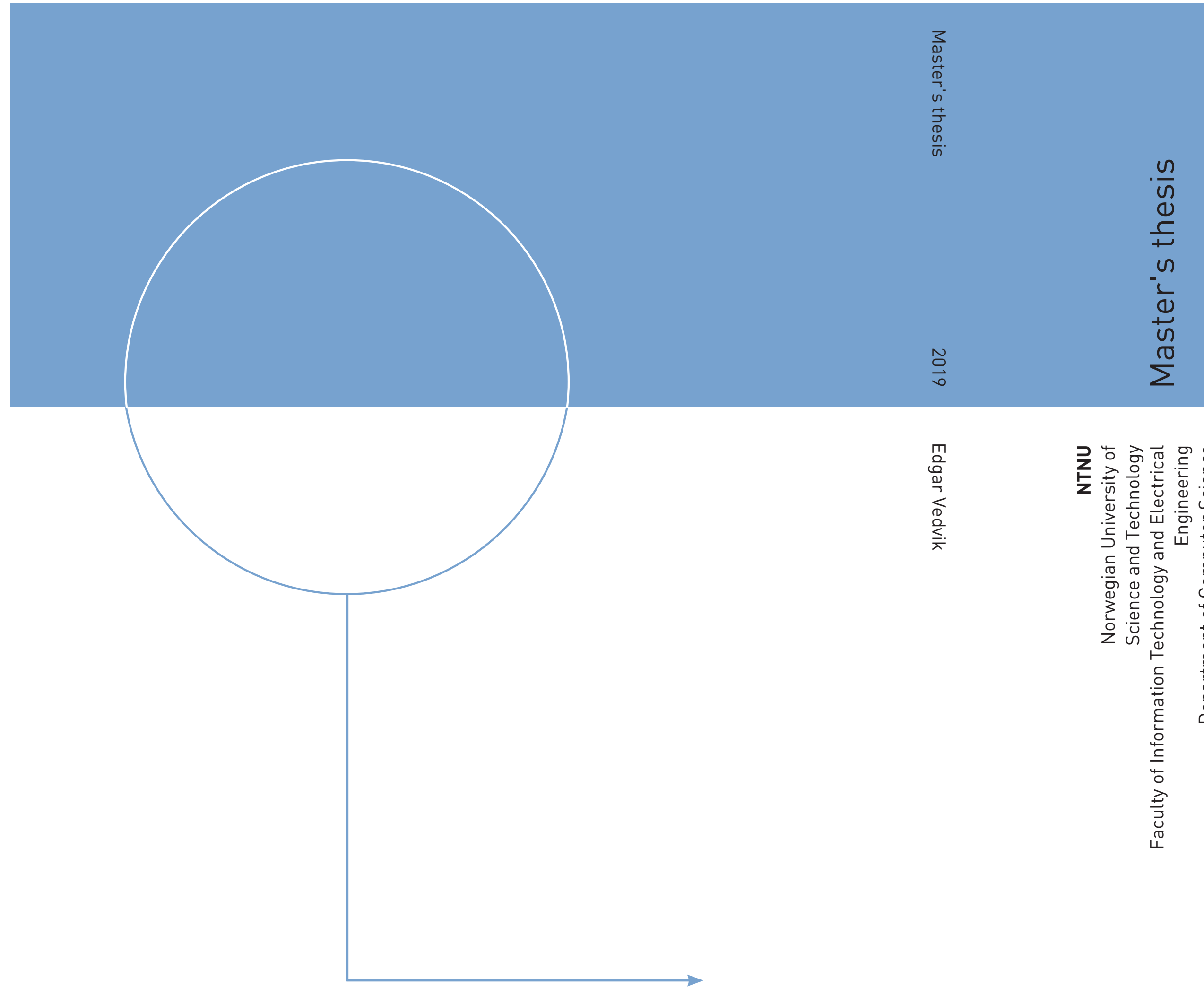


Edgar Vedvik

Implementing Data Cache Access Memoization (DCAM) in hardware to measure L1 DC and DTLB energy efficiency

June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

Implementing Data Cache Access Memoization (DCAM) in hardware to measure L1 DC and DTLB energy efficiency

Edgar Vedvik

MIDT
Submission date: June 2019
Supervisor: Magnus Själander

Norwegian University of Science and Technology
Department of Computer Science

Abstract

The level-1 data cache (L1 DC) and data translation lookaside buffer (DTLB) are essential in contemporary memory hierarchies by providing faster data access and reducing the number of stall cycles in processors. Accesses to these structures are common and they use significantly more energy than registers. A large portion of a processors energy budget is spent servicing data through the L1 DC and DTLB. Stokes et al. (2019) recently proposed the data cache access memoisation (DCAM) technique to reduce energy usage by the L1 DC and DTLB. We will implement this technique in VHDL and test it on an FPGA. We will also investigate the performance, energy usage and critical path of the technique. DCAM identifies the last instruction to update a register before it is referenced by a memory instruction. By performing the tag check along with this prepare to access memory (PAM) instruction, we are able to access a single data array in a set associative cache. By memoising this information between instructions, we are able to reduce the number of DTLB accesses and L1 DC tag checks. We show an implementation of the DCAM-technique that does not increase the critical path and uses significantly less power than a standard pipeline.

Sammendrag

Nivå-1 data-hurtiglager (L1 DC) og mellomlager for dataoversetting (DTLB) er essensielle i nåtidens minnehierarki og å gi raskere tilgang til data og redusere antall ventesykluser. Disse strukturene bli aksessert ofte, og bruker betydelig mer energi enn prosessorregistrer. En stor del av prosessorens energibudsjett går med til å betjene data gjennom nivå-1 hurtiglageret og dataoversettingsmellomlageret. (Stokes et al., 2019) foreslo nylig «data cache access memoisation» (DCAM), som er en teknikk for å redusere energiforbruket i disse strukturene. Vi vil utforske ytelsen, energiforbruket og den kristiske stien til DCAM-teknikken og se hvordan den sammenligner med en standard implementasjon. DCAM-teknikken identifiserer den siste instruksjonen som oppdaterer et register som senere blir brukt av en minneinstruksjon. Ved å utføre tagg-sjekken sammen med instruksjonen som oppdaterte registeret sist, kan vi aksessere kun én datatabell i et sett-assosiativt hurtiglager. Ved å memoisere denne informasjonen mellom instruksjoner er vi i stand til å redusere antall DTLB-aksesser og L1 DC-tagg-sjekker. Vi viser at en implementasjon av denne teknikken ikke forlenger den kristiske stien, og bruker betydelig mindre kraft enn en standard implementasjon.

Preface

I would like to thank my supervisor Magnus Själander for his insights and guidance with this project, and for introducing me to the world of energy efficient caching and helping me find related research. I would also like to thank Michael Stokes, Ryan Baird, Zhaoxiang Jin, David Whalley and Soner Onder for their research which this thesis is based upon.

Table of Contents

Abstract **i**

Sammendrag **ii**

Preface **iii**

Table of Contents **vi**

List of Tables **vii**

List of Figures **ix**

Abbreviations **x**

1 Introduction **1**

2 Related work **3**

3 Background **5**

3.1 MIPS I ISA 5

3.1.1 Instruction formats 6

3.1.2 Loads and stores 6

3.1.3 ALU 7

3.1.4 Jumps and branches 7

3.2 Caching 8

3.2.1 Set associative caches 9

3.2.2 Replacement policies 9

3.3 Data translation lookaside buffer 11

4 Baseline implementation **13**

4.1 Hazard detection and forwarding 13

4.2 Instruction Decode 14

4.3 Execute 15
4.4 Memory hierarchy 15
4.5 DTLB 16
4.6 L1 Cache 16

5 PAM instructions and pipeline changes **19**

6 Improved pipeline **21**
6.1 DCAS 21
6.1.1 Updating DCAS 23
6.2 DCAV 24
6.3 Pipeline example 24

7 Methodology and results **27**
7.1 Resource usage 27
7.2 Critical path 29
7.3 Power estimation 32
7.4 Reduced DTLB and L1 DC tag checks 33

8 Discussion **35**

9 Conclusions **37**

Bibliography **39**

List of Tables

4.1 Memory configurations 16

5.1 Last instruction to compute data address 19
5.2 Pipeline stages 20
5.3 Pipeline stages used by various instructions 20

6.1 DCAS pipeline example 25

7.1 Baseline pipeline resource usage. 28
7.2 Improved pipeline resource usage. 30
7.3 Memory accesses. 33

List of Figures

1.1 Memory access micro-operations 2

3.1 The three types of instruction formats in MIPS. *Rs*, *rt* and *rd* are register addresses. *Sa* is the shift-amount used in shift-instructions and *funct* specifies the ALU-operation. 6
3.2 Address fields. 9
3.3 Conventional L1 DC pipelined access. 10
3.4 PLRUm sequence 10
3.5 Overview of the address translation process. The TLB is indexed with the virtual page number and yields the physical page number. The page offset is the same for both the physical and virtual page. In this figure the page size is 2^{12} bytes or 4 kB. Although each table entry here is 20 bits wide, there are usually extra bits stored for each entry, such as validity and page protection bits. 11

4.1 Simplified overview of the 5-stage MIPS processor. 14
4.2 Overview of the memory hierarchy 15
4.3 Implementation of the 4-way set associative cache with separate data arrays. 17

6.1 Simplified overview of the additions for the improved pipeline. 22
6.2 Memory access patterns 22
6.3 Data Cache Access Structure (DCAS) 23
6.4 Detecting cache line and page changes. 24
6.5 DCAS Valid Info (DCAV) 25

7.1 The blue lines indicate the critical path in the baseline implementation. . . 30
7.2 The blue lines indicate the additions for the improved pipeline. 31
7.3 Power consumption for both pipelines. 32
7.4 Power consumption for both pipelines with forced flip flops. 33

Abbreviations

ALU	=	Arithmetic-Logic Unit
BRAM	=	Block RAM
CPU	=	Central Processing Unit
DA	=	Data Access
DAS	=	Data Access Single
DCAM	=	Data Cache Access Memoisation
DCAS	=	Data Cache Access Structure
DCAV	=	DCAS Valid info
DSP	=	Digital Signal Processing
DTLB	=	Data Translation Lookaside Buffer
DWV	=	DTLB Way Valid
EX	=	Execute
FPGA	=	Field-Programmable Gate Array
FPU	=	Floating Point Unit
GPR	=	General Purpose Register
HDU	=	Hazard Detection Unit
ID	=	Instruction Decode
IF	=	Instruction Fetch
ISA	=	Instruction Set Architecture
LRU	=	Least Recently Used
LUT	=	Look-Up Table
LWV	=	L1 DC Way Valid
LWVN	=	L1 DC Way Valid Next
L1 DC	=	Level-1 Data Cache
L1 IC	=	Level-1 Instruction Cache
MEM	=	Memory
MRU	=	Most Recently Used
MUX	=	Multiplexer
OS	=	Operating System
PAM	=	Prepare to Access Memory
PC	=	Program Counter
PP	=	Page Protection
PLRU	=	Pseudo Least Recently Used
PLRUm	=	Pseudo Least Recently Used with MRU
RAM	=	Random Access Memory
RISC	=	Reduced Instruction Set Computer
TC	=	Tag Check
TLB	=	Translation Lookaside Buffer
VHDL	=	VHSIC Hardware Description Language
WB	=	Write Back

