unified Amiga Emulator which is used in the Hatari project [Hat13].

The decoder uses a textual description of decode information. The representation describes how bit patterns are mapped to instruction mnemonics and how operands for the instructions are encoded. The textual representation also details how condition flags are affected by each instruction. The textual representation is parsed by an instruction decoder builder. This software builds a table of 2^{16} entries. The first 16 bits of an m68k instruction are used as an index into this table. The result is a pointer to a structure that uniquely describes the instruction being decoded, how many extension words it uses, and what the meaning of the extension words are. The UAE software goes on to use this information to build an emulator for the m68k architecture. However, the

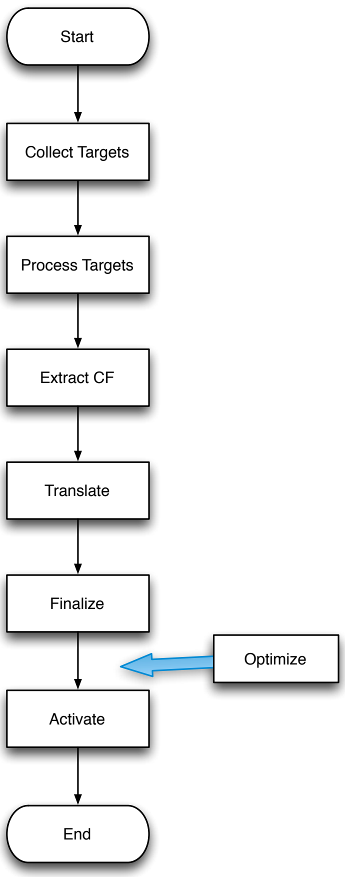

Figure 7.1: Overview of the optimization process.

host CPU software of this project only uses the decode table constructed by the UAE software.

7.2 Optimization Process

When the host CPU is notified of a hot spot detection by the execution engine, the hot spot optimization process is activated. The major phases of the process is depicted in figure 7.1.

The first phase of the optimization process collects targets of hot branches found by the HSD. These targets are processed into an internal representation of basic blocks by *Process Targets*. *Extract CF* connects these basic blocks into a control flow graph, CFG. *Translate* performs a translation that maps the decoded m68k instructions to a symbolic representation of internal VLIW instructions. *Finalize* patches the translated CFG with proper exits, so that control can enter and leave the translated code segment. *Activate* performs the necessary actions to notify the execution engine of the existence of the newly translated code.

The initial implementation does not perform any optimization. It does however achieve a much better instruction cache utilization than the hardware translator is able to achieve. After translation, the translated code is kept in a symbolic form with the intention of performing peephole optimization on the code. Such an optimization pass would be inserted after the *Finalize* pass, where it may capture most optimization opportunities.

An example of a pattern that might be targeted by a peephole optimizer is given below. Consider the following m68k instruction sequence:

```
add  d0,  d1
move d2, (a2)
```

If we represent the VLIW instructions as `<dalu> | <aalu>` then the result of the translation to internal VLIW instructions can be represented as:

```
add  d0,  d1   | nop
move d2, (a2) | nop
```

The peephole optimizer could change this sequence by moving the second instruction to the empty slot in the first instruction:

```
add  d0,  d1 | move d2, (a2)
```

This gives an excellent example of how the address ALU may be utilized to off-load the data ALU.

7.2.1 Collect Targets

When the optimization process is started, hot spot information has been dumped to main memroy by the HSD Controller. The dumped hot spot information contain addresses of hot branches and branch afinity information. The *Collect Targets* process decodes the branch instruction at the specified address and pushes the branch targets on a queue structure along with the afinity information.

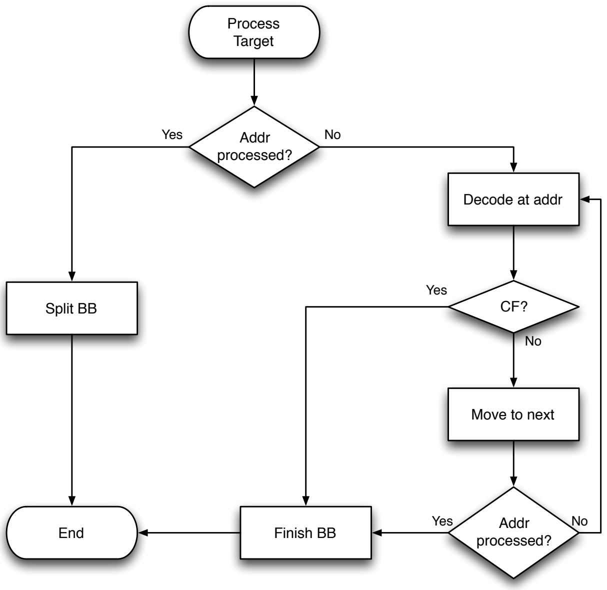

Figure 7.2: Overview of the *Process Targets* process. This process decodes m68k instructions.

7.2.2 Process Targets

When the targets have been collected and pushed to an internal work queue, the *Process Targets* process creates an internal structure representing a basic block for each target address. The flow of the process is sketched in figure 7.2. The process decodes instructions starting at the target address in a linear fashion. If the decoding point reaches a control flow instruction, or an address that has already been decoded, the current basic block is finished and pushed on a result queue.

If the target to be processed is contained in a basic block that has already been processed, then this basic block is split into two blocks at the target. This process is depicted in figure 7.3. Figure 7.3a shows the input blocks, the control flow unknown to the process. The process starts by decoding the instructions in the *Init* block in a linear fashion (7.3b) from the beginning of the block. The decoding continues until the control flow instruction at the end of the *body* block is decoded (7.3c). The target of the control flow instruction is internal to a block that has already been processed, namely the currently processed block, which contains both the *init* and *body* blocks. The block is split at the target of

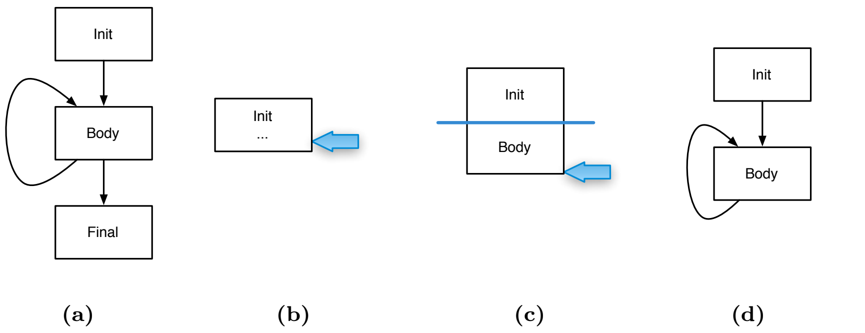

Figure 7.3: Decoding a block and splitting it in two. The blue arrow points to the instruction being decoded.

the control flow instruction, resulting in the correct control flow graph (7.3d).

7.2.3 Extract Control Flow

Figure 7.4 gives an overview of the *Extract Control Flow* process. The purpose of the process is to detect control flow edges between blocks that were decoded in the *Process Targets* process. Before the process starts, all the decoded blocks are pushed on a queue. The process takes blocks off this queue until it is empty.

For each block, the fall through address of the block is checked against the list of block start addresses. If a match is found, the fall through successor of the block is set to be the block that starts at the given address. If the block ends with a control flow instruction (finalizer), the branch target of this instruction is checked. If the instruction is an indirect jump, no further action is taken. If the target is a direct jump where the target can be calculated, the list of decoded address ranges is checked. If a range is found that contains the jump target, the taken successor of the block being processed is set to that block. If the jump target is not the beginning of the target block, the block is split at the target address to correctly reflect the discovered control flow.

7.2.4 Translate

The translation of instructions from m68k instructions to internal VLIW uses a straight forward pattern matching algorithm. Specific m68k instructions are always translated to the same sequence of internal VLIW instructions.

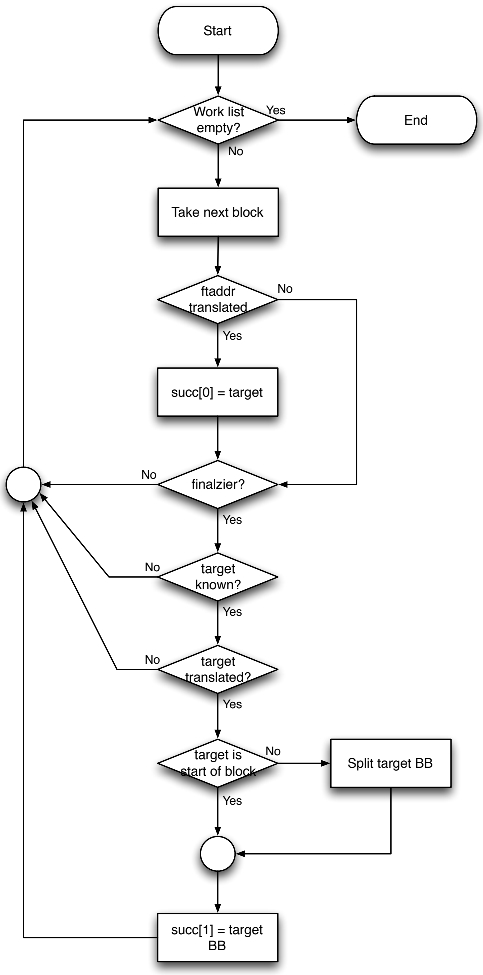

Figure 7.4: Overview of the control flow extraction pass.

A better result might be achieved by translating the instructions to an intermediate form before converting them to internal VLIW instructions. The intermediate form of the code might expose optimization opportunities not otherwise available. The intermediate form would be lowered to intermediate VLIW instructions by an instruction selection pass such as peephole selection [TC11] or a DAG based selection algorithm [KG08, Ert99]. This would also open up for other optimization passes on the intermediate code.

7.2.5 Finalize

Figure 7.5 gives an overview of the *Finalize* process. The purpose of the process is to insert control flow instructions at the end of the translated basic blocks and to select an ordering for the blocks to be emitted back to main memory. The process takes blocks from a queue until this queue is empty. The blocks are appended to an emit queue in the order which they should be laid out in memory.

If the block from which the translated block originates (original block) has a final control flow instruction (finalizer), but the target of the finalizer is unknown, the translated block is finalized by translating the original finalizer to an indirect jump.

If the finalizer is a direct jump and the target is a translated block, then the translated block is finished with a branch to this translated block. Otherwise the finalizer is translated to a branch to the original code. If the finalizer is conditional, care is taken to keep the semantics of the condition in the translated instructions.

After the taken successor of the block has been processed, the fall through successor is processed. If the fall through target of the block is not translated, an unconditional branch instruction to the original code is appended to the translated block. Otherwise, if the address of the translated block is translated, but not yet queued for emission, it is moved to the front of the work list. This will cause the block to be emitted after the current block in memory. If the block of the through address is already queued for emission, an unconditional branch to this block is inserted.

The described process does not take branch affinity into consideration when laying out basic blocks. An interesting optimization is to lay out the blocks of the mostly taken path as straight line code. To achieve this, the conditional finals need to be inverted if the taken target is the most frequent target.