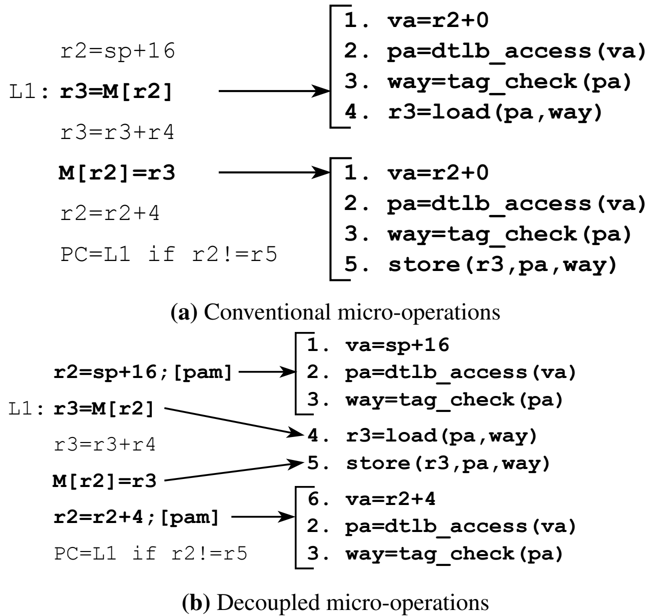
Chapter 1

Introduction

The demand for performance and the growth of computing devices in the world continues to increase. The environmental impact of these demands are huge and the need for energy efficiency is becoming increasingly important. Energy efficiency is also important for mobile devices and embedded processors with limited power supply capacity. The end of Dennard scaling has caused general-purpose processors to no longer experience the same improvements in energy efficiency as before (Johnsson and Netzer, 2016; Horowitz et al., 2005). This means clock rates and single-core performance in processors are now limited by the amount of heat they emit. By introducing more energy efficient solutions, more of the processor power budget can be spent on performance improvements (Huang et al., 2011).

Level-1 Data Caches (L1 DC) and Data Translation Lookaside Buffers (DTLB) are essential in today's memory hierarchy, providing both increased performance and energy efficiency compared to main memory access. However, L1 DC and DTLB access still uses significantly more power than register file access. Studies show that roughly 25% of an embedded processor's total power budget is spent on data accesses to the L1 DC and the DTLB (Dally et al., 2008; Hameed et al., 2010). Increasing the energy efficiency of data supply is then a reasonable target for improvement.

In reduced instruction set computers (RISC), most instructions are implemented using a single hardware micro-operation (μop). However, memory operations such as load and store are implemented using multiple μops by the hardware. These μops consume a significant amount of energy, and also forms dependency chains. Figure 1.1a shows some example code including a load and store with their μops highlighted. For a load instruction there are four μops the processor has to execute. First it has to generate the virtual address (*va*). This is usually done by adding the base register to an offset. Secondly, use the *va* to access the DTLB to obtain the physical address (*pa*). Thirdly, perform the tag check to determine which *way* the data resides in a set-associative cache. Finally, use the *pa* and *way* to index the data array to retrieve the data value. Store instructions perform the same first three μops as load instructions, but uses the *pa* and *way* to determine where to store a data value. If a load is followed by a store that accesses the same memory location (Figure



(a) Conventional micro-operations

(b) Decoupled micro-operations

Figure 1.1: Memory access micro-operations

1.1a), then the first three μops are redundant for the store instruction since the values of *pa* and *way* have not changed.

One way to remove this redundancy is to associate the first three μops with the instruction that updates the base register for the memory address (Stokes et al., 2018). This creates a prepare to access memory (PAM) instruction which can be seen in Figure 1.1b. From the programming perspective the instruction functions as before, it is only a change in how the hardware implements the μops. By doing this, results of the virtual address calculation, DTLB access and L1 DC tag are saved together with the register (*r2* in Figure 1.1), and can be reused by subsequent instructions.

Stokes et al. (2018) propose to alter the instruction set architecture (ISA) and make the compiler annotate the PAM instructions. However, it is also possible to dynamically detect PAM instructions in hardware. This approach does not require ISA modification and custom compilers and is the approach that will be used in this thesis. Additionally we will implement the Data Cache Access Memoisation (DCAM) (Stokes et al., 2019) technique in order to retain data access information calculated by PAM instructions.

In this thesis we will use VHDL to implement a baseline processor supporting the MIPS I instruction set, a DTLB and a 4-way set-associative cache to be used as the both the L1 DC and L1 instruction cache (L1 IC). We will then improve upon the baseline implementation and add the aforementioned PAM and DCAM techniques to it. Once we have both a baseline implementation and an improved version we will compare the energy efficiency, performance and resource usage between the two.

Chapter 2

Related work

There are many other techniques which aim to reduce the energy usage of data accesses. Many of the techniques require trade-offs which may limit their usage and some are compatible with the PAM / DCAM approach and can be combined to further reduce the data access energy usage.

The way-halting cache (WHC) is one such technique which aims to reduce L1 DC energy usage (Zhang et al., 2005). The WHC stores the low order tag bits in a fully associative cache called the halt tag array. On a data access, this small cache is used to perform a partial tag match against the low order bits of the address tag. Only the ways with a partial match will access their tag and data arrays. This halts access to the ways that cannot contain the data. An issue with the WHC is that it requires a custom SRAM implementation which is more costly and may reduce the maximum operating frequency.

A technique which aims to improve upon the WHC, is the speculative halt-tag access (SHA) approach (Moreau et al., 2016). This technique speculatively accesses the halt tag array in the address generation stage, and if speculation is successful, only the partially matching ways are accessed in the SRAM access stage. Because the halt tag array is accessed in the address generation stage, conventional SRAM implementations can be used. The speculation is only done when the magnitude of the displacement in the address calculation is small. If the displacement is small, there is a high probability that only the cache-line offset will change. For large displacements or on a speculation miss, the cache is accessed conventionally without any decrease in performance. Speculation is successful when there is no carry from the line offset to the line index for positive displacements. Conversely, if the displacement is negative, the speculation is successful when there is a carry. Both the original way-halting cache and the SHA approach can be combined with the PAM technique to reduce energy usage even further.

Tag check elision (TCE) is another technique which aims to reduce energy usage in set associative caches by avoiding tag checks (Zheng et al., 2014). TCE is similar to the PAM technique in that it stores an L1 DC way together with each register. However, TCE works in a very different way than PAM. In TCE, the cache way records (CWR) stores the cache line bounds and L1 DC way last used for each register. On a memory operation the L1 DC

way and cache line offsets that were accessed will be stored in the CWR for the register used as base address. When a register is modified, its record in the CWR is invalidated. When a cache line is evicted, all records in the CWR pointing to it must be invalidated to ensure correctness. This is done by storing a vector for each cache line, indicating which registers are pointing to it. If the base register in a memory operation has a valid record in the CWR, the stored bound will be compared with the address displacement. If the displacement is within the bounds only the stored way will be accessed, and the DTLB will be bypassed. If, however, there does not exist a valid record, or the bounds check fails, all ways are accessed in parallel as normal.

L1 DCs have to access their tag and data arrays in parallel in order to avoid stall cycles. However, if the subsequent instructions does not depend on the loaded data, energy is wasted. The early load data dependence detection (ELD^3) technique aims to save energy when there is no data dependency between the load instruction and the subsequent instructions (Bardizbanyan et al., 2014). If no dependency is detected, the tag and data arrays are accessed sequentially over two consecutive cycles. The result of the tag check can then be used to only access a single data array. If a dependency is detected, the tag and data arrays will be conventionally accessed in parallel. The dependency check is implemented by storing a bit for each instruction in the L1 instruction cache (IC). This bit indicates that an instruction is a load operation and if there are dependencies with the following three instructions. When a load instruction is committed to the pipeline, a check is performed to see if it has dependencies with the subsequent three instructions. Depending on the value of the bit, the tag and data arrays will either be accessed in parallel or in sequence over two stages. When a cache line is evicted, the dependency information might be incorrect. However, in-order pipelines have data dependency checks, which means at worst it will only introduce a single stall cycle. Cache line evictions are rare, and the instruction will be updated correctly the first time it is committed after an eviction.

Another technique uses an extended TLB (eTLB) to create a tag-less cache (TLC) (Sembrant et al., 2013). The eTLB contains extra information about the cache line locations. Each eTLB entry has information about every cache line belonging to that page. By storing this information in the eTLB, it is possible to avoid the normal tag check in caches. In order to invalidate a cache line in the eTLB, each cache line must store a back pointer to the page that it is contained in. This is because we no longer store a tag which identifies what page a cache line belongs to. The eTLB can be further improved by accommodating two different page sizes. This allows the eTLB to hold more sparse data by using micropages which only contain a few cache lines. The TLC can significantly reduce energy usage, however, their design assumes a phased cache, in which the eTLB is accessed first before accessing the data arrays. This allows for accessing a single data array, but at the cost of either increasing the cycle time or requiring an additional cycle for data access. The TLC also needs to deal with synonyms and other issues related to virtually addressed data access.

Chapter 3

Background

In order to test our energy efficiency improvements we have implemented a microprocessor supporting the MIPS I instruction set, with a few omissions. We have also implemented a memory management unit (MMU) containing a L1 DC, L1 IC and DTLB. In the following sections we will briefly explain the MIPS I instruction set architecture, the cache architecture and virtual to physical address translation process.

3.1 MIPS I ISA

MIPS is a reduced instruction set computer (RISC) ISA (MIPS Technologies, 2019). MIPS I is a 32 bit ISA, and is the first version of the MIPS architecture. It supported a total of 58 instructions, excluding coprocessor and floating point instructions. Newer versions of the MIPS architecture are still backwards compatible with these instructions (MIPS Technologies, 2016). MIPS is a load/store architecture, which means that only memory instructions are able to data between registers and memory, and arithmetic-logic unit (ALU) instructions can only operate on values in registers. This is in contrast to a register memory architecture where ALU operands and its result can both come from/be written to registers and memory. Using a load store architecture greatly simplifies the instruction set because of the limited number of ways an instruction can be used.