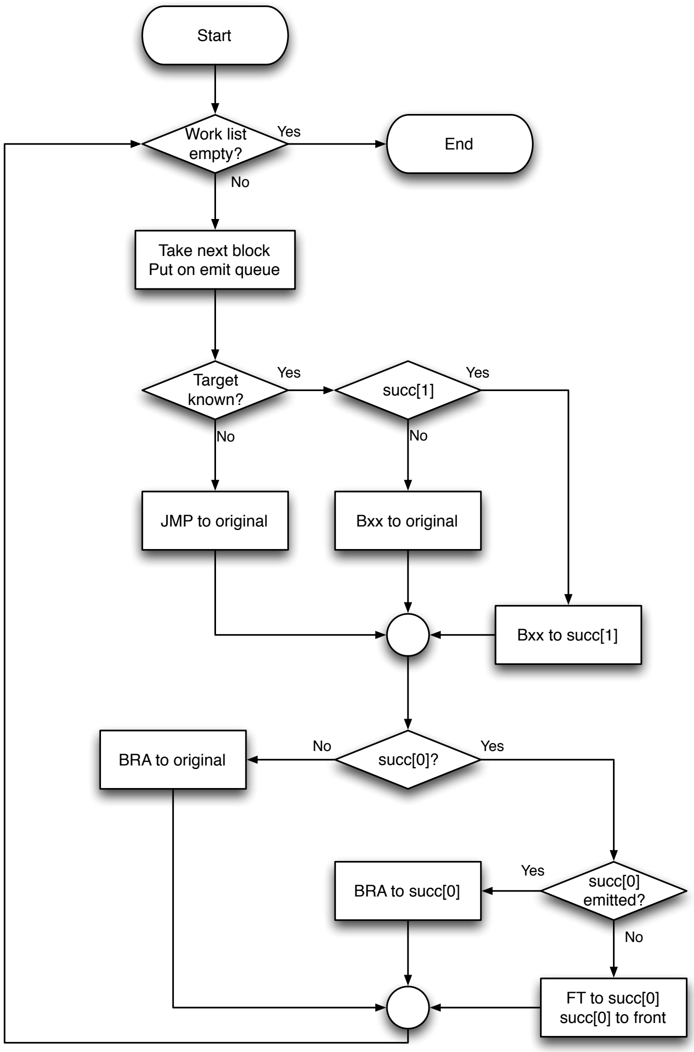

Figure 7.5: Overview of the finalize pass.

Chapter 8

Simulation

In order to verify a system like the one produced in this project, a proper simulation platform is required. Once the VHDL model is synthesized and loaded into actual hardware, it becomes very difficult to determine the source of errors in the model. The presence of errors can be verified easily by setting up proper tests, but to locate the source of the problems is another story.

Matters are further complicated by the fact that the system to be tested contains RTL logic described by VHDL code and host CPU software written in C code. One way to handle such a situation is by simulating the VHDL model and running the C code on the host CPU. Xilinx supports such a feature in their development tools. The feature is called HIL, short for *Hardware In the Loop*. However, Xilinx do not support HIL for the ZedBoard, which is the platform available for testing. Only the more expensive development kits manufactured by Xilinx currently support HIL.

In order to effectively track down bugs in the system, the GHDL HDL simulator was utilized. GHDL is a HDL simulator that is able to compile a VHDL model to a native x86 executable [GHD13]. To simulate the VHDL model, the native executable produced by GHDL is executed. The executable is capable of producing a value change file in the `ghw` format. The ghw file contains information on signal changes during simulation of the VHDL model. The file can be viewed in a viewer such as GTKWave [GTK13].

GHDL is implemented as a front end to GCC. It uses GCC to generate executable code that links with the GHDL run-time library, which is written in ADA. GHDL has a fairly undocumented feature that allows a VHDL procedure to be implemented in a foreign language. To activate this feature, the name of the foreign language procedure is given in an attribute for the VHDL procedure, and the procedure must be declared as having foreign linkage:

```
attribute foreign of <proc_name> : procedure is "VHPIDIRECT␣<↩
    foreign_link_name>";
```

This causes the procedure `proc_name` to have external linkage. Technically, GHDL produces an executable file where `proc_name` is an undefined symbol. During the link step of the compilation, an external object file or library must supply the location of the symbol. This corresponds to having undefined symbols in a regular C object file.

The interface that GHDL uses when calling externally supplied functions is not easily deduced from the sparse GHDL documentation. GHDL uses the same mechanics when linking an external C function to a VHDL procedure as GCC uses when linking ADA programs with C functions. However, it is not clear from the documentation what ADA data types are passed to the called external C function. By reading the GHDL ADA code and using a combination of trial and error, it was possible to construct a working interface between the GHDL compiled VHDL model and the GCC compiled C code.

This interface is used to construct a simulation environment where the VHDL model can be simulated along with the host CPU software. AXI master and slave interfaces were implemented in part VHDL and part C code.

In order to run the host CPU software simultaneously with the simulator, the GHDL entry point is wrapped in a custom initialization routine written in C. This routine takes care of initialization tasks and spawns a POSIX thread for the host CPU software to run in. When the host CPU thread has been spawned, the simulation is started in the main thread.

8.1 Virtual AXI Master

Figure 8.1 shows how a virtual AXI master interface is coupled with the host CPU software during execution of a GHDL simulation. The host CPU software initiates AXI transactions using the C AXI Master API. The transactions are converted to messages that are passed over a thread safe queue, the *AXI Request Queue*, to the simulator thread. The transactions are dequeued by the *C AXI*

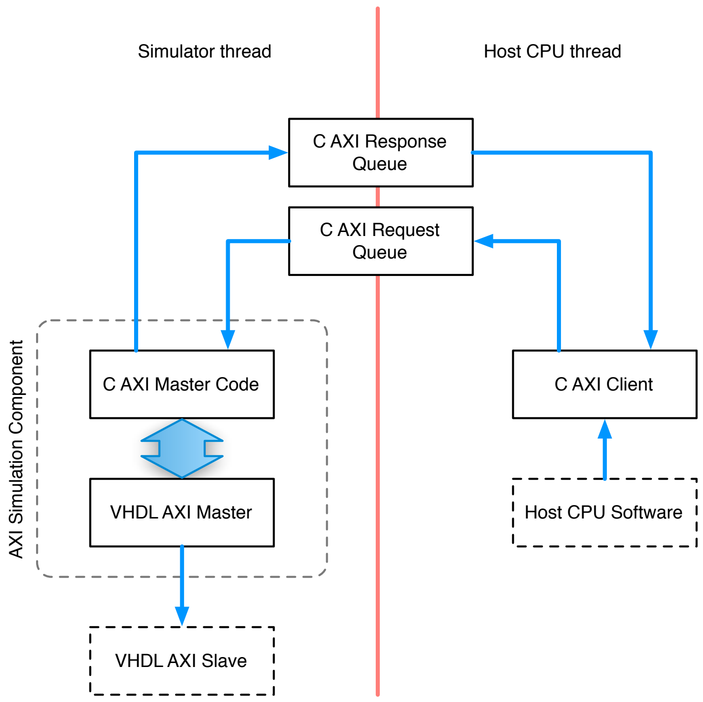

Figure 8.1: Overview of the virtual AXI master interface for the simulation environment.

Master Code in the simulator thread when invoked by the simulator. The *C AXI Master Code* is invoked by the simulator via the *VHDL AXI Master* component every time input signals to the procedure mapped to the C function is updated. Care is taken to only dequeue at most one message in each clock cycle, and only when the master interface is not currently processing another transaction. The VHDL AXI Master component communicates with its VHDL peer using valid AXI signaling. The response is pushed back to the host CPU thread via the *C AXI Response Queue.*

8.2 Virtual AXI Slave

A similar system has been constructed for providing a virtual AXI Slave interface, as depicted by figure 8.2. In the slave setup, the virtual AXI slave interface simulates slave interfaces to a shared block of memory. AXI requests originating in the simulator thread are routed over the thread boundary via the thread safe queue. In the host CPU thread they are converted to reads and writes to a block of memory.

8.3 Interrupt Delivery

In addition to the AXI interfaces, an interrupt delivery component was also constructed. The component allows the simulated VHDL model to deliver interrupts to the host CPU thread. However, the interrupt delivery is not performed by interrupting the host CPU thread. The thread must check for interrupts by polling a shared variable. This choice of using a polling mechanism was taken in order to cut down implementation time. A better implementation could be achieved by using UNIX signal delivery to interrupt the host CPU thread.

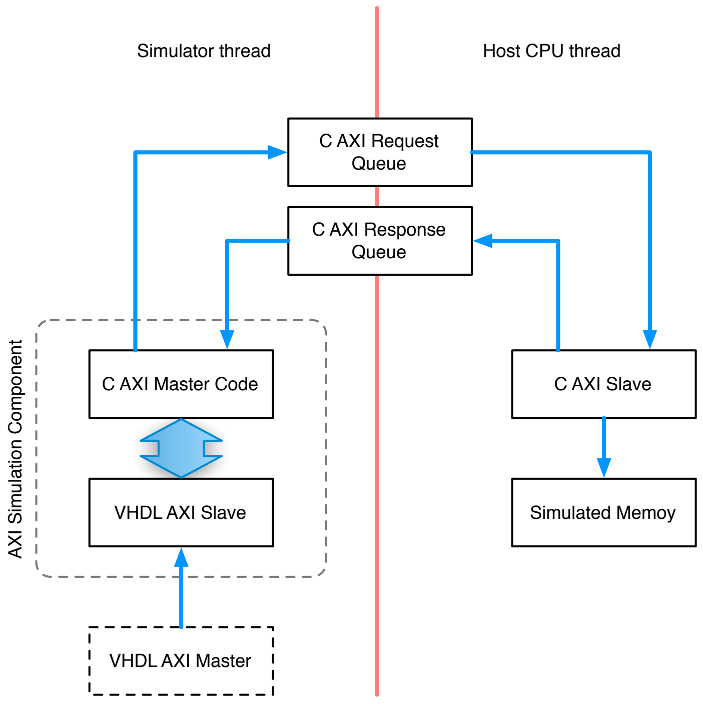

Figure 8.2: Overview of the virtual AXI slave interface.

Chapter 9

Results

When the project was kicked off, the intention was that this section should present the performance of a couple of benchmarks executed on the developed emulator, on a software interpreter, and a software binary translator. Unfortunately, this has not been possible. The system currently runs in the GHDL simulator. But because the simulator does not account for memory latency, and because the host CPU code is executed natively on the simulator host, running benchmarks on the simulator would provide misleading results. The following reasons also apply:

- The developed emulation system does not yet support enough instructions to run complete benchmarks.
- In order for the host CPU software to run on an ARM core in the Zynq device, some change to the software is required.
- Even though a significant amount of time has been spent on testing and verificaiton, the system probably still contains many bugs.

In the rest of this chapter, a theoretical absolute maximum performance for the emulation system is calculated. The calculations are based on synthesis results of the VHDL model and provides an upper bound for the performance of the system. The size of the synthesized logic and the size of the implementation is also discussed.

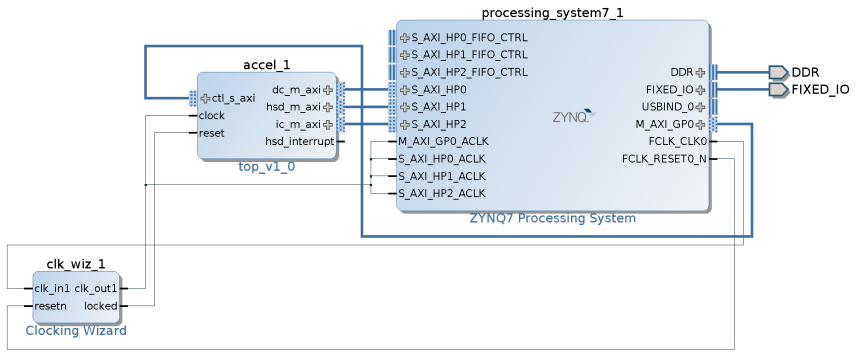

Figure 9.1: Using Vivado to connect the accelerator to the ARM cores.