In MIPS I, every instruction is 32 bits wide, each general purpose register (GPR) is 32 bits wide and load/stores work on at most 32 bits at a time. This generalisation further simplifies the design of a MIPS processor. There are 32 GPR available, however, register $0 ($zero) and register $31 ($ra, return address) are special. $0 is hardwired to zero, and writes to it are discarded. Register $31 stores the current address on certain instructions to enable procedure calls. The other GPRs also have names and designated purposes, however they have no special properties in the architecture itself. In addition to the GPRs, there are two special registers called *HI* and *LO* which are used for multiplication and division. These special registers are also 32 bits wide.

MIPS also supports multiple optional coprocessors, such as a floating point unit (FPU). In the original MIPS I architecture, the first coprocessor (*CP0*) was the system control

(Price, 1995). This provided processor control, memory management and exception handling functions. If the system had a FPU, this would be second coprocessor (*CP1*). MIPS I supported four coprocessors, however the last two were never used. For this paper, coprocessors, system control and floating point arithmetic is not needed, and therefore not discussed further.

3.1.1 Instruction formats

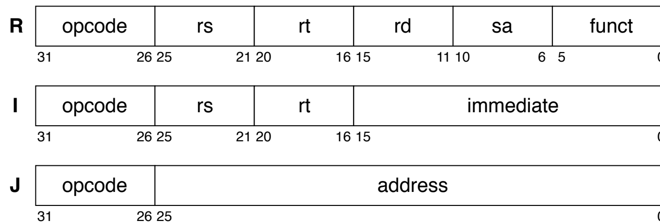


Figure 3.1: The three types of instruction formats in MIPS. *Rs*, *rt* and *rd* are register addresses. *Sa* is the shift-amount used in shift-instructions and *funct* specifies the ALU-operation.

As mentioned, instructions in MIPS ISA are always 32 bits wide and can only be formatted in three different ways as seen in Figure 3.1. Because of the low number of variations and the regular structure of the instruction types, it is possible to decode the instruction and read the required registers in parallel. R-type instructions are mostly used for ALU operations where both operands (*rs* and *rt*) come from registers and the result is stored in register *rd*. These instructions always have an opcode of 000000_2 to signify this, and their operation is instead specified in the *funct* field of the instruction. The J-type instruction is only used for jumps with a 26-bit jump address.

I-type instructions are more complex and are used in many different ways. It is used for ALU operations with immediate operands, loads and stores, and branches. In ALU operations, the first operand will be in *rs* and the second operand will be the immediate value. The result is stored in *rt*. For loads and stores, *rs* and the immediate-value is used for address calculation whereas register *rt* contains either the value to be stored to memory, or the register where data should be loaded into. In branch instructions, *rs* and *rt* is used for the comparison and the immediate value is the relative change in the program counter when a branch is taken.

3.1.2 Loads and stores

MIPS supports loads and stores of 8-bit bytes, 16-bit halfwords and 32-bit words. Memory addresses are calculated by adding the value of a GPR to the instruction's sign-extended immediate value. All loads and store instructions are I-type instructions. The instructions for loading bytes and halfwords have two versions for specifying how the values will be extended to 32 bits. The normal load will sign-extend the value and the unsigned load will zero-extend the loaded value. Every load is followed by a one cycle load delay slot. This slot must be filled with an instruction that does not use the loaded data to ensure correct

operation. If no independent instruction can be scheduled, the slot must be filled with a *nop*. MIPS I requires all memory accesses to be aligned to their word boundaries. In order to support unaligned memory accesses, there are special load/store instructions with the "right" and "left" suffix. These instructions will load/store only a partial word. Using these instructions, a unaligned word can be loaded/stored in two instructions.

3.1.3 ALU

The ALU is used for most instructions in MIPS I. In addition to the normal arithmetic and logic operations, it is also used for generating memory addresses. In R-type instructions, the ALU operation is specified in the *funct* field. For I-type instructions it is specified in the opcode. Not every R- and I-type instruction uses the ALU. The exceptions are jumps and branches, which are handled earlier in the pipeline, and moving data to/from the *HI* and *LO* registers. By default, addition and subtraction in the MIPS ISA will trigger an exception if the result overflows. However, there are variants of these instructions with the "unsigned" suffix that do not cause an exception on overflow. The term "unsigned" here is misleading, and must not be confused with its use in other instructions where it means not sign-extended. The "set on less than"-instructions will set the destination register to one if the relation is true and zero otherwise.

All ALU operations takes a single cycle to complete, except for multiplication and division. These instructions takes several cycles, and are therefore performed asynchronously in their own units, allowing other instructions to be executed at the same time. An attempt to accessed the results before they are ready will stall the pipeline until the results are ready. The results of multiplication and division is stored in the special *HI* and *LO* registers. The 64-bit product of a multiplication is split in half, storing the 32-bit high order word in *HI*, and the 32-bit low order word in *LO*. For division, the quotient is stored in *LO* and the remainder in *HI*. The result from both multiplication and division is always sign-extended before being stored in *HI* and *LO*. These special registers cannot be used by normal instructions, and must therefore be moved to the GPRs to be usable. This is handled by the *mfhi* and *mflo* instructions. There are also two instructions which move data from a GPR to either *HI* or *LO*, but their only purpose in MIPS I is to restore state after exception handling, which means we can ignore them.

3.1.4 Jumps and branches

Jumps and branches are evaluated in the instruction decode stage using a small ALU separate from the main ALU in the execute stage. All jumps and branches have a delay of one instruction, similar to the memory loads. The instruction immediately following the jump or branch, in the branch delay slot, is executed before the branch is taken. The instruction in the branch delay slot will always be executed, regardless of the outcome of the branch condition. If no suitable instruction can be scheduled, a nop instruction should be used. If the decoded instruction is determined to be a jump or branch, the address calculation will happen in parallel with reading the registers. The branch address is generated by first shifting the 16-bit *immediate*-field left twice and-sign extending it. This value is then added to the address of the instruction following the branch (the branch delay slot). Jump addresses are absolute, and can jump within the current 256 MB aligned memory region. The low

28 bits of the jump address is the *address*-field left shifted twice. The remaining four high order bits comes from the corresponding bits of the address of the instruction in the branch delay slot. When using a register value as a jump address, jumps are no longer limited in range, but their addresses must be word-aligned to avoid triggering an exception.

Some jump and branch instructions have the "link" suffix. This means they will store the return address in GPR 31 (*ra*) before executing the jump or branch. The return address is the address of the second instruction following the jump/branch. This is to avoid infinite loops and to avoid re-execution of the branch delay slot. The *jalr* instruction allows for storing the return address in any GPR.

3.2 Caching

Main memory can take up to a hundred CPU cycles to access. This problem would have lead to many stall cycles had it not been addressed. Caches are used in order to hide this latency, by providing a smaller but faster memory between the CPU and main memory. Caches are much smaller than main memory, but are able to exploit temporal and spatial locality to achieve high hit-rates. On modern processors there is usually a hierarchy of caches, with the smallest and fastest closest to the CPU, and larger and slower closer to main memory. The cache closest to the CPU is the called the level-1 (L1) cache and behind it is the level-2 (L2) cache, and so on. When the cache does not contain the requested data, a cache miss occurs, and the cache will request the data from the next cache in the hierarchy. In order to exploit spatial locality, caches usually fetch a large memory block containing adjacent words in addition to the requested data during a cache miss.

Cache lines are not randomly placed, but adhere to some placement policy. The simplest placement policy is the direct mapped cache. Based on the address of the memory block, it can only be placed in a single cache line. This means there is no need for a replacement policy. The low order bits of the address determines where in the cache it should be placed. The high order bits of the address is called the tag (see Figure 3.2), and is also stored with the cache line. When a direct mapped cache is accessed, the low order bits of the address is used to index the cache and the high order bits are compared with the stored tag. If the tag matches, it is a cache hit and the data is returned to the CPU. Direct-mapped caches have the advantage of being simple and inexpensive, but at the cost of lower hit rates.

L1 caches typically access the tag and data arrays in parallel in order to reduce latency during load instructions. This is commonly known as a *conventional cache*. In level-two (L2) and level-three (L3) caches it is more common to first access the tag arrays before accessing the data arrays. This is known as a *phased cache* and is more energy efficient since it only need to access (at most) a single data array since the tag check has already completed (Megalingam et al., 2009). Phased caches are uncommon at the L1 stage because of the longer execution times offsetting the energy savings from fewer data array accesses.

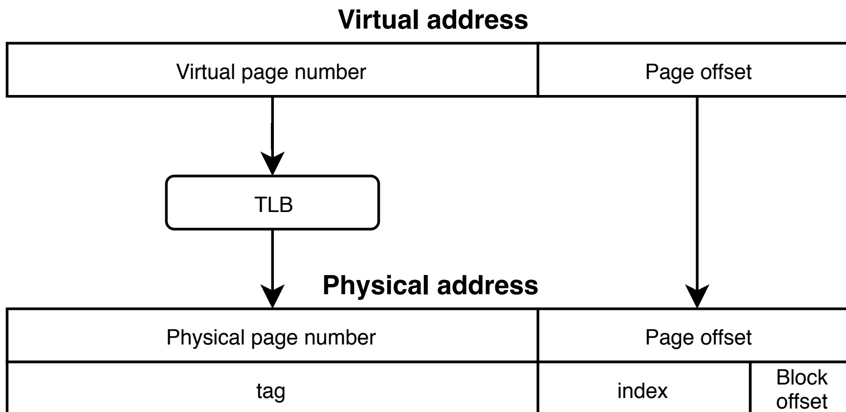


Figure 3.2: Address fields.

3.2.1 Set associative caches

Set associative caches enables the memory block to be placed in multiple cache lines. The cache is split into n sets, where each set contains m cache lines. 2-way set associative caches have two cache lines per set, and 4-way have four cache lines per set etc. A special case is fully associative caches which allows memory blocks to be placed anywhere in the cache. The hardware complexity and energy consumption increases with the associativity, but also increases the hit rate which leads to fewer stalls and lower energy usage.

Set associative caches are still indexed by the low order bits of the address, however each index can now store multiple cache lines. When an N-way set associative cache is accessed, every tag in the indexed set will have to be compared in order to find the requested data. Figure 3.3 shows how an in-order processor performs a load from memory. In order to reduce the critical path, both the tag and data arrays are accessed in parallel. When the tag check is complete, the accessed data words are multiplexed so the correct data word is selected. This is an inefficient approach, as the data can only reside in at most one way. Performing the tag check before accessing the data arrays would increase the critical path and lead to higher power consumption. Parallel tag- and data-array access is mostly done in L1 caches (which are accessed very often). L2 and L3 caches are not accessed very often, and can therefore employ a phased cache where the tag array is compared in one cycle, and at most a single data array is accessed in the next cycle.

3.2.2 Replacement policies

When a set is full in a set associative cache, a replacement policy is required to determine which cache line should be discarded in order to make room for a new line. Least recently used (LRU) is a common replacement policy, but is expensive to implement and has a high power consumption. LRU also scales poorly with associativity, becoming increasingly more expensive. Pseudo-LRU (PLRU) is a group of replacement policies that approximate the LRU policy, but with simpler logic and less power consumption.

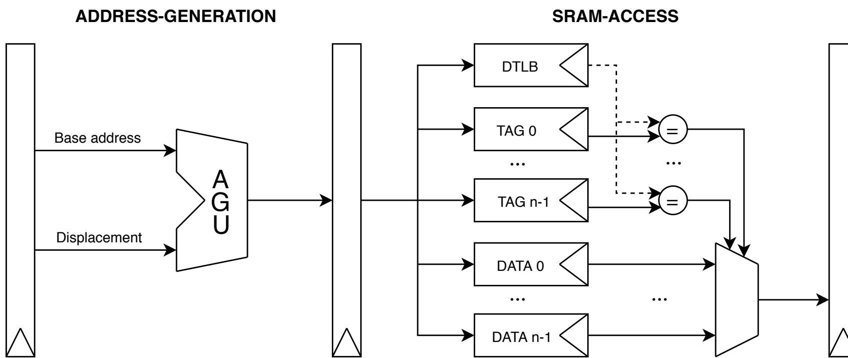


Figure 3.3: Conventional L1 DC pipelined access.

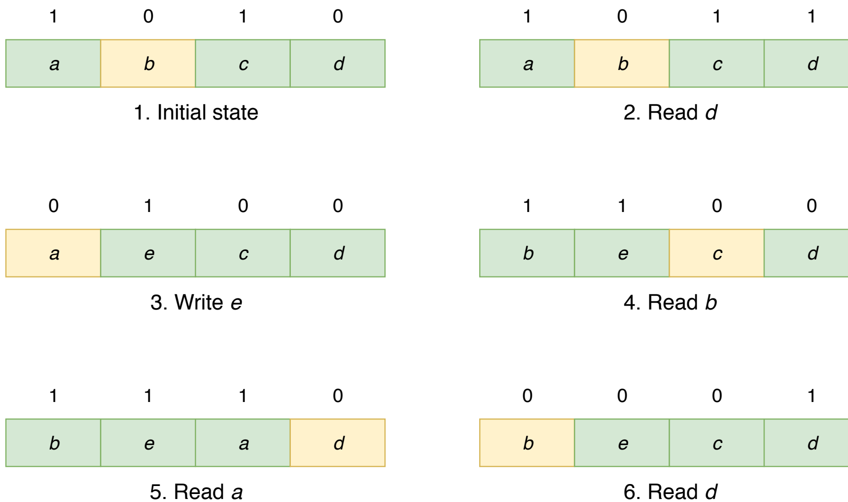


Figure 3.4: PLRUm sequence

MRU based pseudo LRU (PLRUm) is one of these replacement policies. It works by assigning a single bit to each cache line. When this bit is set to one, it indicates that this line was used more recently than the lines without the bit set. When a cache line is accessed, its corresponding bit is set to one. When a cache line should be replaced, the cache controller searches for a cache line with a zero bit, replaces the line, and sets the MRU bit to one. If all MRU bits are set to one, they are all reset to zero, except the MRU bit for the current access. An example of the PLRUm algorithm can be seen in Figure 3.4. The yellow boxes indicates which cache line is next to be replaced. The PLRUm policy can be conceptualised as a finite state machine, and can therefore be implemented in many ways (Fatemi et al., 1994). The PLRUm policy also requires less storage and consumes less power while performing nearly as well as LRU, and sometimes outperforming it (Al-Zoubi et al., 2004; Gille, 2007).

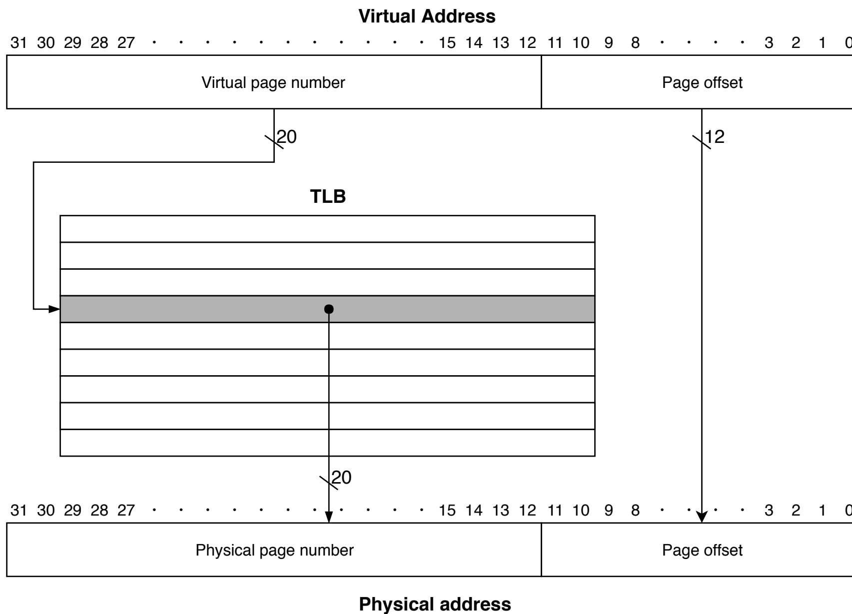


Figure 3.5: Overview of the address translation process. The TLB is indexed with the virtual page number and yields the physical page number. The page offset is the same for both the physical and virtual page. In this figure the page size is 2^{12} bytes or 4 kB. Although each table entry here is 20 bits wide, there are usually extra bits stored for each entry, such as validity and page protection bits.

3.3 Data translation lookaside buffer

In order to make computer programs simpler to program, they use the illusion of a large and continuous address space. In reality there are multiple programs running in the same finite memory space, and it is the OS's job to ensure this illusion is maintained. This technique is called virtual memory, and each process has their own virtual memory address space. This also has the benefit of protecting memory between processes. In order to access main memory, the virtual address must be translated to a physical address. This task is a joint operating system and hardware effort. The operating system will allocate large memory blocks called pages when a program requests memory. The mapping between a program's virtual pages and the physical pages is stored in the page table which is usually in the operating system's portion of main memory. In some cases, the page table will become too large, and parts of it must be swapped out to secondary memory.

A small portion of the page table is cached for faster access, in a structure called the translation lookaside buffer (TLB). This cache works very similarly to normal caches, and can be direct-mapped, set associative or fully associative. However, since these caches are usually small, fully associative caches are commonly used. The low order bits of an address is called the page offset and is the same for both physical and virtual addresses (see Figure 3.2). This offset therefore does not need to be cached. Figure 3.5 shows the translation process. When memory is accessed, the virtual page number goes to the TLB,

which translates it into a physical page number, which is then used in the L1 cache. On a TLB miss, the correct entry must be fetched from the page table. This can either be done in software or hardware. In software, an exception is raised on a TLB miss, and the OS is then responsible for putting the correct data in the TLB with privileged instructions and restarting the program at the instruction which raised the exception. If miss handling is done in hardware, the memory management unit will walk the page table, searching for a correct mapping before continuing.

Chapter 4

Baseline implementation

The implemented processor is an 32-bit, 5-stage pipelined processor which supports most of the MIPS I instruction set. Since the goal of this paper is to explore energy efficiency in caching, some instructions have been omitted. There are no coprocessors implemented, removing the need for floating point instructions and coprocessor communication instructions. In addition we will not be running an operating system, so we do not implement *syscall* and *break*. Also, overflow exceptions from addition and subtraction instructions will be ignored. Finally, *mthi* and *mtlo* are also not implemented since their only function in MIPS I was to restore state after an exception. This leaves us with a total of 51 supported instructions out of the original 58 (excluding coprocessors).

A simplified overview of the processor can be seen in Figure 4.1. The program counter (PC) is located in the instruction fetch (IF) stage, and contains the address of the next instruction in memory. The PC is incremented by 4 every cycle unless there is a branch, in which case the address is set to the branch address. In the instruction decode (ID) stage, the registers specified in *rs* and *rt* are read and sent to the next stage together with the *immediate* value. The execute (EX) stage computes some operation on the operands, either both from registers or one from a register and the other from the *immediate* field. The result is sent to the next stage and either used as a memory address or passed through to the next stage. If the instruction is a load or store, the memory is accessed in the memory (MEM) stage. Finally, in the write-back (WB) stage, the result from the ALU operation or the loaded data is written back to register file. Some components and logic have been omitted to make the figure more clear. These components will be discussed in further detail in the following sections.

4.1 Hazard detection and forwarding

Pipelines are subject to certain hazards which happen when an instruction is unable to execute during its cycle, or would cause incorrect computation results. In Figure 4.1 the hazard detection unit (HDU) takes care of both detecting hazards in the pipeline and issuing stalls and bubbles where necessary. Our pipeline has no control hazards since there

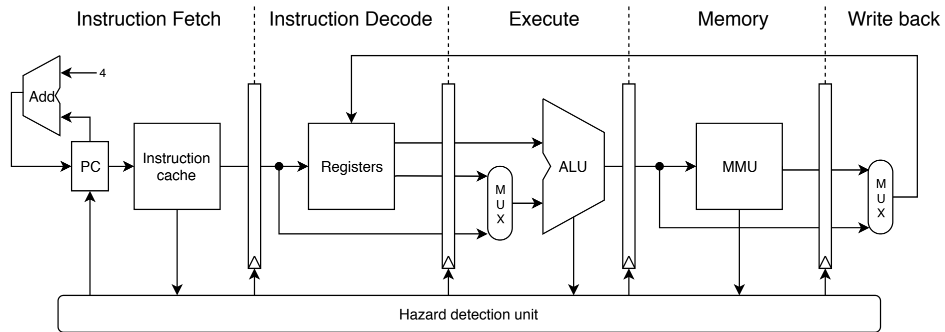


Figure 4.1: Simplified overview of the 5-stage MIPS processor.

is only a single cycle between the instruction is fetched and the branch is handled, and in the MIPS I architecture this cycle is filled with the branch delay slot. The instruction occupying this slot will be executed regardless of the outcome of the branch condition. We evaluate jumps and branches in the instruction decode stage with a small ALU, separate from the main ALU in the execute stage. This allows us to know what instruction to fetch after the branch delay slot and does not require branch prediction. Likewise, our pipeline has no real structural hazards either. The pipeline is in-order, and every component will only be used by the instruction currently in that pipeline stage.