Device	Speed Grade	Frequency (MHz)
XC7Z020	-1	120
XC7Z020	-3	156
XC7Z030	-3	237

Table 9.1: Implementation results.

9.1 Synthesis Results

The VHDL model was synthesized using Xilinx Vivado 2013.2. The processing system was configured using the default ZedBoard settings, but with varying FPGA device. The Processing System is used to generate a 100 MHz clock for a PLL in the FPGA fabric. The PLL synthesizes a clock that drives the AXI interfaces and the execution engine component. The setup is depicted in figure 9.1. The figure is a screen-shot from Vivado.

The Vivado tools are able to achieve timing closure using a 120 MHz clock frequency on a XC7Z020 speed grade -1 device and 156 MHz clock frequency on a XC7Z020 speed grade -3 device. On a XC7Z030 speed grade -3 device Vivado is able to achieve timing closure using a 237 MHz clock frequency.

In comparison, Xilinx reports that the Xilinx MicroBlaze RISC processor is capable of operating at 165 MHz on an unspecified speed grade -3 Zynq device [Xil13a]. MicroBlaze is a commercially available soft processor system, developed by Xilinx that is optimized for Xilinx FPGAs.

The Tinuso soft processor developed by Schleuniger et al. achieves 220 MHz on a Spartan 6 speed grade -3 device [SMK12]. Tinuso is a state of the art research processor system that aims to explore the limits of performance achievable for soft processors on FPGA devices.

It is possible that the performance of the execution engine could be increased by applying some of the design principles advocated by Schleuniger et al.. However, the instruction set of the VLIW engine is much wider and more complex than the instruction sets of both the MicroBlaze and the Tinuso core. This may be a limiting factor in achievable operating frequency.

The utilization reports for implementation in the XC7Z020 speed grade -1 device are given in figure 9.1 and 9.2. They show that only 4 block RAM primitives were used. Close investigation of the synthesis report has shown that these are used to implement data and instruction caches. The Vivado tool has chosen to implement the register files using CLB registers. This is probably because the register files are too small to spend an entire block ram. It is a bit of a mystery why LUT RAM was not used for the register file implementation. LUT RAM should be useful to implement wide, shallow memories such as the register files. The tool could be forced to use the LUT RAM by directly instantiating the LUT primitives.

The memory array of the HSD is not implemented as a block RAM either. The reason for this is an implementation technical detail. The Vivado tool does not support inferring block RAM structures for the coding pattern that was used to describe the HSD memory.

Listing 9.1: Utilization of FPGA CLB resources.

```
+-------------------------+-------+-------+-----------+-------+
|        Site Type        |  Used | Loced | Available | Util% |
+-------------------------+-------+-------+-----------+-------+
| Slice LUTs*             |  7370 |     0 |     53200 | 13.85 |
|   LUT as Logic          |  7370 |     0 |     53200 | 13.85 |
|   LUT as Memory         |     0 |     0 |     17400 |  0.00 |
| Slice Registers         | 10767 |     0 |    106400 | 10.11 |
|   Register as Flip Flop | 10624 |     0 |    106400 |  9.98 |
|   Register as Latch     |   143 |     0 |    106400 |  0.13 |
| F7 Muxes                |  1158 |     0 |     26600 |  4.35 |
| F8 Muxes                |   549 |     0 |     13300 |  4.12 |
+-------------------------+-------+-------+-----------+-------+
```

Listing 9.2: Utilization of FPGA memory resources.

```
+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Loced | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |    4 |     0 |       140 |  2.85 |
|   RAMB36/FIFO*    |    4 |     0 |       140 |  2.85 |
|     RAMB36E1 only |    4 |       |           |       |
|   RAMB18          |    0 |     0 |       280 |  0.00 |
+-------------------+------+-------+-----------+-------+
```

Even though a significant amount of time was spent on optimizing and pipelining the design, the optimization phase of the process was ended prematurely because other parts of the project required attention. It may be possible to raise the clock frequency further by spending some more time scrutinizing the design for inefficiencies and optimization possibilities.

One interesting observation that can be made from the utilization data is that it should be possible to implement a multi-processor system on a Zynq 7020 device. The implementation utilizes less that 14% of the LUTs in the device. This number is not expected to increase much with the implementation of the remaining instructions. The reason for this is that no extra inputs to the ALU are required to implement the extra instructions, and thus no more LUTs should be required.

Figure 9.2 shows how the design was mapped to the FPGA fabric. The figure represents the entire area of the Zynq chip. The ARM cores reside in the upper left corner of the chip. Even though the design utilizes less than 14% of the chip, it is clear that the tool has spread the implementation over a large area. The tool might not be able to achieve the same clock frequency for the design, if it is constrained to use a smaller space.

9.2 Estimated Performance

An absolute best case performance estimate can be constructed for the execution engine using the operating frequency reported by Vivado.

The best performance is achieved when executing a long running loop that has enough ILP to keep the pipeline busy without stalling. The loop will be detected by the HSD and executed in optimized mode. If all these conditions are true, the execution engine will retire one instruction per clock cycle, resulting in a theoretical peak performance of 237 million instructions per second (MIPS) for