The third type of hazard is the data hazard and means that an instruction cannot execute because it is waiting for data that is not yet available. This happens because there are several stages between reading a register and writing back the result. This happens both to the main ALU and to the small branching ALU. To solve this problem, data is forwarded from either the MEM stage or WB stage to the appropriate ALU when a data hazard is detected. This solves most problems, but when a load is immediately followed by an instruction requiring the result of that load, the pipeline must stall for one cycle. If the subsequent instruction is a branch on the loaded value, two cycles must be stalled before the result can be forwarded. There is also forwarding inside the register file itself, allowing the instruction currently in ID to obtain the values before they are written to the registers.

The HDU is also responsible for stalling when misses occur in either the L1 IC, L1 DC or the DTLB. Stalls are also inserted when the *HI/LO* registers are accessed while a multiplication or division is in progress. Stalls are solved by freezing the program counter and pipeline registers that are earlier in the pipeline than the stalling component.

4.2 Instruction Decode

In addition to the registers pictured in Figure 4.1, the instruction decode (ID) stage also contains the control unit and the branching unit. The control unit decodes the current instruction based on its opcode and sets the appropriate control signals, such as ALU-operation, memory operation, write back etc. In case of a branch instruction, or unsupported opcode, all control lines will be 0, meaning no action will be taken. The branching unit determines if the current instruction is a branch instruction and also checks the branch

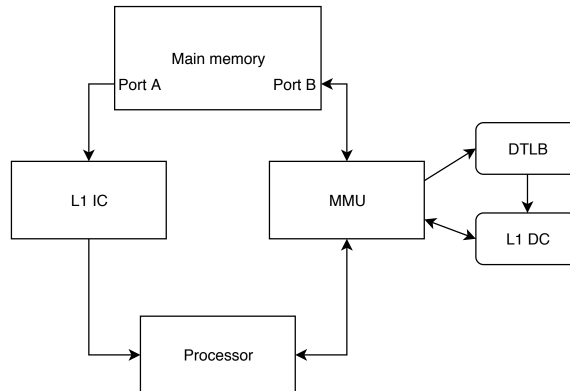


Figure 4.2: Overview of the memory hierarchy

condition. On a successful branch, the new address is computed and sent back to the program counter in the IF stage. Operands to the branching unit either come directly from the registers, but can also be forwarded from later pipeline stages. The instruction immediately following a branch (occupying the branch delay slot) will be executed regardless of outcome of the branch. This simplifies the design since no branch prediction is needed.

4.3 Execute

The main component in the execute stage is the ALU. Operands for the ALU can come from either the registers, the immediate value in the instruction, or be forwarded from later pipeline stages. The immediate value is either sign-extended for arithmetic operations or zero-extended for logical operations. The ALU operation is decided by a combination of the control unit in the ID stage and the *funct* field in the instruction. In addition to the ALU, there are two separate components for multiplication and division. Since multiplication takes six cycles and division 36 cycles, they are designed to operate asynchronously to the main pipeline. The result of multiplication and division is stored in the *HI* and *LO* registers, which are located in the execute stage. These registers can only be read by *mfhi* and *mflo*.

4.4 Memory hierarchy

An overview of the memory hierarchy can be seen in Figure 4.2. Main memory is implemented as a dual-ported RAM. Port A is connected to the L1 IC and only supports reads when the processor is running. During setup it is possible to write through this port to initialise the RAM with data. Port B is connected to the memory management unit (MMU) depicted in Figure 4.1 and always supports reads and writes. This component is responsible for the communication between processor, DTLB, L1 DC and RAM. Additionally it decodes the memory operation from the control unit and sets the correct enable signals depending on the operation. During *swl*, *swr*, *lwl* and *lwr* instructions it will also format

Table 4.1: Memory configurations

Page size	256 B
Main memory	256 kB
L1 DC	1 kB size, 16 B line size, 4 way set associative, 1 cycle hit, 10 cycle miss
DTLB	32 entries, fully associative, 1 cycle hit, 10 cycle miss

the data correctly and setting the correct byte-write enable signals for the registers. The memory configuration for the DTLB and L1 DC can be seen in Table 4.1.

4.5 DTLB

The DTLB is implemented as a state machine with three states: *ready* and *miss*. In the *ready* state, the DTLB can be enabled and will then perform a tag check against the given virtual page and the 32 tags stored in the DTLB. If one of these tags are a match, the corresponding physical page will be served as output. In the case that no tags match the given virtual address, the DTLB will set the stall signal and move to the *miss* state. In this state we simulate a miss, and will stall for ten cycles. During this we will also copy the virtual page to both the tag arrays and data arrays. After ten cycles we turn off the stall signal, move to the *ready* state and serve the correct physical page as output. If the DTLB is full, and a new page translation needs to be added we use the PLRUm replacement policy as described in Section 3.2.2.

4.6 L1 Cache

Both the L1 DC and the L1 IC are implemented as 4-way set associative caches, and only have two differences. First, the instruction cache is always enabled, unless there is a stall, whereas the data cache is only enabled when there is a memory operation. Secondly, the instruction cache is only read, whereas the data cache is both written to and read from. The cache implementation can be seen in Figure 4.3. The caches use a state machine similar to the DTLB, but has one an *allocate* state in addition to the *ready* and *miss* states. In the *ready* state, the physical page from the DTLB is compared against the four tags of the current index. If the memory operation is a load, the four data arrays are also accessed in

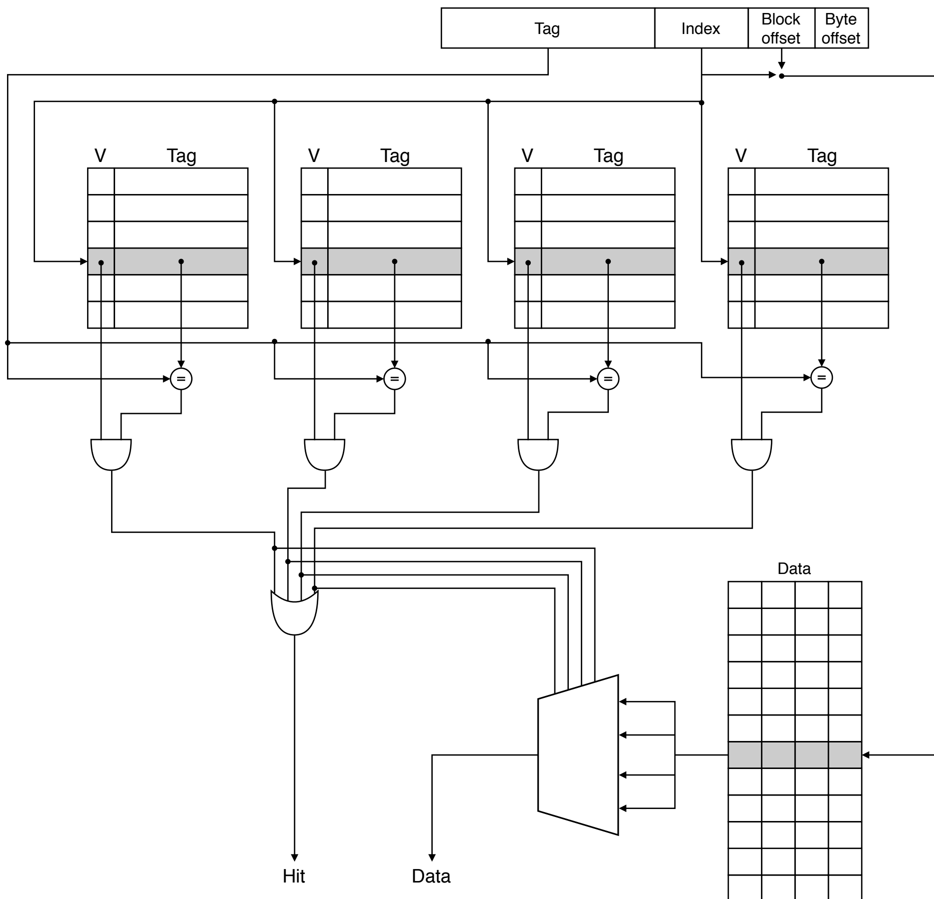


Figure 4.3: Implementation of the 4-way set associative cache with separate data arrays.

parallel. If the memory operation is a hit, the corresponding data array is sent as output on a load and the corresponding data array is written on a store. In the case that none of the tags match the physical page number, the stall signal is set and the state is changed to the *miss* state.

In the *miss* state we enable main memory and set the correct signals for the missing data. The RAM in our FPGA has a two-cycle access time, so we also wait for 7 more cycles before continuing to simulate a slower RAM. After nine cycles, we move to the *allocate* state. In this state we find the least recently used cache line for this index and replace it with the one from RAM. After this is done we transition back to the *ready* state. In the case of a store operation we utilise the write-through policy with no-write allocate. This means data is written directly to RAM, regardless the result of the tag check. If the tag check hits, we also write the stored value to our cache line to keep cache and RAM consistent. After writing the data, we transition back to the *ready* state.

Chapter 5

PAM instructions and pipeline changes

Not every instruction needs to be considered to be a potential PAM instruction. In fact, only a handful of instructions needs to be considered to cover almost all cases. These instructions can be seen in table 5.1. As mentioned in Section 4, the overflow versions of add and subtract instructions are simply treated as their non-overflow counterparts. All of these instructions can be detected at run time as potential PAM instructions, but we will not be considering *lui* and *lw* because they are relatively infrequently used. Tracking *lui* and *lw* can potentially identify more PAM instructions, but the energy consumed detecting them might outweigh the energy saved.

Table 5.1: Last instruction to compute data address

Instruction	Operation	Effect
add / addu	Add	rd = rs + rt
addi / addiu	Add immediate	rt = rs + immediate
sub / subu	Subtract	rd = rs - rt
lui	Load upper immediate	rt = immediate «16
ori	Logical OR immediate	rt = rs + immediate
lw	Load word	rt = MEM[rs + immediate]

In order to introduce the DCAM technique we need to perform some changes in the pipeline. Table 5.2a shows the five standard pipeline stages and Table 5.2b shows the addition three stages that have to be introduced. In the conventional pipeline, the MEM stage consists of the DTLB access, L1 DC tag check and L1 DC data access which are all executed in parallel. In the new pipeline we have moved these steps into two separate pipeline stages. We also have a new stage, DAS, which uses previously calculated way information to only access a single data array. We assume that the TC, DA and DAS stages take a single cycle on a hit, although they can easily be extended to take multiple

cycles. The pipeline length is still five stages, however various instructions will now utilise different parts of the pipeline, most notably in the MEM stage.

Table 5.2: Pipeline stages

(a) Conventional

Stage	Explanation
1. IF	Instruction Fetch
2. ID	Instruction Decode
3. EX	Execute
4. MEM	TC + DA
5. EX	Execute

(b) DCAM

Stage	Explanation
6. TC	DTLB access and L1 DC tag
7. DA	L1 DC data access all ways
8. DAS	L1 DC data access single way

Table 5.3 shows what stages various instructions will use as they move through the pipeline. Some stages are marked with N/A, meaning the instruction performs no action in this stage, and only passes information to the next stage (if it is not the last stage). The biggest change from the conventional pipeline is that during *pam* ALU instructions we now perform the DTLB access and L1 DC tag check in the fourth stage. This stage is normally not used, and information is passed through to the WB stage. Conventional loads and stores have to perform the DTLB access, L1 DC tag check and access the L1 DC data array all in a single stage. With the new additions, loads and stores which are able to use PAM data access information, can now access a single data array and skip both the DTLB access and L1 DC tag check, and access a single data array.

Table 5.3: Pipeline stages used by various instructions

Instruction	Pipeline stages				
ALU instruction	IF	ID	EX	N/A	WB
Load instruction	IF	ID	EX	MEM	WB
Store instruction	IF	ID	EX	MEM	N/A
ALU instruction [*pam*]	IF	ID	EX	TC	WB
Load after *pam*	IF	ID	EX	DAS	WB
store after *pam*	IF	ID	EX	DAS	N/A

Chapter 6

Improved pipeline

The improved pipeline is based on the baseline implementation, but has several additional components and logic. The most important part is the data cache access structure (DCAS) and the accompanying DCAS valid info (DCAV). In addition to these structures, there are new datapaths between the DCAS, DCAV, ALU and the MMU. We also detect PAM instructions in the execute stage which tells the DCAS whether to issue a tag check or not. A simplified overview of the additions to the pipeline can be seen in Figure 6.1. The new components and datapaths are marked in blue. The DCAS and DCAV are tightly connected and are therefore visualised in the same box in this figure. We also see that we need some carry signals from the ALU in order to know when we cross page boundaries and cache line boundaries. The line that goes from the DCAS/DCAV to the MMU contains information such as when to tag check and when we can use a stored way. The line from the MMU back to the DCAS/DCAV contains the way that should be stored and what the destination register is and invalidation information from the L1 DC. The DCAS also needs all register addresses (*rs*, *rt* and *rd*) to know which register to associate tag check information with. It also needs the immediate to check if it is small enough.

There are other small additions which is not shown in the figure. The DTLB and L1 DC both have an extra multiplexer to select between the way from the tag check or the DCAS stored way. Both structures also have new datapaths which route the correct way from the tag check back to the DCAS. The L1 DC is also able to run two tag checks in parallel in order to track both the current cache line and the next sequential cache line. The remainder of this chapter will focus on the DCAS and DCAV structure.

6.1 DCAS

The Data Cache Access Structure (DCAS) will be used to store the result from DTLB look-ups and L1 DC tag checks performed by PAM instructions. Storing this information in a structure has the benefit of supporting PAM instructions which are separated by several instructions from their corresponding load or store instruction. In addition, by storing this information in a structure, other instructions which access the same cache line or memory

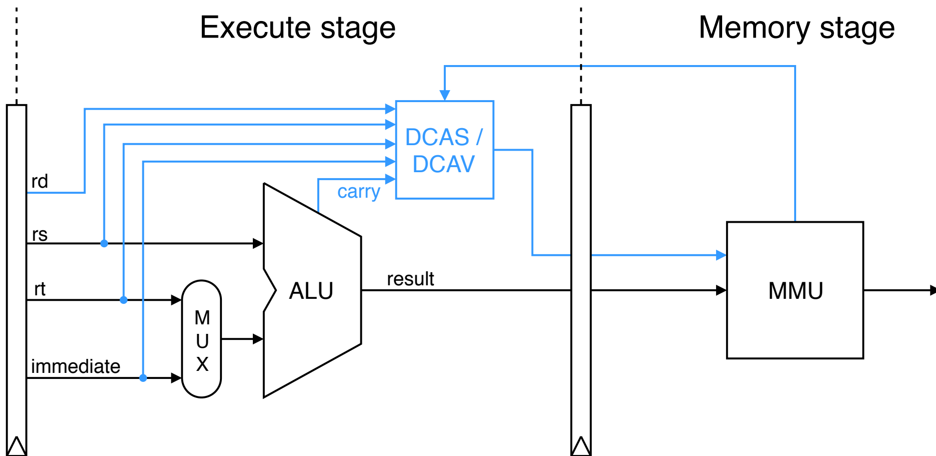


Figure 6.1: Simplified overview of the additions for the improved pipeline.

```
1. r2=sp+16;[pam]
2. r3=M[r2]
3. r5=r3
4. r3=r3+r4
5. M[r2]=r3
```

(a) Load store same address

```
1.     r2=sp+16;[pam]
2. L1: r3=M[r2]
3.     r4=r4+r3
4.     r2=r2+4;[pam]
5.     PC=L1 if r2!=r5
```

(b) Strided access

Figure 6.2: Memory access patterns

page can use it without performing their own DTLB look-ups and tag checks. This will remove several redundant DTLB accesses and tag checks in common programming patterns as can be seen in Figure 6.2. In Figure 6.2a, both the load on line two and store on line five can use the access information from the PAM instruction. In Figure 6.2b, The load at line two can skip the DTLB and tag check. Furthermore, the PAM instruction at line four can skip the DTLB access and tag check if the new calculated address is still within the same cache line as the previous PAM instruction.

One problem is that the address in the base register may not refer to the same cache line as the effective address computed by adding the base register and the displacement. In the DCAS we track both the current and next sequential cache line associated with the base register. This means we can allow small non-negative displacements as long as they are smaller than the L1 DC line size. This has the additional benefit that once we use the next sequential line, we can copy that to the current line and look perform the tag check for the next sequential line in the next PAM instruction. In loops with small increments, this will allow us to continually access a single L1 DC data array and skip DTLB tag checks (untill we cross a page boundary).

	DWV	DTLB way	LWV	L1 DC way	LWVN	L1 DC N way	PP
0							
1							
...							
31							

Figure 6.3: Data Cache Access Structure (DCAS)

The DCAS structure can be seen in Figure 6.3. This structure has one row of fields for each integer register. The *DWV* bit indicates that the DTLB way field is valid. If the *DWV* bit is not set then the rest of the entry is also considered invalid. The *DTLB way* field holds the DTLB way where the associated physical page is located. The *LWV* bit indicates if the *L1 DC way* field is valid. The *L1 DC way* field contains the way in which the cache line associated with the base register resides. The *LWVN* bit indicates if the *L1 DC N way* field is valid. The *L1 DC N way* field contains the way for the next sequential line for the address in the base register. The *PP* field contains page protection bits from the DTLB. This is necessary because we sometimes avoid DTLB accesses, but still need to make sure pages are accessed correctly. We only need to store the way to access and not the set index into the cache since it will be calculated anew during address generation.

The DCAS structure is accessed in the EX stage when either a PAM instruction has been identified or during a load or store instruction. For PAM instructions the destination register is used to index the structure and for load and stores the base register is used. In both cases the DCAS is checked to see what fields are valid and determine which tag checks and read signals must be set.

6.1.1 Updating DCAS

It is important to recognise when the address in a register is updated, but still points to the same cache line or the same page. Fortunately it is simple to detect when the cache line or page changes during an effective address computation as can be seen in Figure 6.4. For both an immediate addition during either a load or store (M[rs+immediate]) or a PAM immediate addition we can simply inspect the carry out values from the ALU and see if they cross either the cache line boundary or the page boundary. During a load or store, if we cross the cache line offset, we can use the *L1 DC N way*, assuming the *LWVN* is set and we do not cross page boundaries. If the *LWVN* is not set we can access a single DTLB way to obtain the tag and use it in the L1 DC tag comparison. If however, we cross the page boundary, then we have to perform both a full DTLB tag check and L1 DC tag check. This technique also works for a PAM register add instruction, since the operation is the same.

Another technique we implement is to copy the DCAS information associated with one register to another during PAM register addition. If one of the source registers in the add instruction has a valid DCAS entry, we can copy that entry to the entry associated with

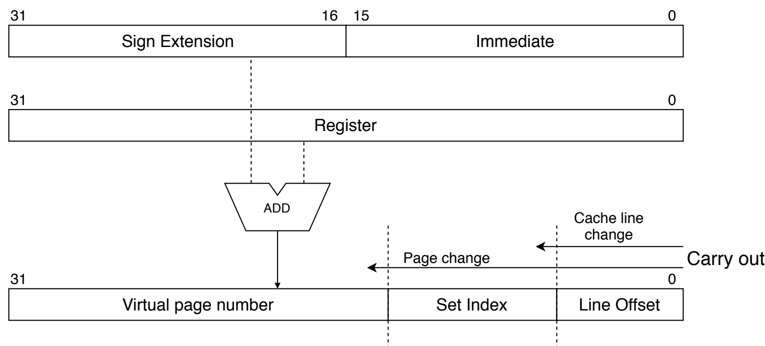


Figure 6.4: Detecting cache line and page changes.

the destination register. Here we also need to check that the source register and destination register differ, and we must inspect the carry values to determine what information to copy. If we don't cross the cache line boundary, we can copy the entire DCAS entry. If we cross the cache line boundary, but not the page boundary, we can copy the *DWV* and DTLB way and issue a L1 DC tag check. If we cross the page boundary, we have to perform both a DTLB tag check and L1 DC tag check.

6.2 DCAV

Figure 6.5 shows the DCAS Valid Info (DCAV) structure that is used to invalidate DCAS entries when a cache line is evicted or invalidated. Each row in the DCAV structure contains 32 bits, one for each integer register. The structure is indexed with the L1 DC way with *n* being the associativity level for the L1 DC. When a DCAS entry is associated with a cache line, the register and L1 DC way is used to set the correct bit in the DCAV. When the *LWV* bit is cleared for a DCAS entry, the corresponding register is used to clear all bits with that register number in the DCAV. When an L1 DC line is evicted or invalidated, the way where it resided is used to index the DCAV and find all DCAS entries that need to have their *LWV* bit cleared. If a page is evicted from the DTLB, the entire DCAV structure is cleared, however this is a rare occurrence.