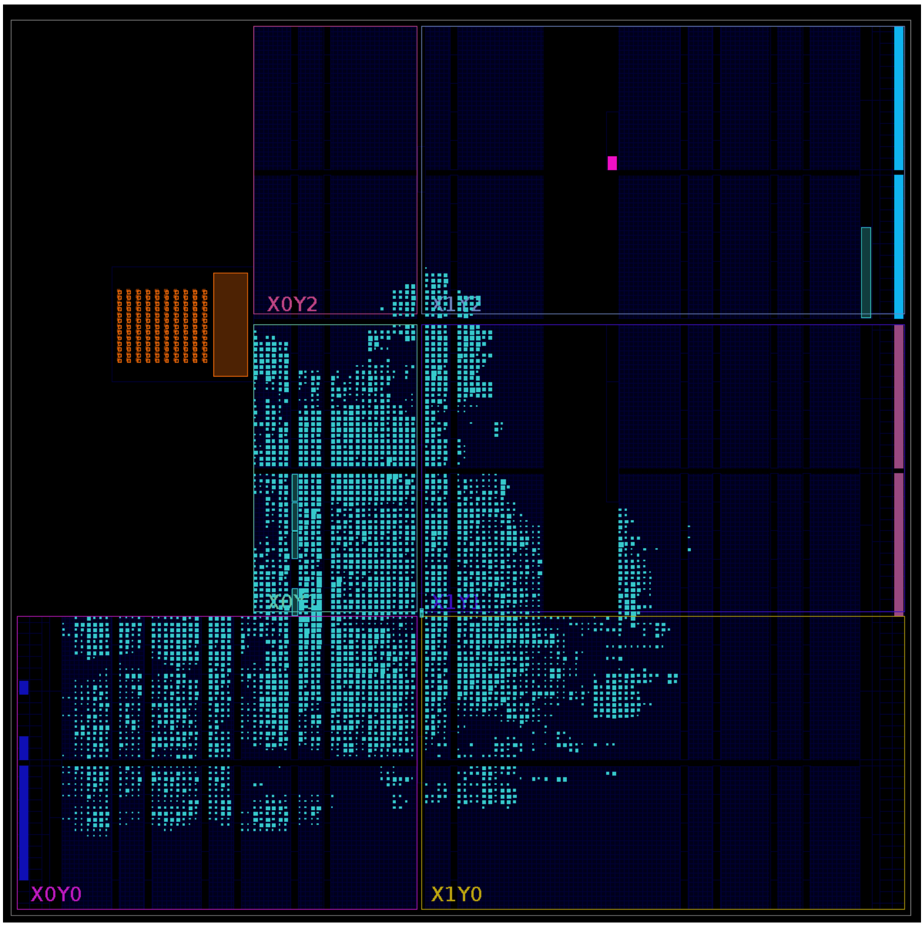

Figure 9.2: A graphical representation of the utilized CLBs on the Zynq device.

the XC7Z030 speed grade -3 device.

Because most m68k instructions are translated to a single VLIW instruction, this translates to 237 emulated MIPS theoretical peak performance, if the optimizer is unable to place multiple m68k instructions in a single VLIW instruction. If the optimizer is able to place more than one than one m68k instruction into a single VLIW instruction on average, the theoretical peak performance will be higher.

9.3 Complexity

In order to give an idea of where the complexity of the emulator is placed, table 9.2 provides a statistical breakdown of the size of all VHDL files in the project.

The file `combinatorics.vhd` implement most of the logic of the execution engine pipeline. Together `combinatorics.vhd` and `core.vhd` implement the entire execution engine pipeline, without the cache memories. These are factored out in the `icache.vhd` and `dcache.vhd` files. The instruction cache controller is the second largest file in the project. The reason that this file is so large is that it contains the simple dynamic translator that translates m68k instructions to internal VLIW instructions on the fly.

One interesting observation is that the HSD implementation in `hsd.vhd` is implemented in only 302 lines of code. This is only 3.4% of the total code size. The conclusion that follows from this observation is that the HSD as proposed in [MTG$^+$99] is simple to implement and easy to add to an existing design.

In addition the VHDL code, 3.3K lines of C code and 0.8K lines of Scala code was also written. The C code implements the hot spot translation software and the software part of the virtual AXI bus. The Scala code implements an assembler that was used to generate test cases for the pipeline before the translator was implemented.

File	Blank	Comment	Code
combinatorics.vhd	344	362	2079
ic_control.vhd	130	142	886
ghdl_top.vhd	90	84	631
types.vhd	76	59	544
inst_func.vhd	70	0	484
top.vhd	73	117	476
ic_decode.vhd	83	58	430
dcache.vhd	90	100	376
dc_control.vhd	59	29	303
hsd.vhd	55	26	302
ghdl_axi_sw_slave.vhd	30	9	277
hsd_axi.vhd	41	21	238
axi_ctl.vhd	27	5	233
ghdl_axi_sw_master.vhd	35	16	211
core.vhd	55	36	207
tb_dc_control.vhd	23	8	205
core_pkg.vhd	17	9	119
dcache_tag.vhd	16	24	100
tb_icache.vhd	25	5	100
icache_altera.vhd	8	2	89
icache.vhd	23	6	88
dcache_data.vhd	15	25	85
dcache_data_altera.vhd	10	4	67
dcache_tag_altera.vhd	10	4	63
tb_core.vhd	10	4	61
synth_tb.vhd	13	1	52
ghdl_intr.vhd	13	0	48
tb_top.vhd	9	0	43
func.vhd	7	0	17
SUM	1457	1156	8814

Table 9.2: Breakdown of the VHDL code.

Chapter 10

Future Work

This section lists tasks which would be interesting to complete, but which could not be completed within the time frame of the project. A number of interesting ideas for future research are also presented.

It was not possible to measure the performance of other emulators in time for the deadline. It would be interesting to see what level of performance that is required in order to have a *better* solution than what is available in software today.

The first point of a future work list for the project should should be to make the implementation feature complete by implementing missing instructions, making the necessary changes to the design to allow it to run on real hardware, and developing a series of automated tests.

In the current implementation, the hot spot regions are not optimized at all. Investigation of the effect of different optimization algorithms on the detected hot spots would be interesting future work. As mentioned in chapter 2, past research has shown that a significant increase of performance is attainable by optimizng hot progam regions dynamically.

Store-to-load forwarding was never implemented, even though a store buffer was added to the data cache. It would be interesting to see what performance gains are available by implementing store-to-load forwarding.

The instruction set for the VLIW processor was developed in a very ad-hoc manner. The result of this is that it contains some redundancies. Removal of these redundacies might decrease the size of the instructions a little bit.

No attention was given to the implementation of (precise) exceptions. A study of how to implement m68k exceptions in the emulation system is required in order to make the emulator useful.

Support for speculative execution and delivery of exceptions under speculative execution could also be interesting.

The emulator currently have no way of detecting self-modifying code. In order to make the emulator practical, it has to be able to detect self-modifying code.

Originally, the implementation was planned to have memory protection and TLB caches. A return address stack and a length prediction feature was also planned. These were not implemented due to lack of time.

Because the execution engine VHDL model is not able to operate at more than 120 MHz on a low end device, it might be feasible to apply some of the techniques presented in section 3.3.1 to create more write ports in the caches and register files. The block RAM primitives of the XC7Z020 device is capable of operating at speeds of up to 388 MHz for the speed grade -1 device [Xil12]. Implementing a double pumped block RAM structure should not be a problem with such block RAM operating frequencies.

It might also be possible to add a *Software Managed Cache*, SMC, to the pipeline. The SMC could be implemented as a couple of block RAM structures. Special instructions could be used to move content to and from the SMC. The optimizer might be able to utilize such a structure to avoid some memory operation delays.

Currently, the execution engine handles all forwarding and hazard detection in hardware. It would be interesting to see if the pipeline could be completely exposed to the optimizer. With an exposed pipeline, the optimizer would have to handle all predictable forwarding and hazard conditions. This might allow the frequency of the pipeline to be increased while operating in optimized mode. However, changing the operating frequency of a circuit that is operating might be quite difficult.

Chapter 11

Conclusion

The original goal of the project was to emulate the m68k ISA on a custom FPGA based execution engine using binary translation. While this goal has not been completely satisfied, many components of the proposed emulation system were implemented and tested. It is possible to emulate a subset of the m68k ISA using the solution implemented during the project.

During the project, a VLIW execution engine targeting FPGA implementation was implemented. The execution engine is capable of operating at a clock frequency of 237 MHz and has moderate resource utilization. The execution engine is the target for a simple hardware-based binary translation engine. The translation engine translates m68k instructions to the ISA of the VLIW.

By using an implementation of a hot spot detection mechanism proposed by Merten et al. [MTG+99], hot regions in the emulated m68k code can be detected. The hot spot detection mechanism has proved simple to implement and operate.

Separate data and instruction caches have been implemented in order for the VLIW execution engine to efficiently interface a memory subsystem. The caches use the AXI bus standard to interface a memory system.

A binary translation software component capable of retranslating hot regions of m68k code was also implemented. The translation system reads output produced by the hot spot detection mechanism and uses this information to drive the translation process. The translation system emits the translated code to system memory and notifies the VLIW execution engine of the existence of the translations. This causes the VLIW engine to execute the translated regions the next time it encounters a branch target for which there exists translation.

A system simulation tool based on GHDL that allows external software to communicate with a simulated VHDL model via emulated AXI transactions was also developed. This system allows software and VHDL hardware model to be co-simulated in a regular PC. The system was used for testing and verification of the implemented emulator system.

Based on the maximum frequency of the VLIW execution engine, the performance of the system is bounded from above at 237 million emulated instructions per second. With the implementation of one or more optimization passes in the translation software, this bound may be moved yet higher.

In order for the project goal to be considered completely satisfied, the implementation of more instructions in the VLIW engine and in the binary translator is required. The reason that these milestones were not reached is an underestimation of the implementation time required to implement these features.

Appendix A

Project Management

Aspects of the Agile Unified Process was used as a basis for managing the project. A project document was produced during the start-up phase of the project. The document defines the scope of the project, a schedule with relevant milestones and a section on risk management. The risk management section was a major help in identifying early risk factors. Based on the risk analysis, all identified problems have been avoided.

Because of external events, the project was not kicked off at the initially planned date. When the project was resumed, the scope was refined, but the milestone dates were not updated, and thus do not reflect the actual project progress.

The initial project management document is available upon request.

Index

Bibliography

[Alt13] Altera. *Cyclone V Device Handbook*, June 2013.

[ARM11] ARM. *AMBA®AXI^{TM}and ACE^{TM}Protocol Specification*, 2011.

[AVN13] AVNET. *ZedBoard Hardware User's Guide*, 1.9 edition, Jan 2013.

[BA97] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25:13–25, June 1997.

[BDB99] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical report, Technical Report HPL-1999-78, Hewlett-Packard Laboratories, 1999.

[BRS99] Bryan Black, Bohuslav Rychlik, and John Paul Shen. The block-based trace cache. In *ACM SIGARCH Computer Architecture News*, volume 27, pages 196–207. IEEE Computer Society, 1999.

[BV00] Stephen D Brown and Zvonko G Vranesic. *Fundamentals of digital logic with VHDL design.* McGraw-Hill New York, 2000.

[CK94] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. *Joint International Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[CS00] Yuan Chou and John Paul Shen. Instruction path coprocessors. *ACM SIGARCH Computer Architecture News*, 28(2):270–281, 2000.

[DGB$^+$03] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM} software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 15–24. IEEE Computer Society, 2003.

[EAGS01] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *Computers, IEEE Transactions on*, 50(6):529–548, 2001.

[Ert99] M Anton Ertl. Optimal code selection in dags. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–249. ACM, 1999.

[FBC$^+$01a] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel, and Steven S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pages 16–27, 2001.

[FBC$^+$01b] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay J Patel, and Steven S Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 16–27. IEEE Computer Society, 2001.

[FMS$^+$04a] Brian Fahs, Aqeel Mahesri, Francesco Spadini, Sanjay J. Patel, and Steven S. Lumetta. The performance potential of trace-based dynamic optimization. Technical report, Center for Reliable and High-Performance Computing, University of Illinois, 1308 West Main Street 1308 West Main Street, Urbana, IL 61801 USA, November 2004.