6.3 Pipeline example

Table 6.1 shows an example with four instructions as they move through the pipeline. We assume for simplicity that both the instruction cache and data cache have a single cycle access time. The stages marked in bold are stages introduced with the DCAS technique and is not found in a conventional pipeline. The first instruction has been detected as a PAM instruction, and will provide access information for the second instruction. When instruction one reaches the execute stage, it will index the DCAS with its destination register *r2*. If the *DWV* bit is set, we can inspect the carry out values. If there is no carry out and the *LWV* bit is set, or if there is carry out and the *LWVN* bit is set we can skip the L1

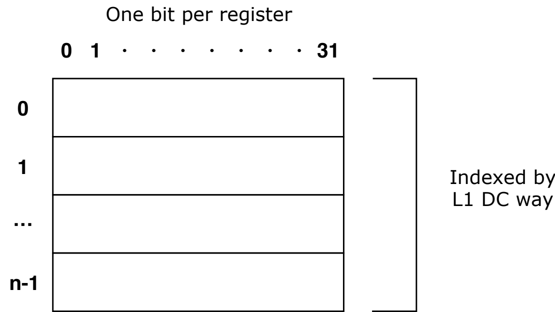


Figure 6.5: DCAS Valid Info (DCAV)

Table 6.1: DCAS pipeline example

Instruction	1	2	3	4	5	6	7	8
1. r2=r1+4 [pam]	IF	ID	EX	**TC**	WB			
2. r3=MEM[r2+0]		IF	ID	EX	**DAS**	WB		
3. r4=r2+4 [pam]			IF	ID	EX	N/A	WB	
4. r5=MEM[r4+4]				IF	ID	EX	**DAS**	WB

DC tag check. If neither of these conditions are met, we have to access a single DTLB way, and perform the L1 DC tag check. If the *DWV* bit is not set, we have to perform a full DTLB tag check and L1 DC tag check. When the first instruction is in the TC stage it will perform the tag checks and data reads determined in the EX stage, and write the results from tag checks back to the DCAS.

The second instruction will use the information calculated by instruction one. There is forwarding implemented which means a memory instruction can immediately follow a PAM instruction and still use the information the PAM instruction calculates. When the second instruction is in the execute stage, it will index the DCAS with its source register *r2*. The value in the DCAS has not yet been written, so a multiplexer gives the memory instruction information directly from the tag check of the PAM instruction. We must also check the carry values in a similar manner as the first instruction to determine which values we can use. However, since this instruction has an immediate displacement of zero. All information from instruction one can be used, and we only have to read a single data array in the DAS stage.

Instruction three is another PAM instruction that will prepare data access information to be used by instruction four. During the execute stage we index the DCAS structure with the destination register *r4*. In this example we assume that there is no valid information stored for this register. However, the source register *r2* of the instruction, has valid DCAS information. We can then inspect the carry values to see which parts of *r2*'s DCAS entry we can copy to *r4*. We assume that we do not cross either the cache line boundary or the page boundary in this example. This means the entire entry can be copied and we do not need to perform an action in the TC stage.

Instruction four operates very similar to instruction two, but has to check the carry out

values since it has a non-zero displacement. If we do not cross the page boundary, we can use either the current cache line or the next sequential line and avoid both the DTLB tag check and L1 DC tag check.

Chapter 7

Methodology and results

Both the baseline pipeline and the DCAM version are written in VHDL. Main memory (Xilinx, 2017), multiplication (Xilinx, 2015) and division (Xilinx, 2016b) is created using IPs provided by Xilinx. The pipelines are synthesised and implemented using Xilinx Vivado (Xilinx, 2018b). The implemented designs have then been tested on a Digilent PYNQ-Z1 board (Digilent, 2019), which contains an Artix 7-series FPGA from Xilinx. Vivado contains industry grade tools which provide accurate estimates for both power usage, timing analysis and resource usage. To test the pipelines we have used the *dijkstra* benchmark from MiBench (Guthaus et al., 2001). The benchmark is compiled using GCC with the -O3 option. We have used the Newlib C library to make interfacing with our bare-metal processor easier. MiBench also provide expected outputs for each of their benchmarks allowing us to verify that our processor is implemented correctly.

7.1 Resource usage

An important metric to consider when adding extra logic to the pipeline is the increase in resources used. The improvement that is implemented has to save more energy than it consumes by adding extra logic. In the case of the DCAM technique there are significantly more resources that have to be utilised to make it work. Table 7.1 shows the resource usage for various components in the baseline implementation. The *Other* category is for signals that don't belong to a specific stage such as stall signals and hazard detection. The flip flops in this category is for logic associated with setup and writing data to memory before starting the processor.

Logic LUTs consists of traditional LUT primitives such as LUT2 to LUT6 which implement some boolean function (Xilinx, 2016a). Memory LUTs, on the other hand, consists of small distributed RAM blocks such as RAM32X1S and RAM64X1S. These primitives have a single port and are able to hold 32x1 bits and 64x1 bits respectively. This category also contains shift registers which are used for multiplication and division in the execute stage. The distributed RAM primitives are much more efficient in terms of both energy and slice usage compared to regular flip-flops. A flip-flop can hold a single

Table 7.1: Baseline pipeline resource usage.

Component	Logic LUTs	Memory LUTs	Flip-flops	BRAM	DSP
Main memory	264	0	8	64	0
L1 IC	399	552	130	0	0
L1 DC	1796	544	514	0	0
DTLB	251	8	296	0	0
IF stage	17	0	96	0	0
ID stage	817	0	1133	0	0
EX stage	2103	23	3641	0	4
MEM stage	87	0	71	0	0
Other	94	0	161	0	0
Total	5828	1127	6050	64	4

bit, and is usually used where there is not enough data to store in order to justify a larger RAM primitive. Block RAM (BRAM) consists of large RAMB36E1 blocks, which can store 36Kb each (Xilinx, 2019). This is only used for main memory because of their large size and limited flexibility.

The digital signal processing (DSP) units is only used in the execute stage for multiplication since it is a complex process. The DSP48E1 blocks are specifically designed to perform fast multiplication, among other things (Xilinx, 2018a). The DSP48E1 slices could also be used to implement efficient division, however they only support fractional output whereas MIPS expects to get the remainder. Division is therefore implemented with LUTs and flip-flops which is the reason the execute stage uses so much resources. In fact, 91% of the LUTs and 61% of the flip-flops in the execute stage is used to implement division. The instruction decode stage has a large number of flip flops, but considering that the 31 GPRs (not counting $zero) takes $31 \times 32 = 992$ bits, this value makes sense. The remaining flip-flops in the ID stage and the other stages are mostly used for pipeline registers between the various stages.

Although the L1 DC and L1 IC are implemented in a very similar way, the L1 DC requires many more logic LUTs than it's counterpart. There are several reasons for this, but most importantly it is because it needs to handle writes from both the processor side when it wants to write something to memory, and from the memory side when data is loaded. The instruction cache only needs to handle writes from main memory. Secondly, the data cache has an accompanying DTLB which makes the process more complicated. Finally, the L1 DC has to support byte-wide write enables from the processor whereas the L1 IC always operates on larger data units.

Table 7.2 shows the resources used in the improved version of the pipeline. The entries which differ from the baseline (Table 7.1) have the difference listed in parentheses. The biggest change by far compared to the baseline, is the additional logic LUTs and flip-flops required in the execute stage. Most of these come from implementing the DCAM technique, but around 600 LUTs and 400 flip-flops are introduced to detect PAM instructions. Still, this leaves us with over 1200 LUTs and over 500 flip-flops to implement the DCAM technique. The 500 flip-flops comes from the fact that the DCAS structure has 32 rows $\times 12$ bits $= 384$ flip flops. The twelve bits per row is from to using a 4-way set associative

cache and 32-way DTLB. Additionally, the DCAV structure takes $32 \times 4 = 128$ bits for a 4-way cache. The remaining flip-flops not accounted for is used in the EX/MEM pipeline register to store the extra signals from EX to MEM.

The amount of logic LUTs used for the DCAS and DCAV is also very high. This is because each field in the structure can be set and unset by many different sources. The *DWV* and *DTLB WAY* can be updated by either the output from the DTLB after a successful PAM tag check, it can be copied from another DCAS entry when an add-instruction references a valid entry or it can be cleared back to zero when a data cache line is evicted. This is similar for the *LWV* and *L1 DC way*, but an additional condition can occur for these fields. This happens when a PAM addition references the next sequential cache line, allowing us to copy the information from the *L1 DC N way* if *LWVN* is valid.

There is also some computation required for determining when information in the DCAS can be used and when to perform new tag checks. We have to check if the current instruction is either a PAM instruction or a memory instruction. Then we also need to know if there is a cache line change, page change, valid displacement and if either the *DWV*, *LWV* or *LWVN* is invalid. One optimisation that reduces the number of conditions to be checked is to assume a potential PAM instruction is valid if the *DWV* is set for its source register.

The L1 IC, IF stage and ID stage are very similar to the baseline, but require some small additions in order to track PAM instructions. We also see that Vivado sometimes optimises slightly different which in this case lead to one less logic LUT in the ID stage, but one more flip-flop in the IF stage in the improved pipeline compared to the baseline pipeline. This can happen when a signal is required in two stages and all its operands are available in both stages. The tool then has the choice to calculate it again (extra LUT) or to store it between stages (extra flip-flop).

The L1 DC has some small additions in both logic LUTs and Memory LUTs. The extra logic LUTs comes from multiplexing between using the provided way from the DCAS or the way from the tag check, and for comparing the next sequential cache line. The extra memory LUTs comes from switching to RAM64X1D which is dual ported and enables both tag checks to read from it simultaneously. These primitives contain two memory blocks each, and with eight of these used per way gives us an increase of $8 \times 4ways = 32$. The DTLB also has a small increase in logic which is simply for multiplexing the way between DCAS and the tag check.

7.2 Critical path

One important result is that the improved pipeline does not extend the critical path and therefore does not reduce the clock frequency we can run at. The critical path in the baseline implementation is the stall signal from the MMU in the memory stage. It starts from the ALU result in the EX/MEM pipeline register and goes through the MMU, DTLB, L1 cache before it is routed all the way back to the IF/ID pipeline register. There are many similar paths, such as stall signals for other parts of the pipeline, that are close to the same length. The blue path in Figure 7.1 shows one of the critical paths. Part of the reason this is the critical path is because of the distance the signal has to travel from component to component. Especially the last hop, from the L1 DC to the IF/ID register is very long.

Table 7.2: Improved pipeline resource usage.