[FMS$^+$04b] Brian Fahs, Aqeel Mahesri, Francesco Spadini, Sanjay J Patel, and Steven S Lumetta. The performance potential of trace-based dynamic optimization. Technical Report UILU-ENG-04-2208, University of Illinois, 2004.

[FPP98] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 173–181. IEEE Computer Society Press, 1998.

[Fre93] Freescale Semiconductor, Inc. *M68000 8-/16-/32-Bit Microprocessors User's Manual*, ninth edition edition, 1993.

[GA01] Michael Gschwind and Erik Altman. Optimization and precise exceptions in dynamic compilation. *ACM SIGARCH Computer Architecture News*, 29(1):66–74, 2001.

[GHD13] Ghdl simulator. `http://ghdl.free.fr`, August 2013.

[GTK13] Gtkwave home page. `http://gtkwave.sourceforge.net`, August 2013.

[Hat13] Hatari home page. `http://hatari.tuxfamily.org`, August 2013.

[HP12] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[Jed12] Jedec. *DDR3 SDRAM Standard*. JEDEC SOLID STATE TECHNOLOGY ASSOCIATION, jesd79-3f edition, July 2012.

[JS99] Quinn Jacobson and James E Smith. Instruction pre-processing in trace processors. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 125–129. IEEE, 1999.

[K$^+$00] Alexander Klaiber et al. The technology behind crusoe processors. *Transmeta Technical Brief*, 2000.

[KA01] Rajeev Krishna and Todd Austin. Efficient software decoder design. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, 2001.

[KG08] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on dags. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 45–54, New York, NY, USA, 2008. ACM.

[LS10] Charles Eric LaForest and J Gregory Steffan. Efficient multi-ported memories for fpgas. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 41–50. ACM, 2010.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification. `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`, Feb 2013.

[McF93] Scott McFarling. Combining branch predictors. Technical report, Western Research Laboratory, 250 University Avenue, Palo Alto, California 94301 USA, June 1993.

[MCH+92] Scott A Mahlke, William Y Chen, Wen-Mei W Hwu, B Ramakrishna Rau, and Michael S Schlansker. Sentinel scheduling for vliw and superscalar processors. In *ACM SIGPLAN Notices*, volume 27, pages 238–247. ACM, 1992.

[Mot92] Motorola, Inc. *MOTOROLA M68000 FAMILY Programmer's Reference Manual*, rev. 1 edition, 1992.

[MS94] Peter Magnusson and David Samuelsson. A compact intermediate format for simics. Swedish Institute of Computer Science, September 1994.

[MS08] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *Proceedings of the Workshop on Architectural and Microarchitectural Support for Binary Translation*, pages 55–70, 2008.

[MTB+01] Matthew C Merten, Andrew R Trick, Ronald D Barnes, Erik M Nystrom, Christopher N George, John C Gyllenhaal, and Wen-mei W Hwu. An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–589, 2001.

[MTG+99] Matthew C Merten, Andrew R Trick, Christopher N George, John C Gyllenhaal, and Wen-mei W Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 136–147, 1999.

[MTN+00] Matthew C Merten, Andrew R Trick, Erik M Nystrom, Ronald D Barnes, and Wen-mei W Hwu. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots. In *Proceedings of the 27th annual international symposium on Computer architecture - ISCA '00*, pages 59–70, 2000.

[NBMH01] Erik M Nystrom, Ronald D Barnes, Matthew C Merten, and Wen-mei W Hwu. Code reordering and speculation support for dynamic optimization systems. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 163–174. IEEE, 2001.

[PFP99] Sanjay Jeram Patel, Daniel Holmes Friendly, and Yale N Patt. Evaluation of design options for the trace cache fetch mechanism. *Computers, IEEE Transactions on*, 48(2):193–204, 1999.

[PH08] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface.* Morgan Kaufmann, 2008.

[POV03] Il Park, Chong Liang Ooi, and TN Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 411. IEEE Computer Society, 2003.

[PTBC00] Sanjay J Patel, Tony Tung, Satarupa Bose, and Matthew M Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 303–313. IEEE, 2000.

[RBS99] Eric Rotenberg, Steve Bennett, and James E Smith. A trace cache microarchitecture and evaluation. *Computers, IEEE Transactions on*, 48(2):111–120, 1999.

[SCF$^+$03] Brian Slechta, David Crowe, N Fahs, Michael Fertig, Gregory Muthler, Justin Quek, Francesco Spadini, Sanjay J Patel, and Steven S Lumetta. Dynamic optimization of micro-operations. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 165–176. IEEE, 2003.

[SMK12] P. Schleuniger, S. A. McKee, and S. Karlsson. Design principles for synthesizable processor cores. In *Proceedings of the 25th International Conference on Architecture of Computing Systems ARCS*, 2012.

[TC11] Linda Torczon and Keith Cooper. *Engineering A Compiler.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[TM11] Jens Tröger and Darek Mihocka. Fast microcode interpretation with transactional commit/abort. 4th Workshop on Architectural and Microarchitectural Support for Binary Translation, June 2011.

[vR13] Guido van Rossum. The python language reference - release 2.7.5. `http://docs.python.org/2/archives/python-2.7.5-docs-pdf-a4.tar.bz2`, August 2013.

[Xil12] Xilinx. *7 Series FPGAs Memory Resources User Guide (UG473)*, v1.7 edition, October 2012.

[Xil13a] Xilinx. Microblaze soft processor core. `http://www.xilinx.com/tools/microblaze.htm`, August 2013.

[Xil13b] Xilinx. *Zynq-7000 All Programmable SoC Overview (DS190)*, v1.4 edition, August 2013.

[Xil13c] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*, 1.5 edition, March 2013.