Component	Logic LUTs	Memory LUTs	Flip-flops	BRAM	DSP
Main memory	264	0	8	64	0
L1 IC	407 (+8)	552	130	0	0
L1 DC	1849 (+53)	576 (+32)	514	0	0
DTLB	258 (+7)	8	296	0	0
IF stage	17	0	105 (+9)	0	0
ID stage	816 (-1)	0	1141 (+8)	0	0
EX stage	4056 (+1953)	23	4582 (+941)	0	4
MEM stage	87	0	71	0	0
Other	94	0	161	0	0
Total	7848 (+2020)	1159 (+32)	7008 (+958)	64	4

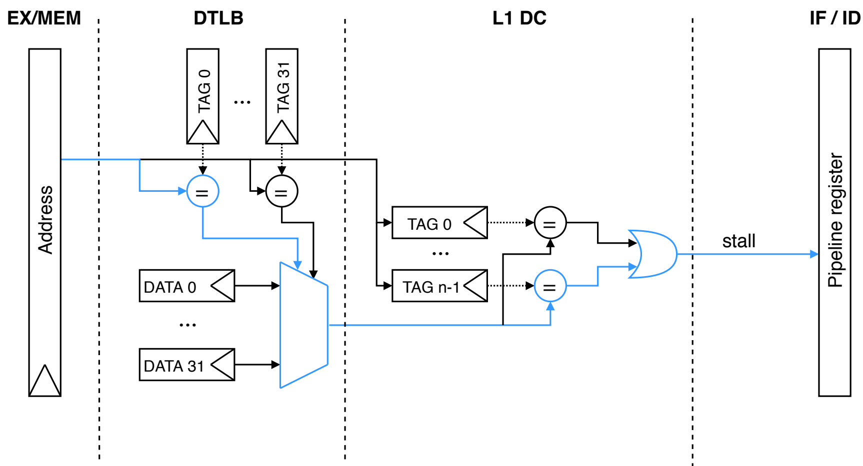


Figure 7.1: The blue lines indicate the critical path in the baseline implementation.

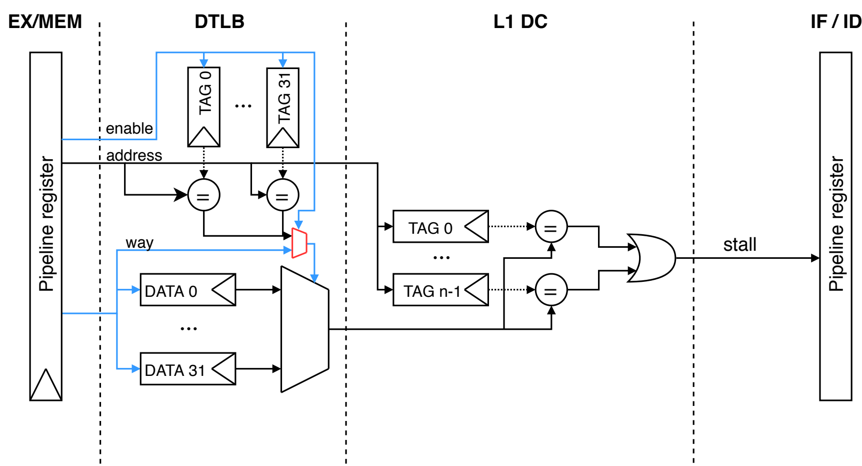
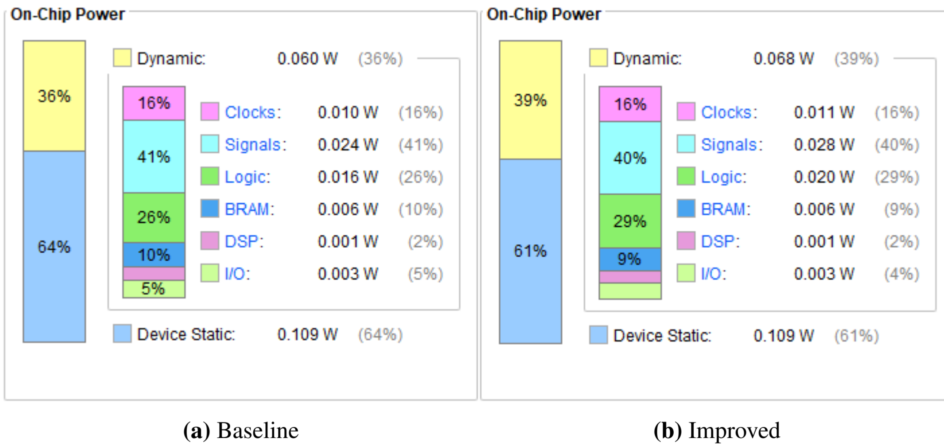


Figure 7.2: The blue lines indicate the additions for the improved pipeline.

The improved solution does not alter the critical path, but adds some logic in both the DTLB and L1 DC that is very close to it. The changes can be seen in blue in Figure 7.2. The extra logic for the L1 DC is not shown since it relates to the data arrays, however the change is exactly the same as for the DTLB. Two additional signals are added to both the DTLB and L1 DC. One signal is the stored way from a PAM instruction, and the other is whether this way is valid. If a valid way is provided, then only the selected data array is read. The *enable* signal works both as a method to select that no tag check should happen, and for multiplexing what way to select for the data arrays. The *enable* and *way* signals are decided in the execute stage, and are therefore not increasing the critical path. The multiplexer marked in red is for selecting either the stored way if it is valid, or to rely on the way from the tag check. This could potentially increase the critical path slightly since it is an additional step. However, during implementation in Vivado, this is optimised away into already existing LUTs in the tag comparators or the large data array multiplexer.

Both the DTLB and L1 DC also has signals back to the DCAS structure located in the execute stage. These signals are also close to the critical path in length since they also depend on the tag check being complete. Fortunately the distance they have to travel is shorter compared to the stall signal and they are therefore fine.

There are also many additions to the execute stage in the improved version. The longest path in the stage increases slightly since we need to wait for the carry out signal to check if we cross into the next cache line or page. This does not require us to wait for the ALU to fully complete its operation, so we can start selecting what way (if any) to pass on early. Additionally, the execute stage is shorter in general than the memory stage so there is still room before we hit reach the length of the critical path. In the baseline implementation, the memory operation signal would select when to enable all tag and data arrays in the DTLB. In the improved pipeline, there is more fine grained control, with being able to select only one data array to be read and no tag checks when we have a valid DTLB way, but no valid L1 DC way.



(a) Baseline **(b)** Improved

Figure 7.3: Power consumption for both pipelines.

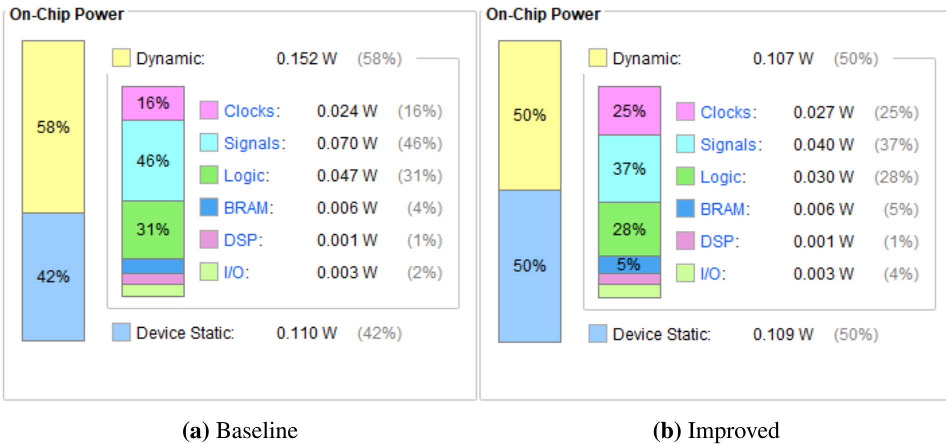
7.3 Power estimation

Perhaps the most important question we are trying to answer in this thesis is whether the DCAM technique leads to increased energy efficiency or not. Unfortunately, answering this question is not easy due to how the various components are implemented using different primitives. As mentioned earlier, the distributed RAM blocks that is used in the DTLB and L1 DC is much more energy efficient than using plain flip-flops. This creates a problem since the DCAS and DCAV are implemented using normal energy-inefficient flip-flops, whereas the DTLB and L1 DC accesses the technique is trying to prevent is implemented in more energy efficient primitives.

Using the power analysis tool in Vivado (Xilinx, 2013) on the first 10 000 instructions in the *dijkstra* benchmark produces Figure 7.3a for the baseline implementation and Figure 7.3b for the DCAM pipeline. Judging by these figures, the DCAM method consumes slightly more power even though it prevents many DTLB and L1 DC tag checks. As mentioned, the different primitives causes this result to not be entirely accurate.

If we force Vivado to use regular flip flops for both the L1 DC and DTLB the amount of resources required increases significantly, meaning signals have to travel further, thus reducing the clock frequency we can run at. However, this should provide a more accurate picture. Figure 7.4a and Figure 7.4b shows the new output from the power analysis tool. This shows that there are huge power saving benefits from using the DCAM technique. We see that the power consumed by the clocks increases from 0.024 W to 0.027 W from the baseline to the improved version. This is due to the extra flip flops in the DCAS and DCAV that needs to be clocked. The power required for both signals and logic is reduced from 0.070 W to 0.040 W and 0.047 W to 0.030 W respectively. The amount of power for BRAM, DSP, IO remains unchanged as is expected.

Forcing Vivado to use flip flops instead of more energy efficient primitives increases the resource usage quite drastically. Flip flop usage in the DCAM version goes from 7 008 (6 050 in baseline) to 18 879 (17 933 in baseline). Logic LUTs in DCAM goes from 7 848



(a) Baseline **(b)** Improved

Figure 7.4: Power consumption for both pipelines with forced flip flops.

Table 7.3: Memory accesses.

(a) DTLB

Type	Count	%
All ways	1 624	59.1%
Single way	69	2.5%
Bypass	1 054	38.4%

(b) L1 DC data arrays

Type	Count	%
All ways	261	20.3%
Single way	1 024	79.7%

(5 828 in baseline) to 22 143 (20 214 in baseline). This makes preventing DTLB and L1 DC tag checks and data reads a very good way to reduce overall power consumption.

7.4 Reduced DTLB and L1 DC tag checks

The main goal of the DCAM technique is to reduce tag checks in both the DTLB and L1 DC, and to reduce the number of ways we have to access during data access. Table 7.3a shows numbers for all memory accesses during the first 10 000 instructions in the dijkstra benchmark, which is the same we measured power from. We see that we can bypass the DTLB altogether 38% of the time. In an additional 2.5% of memory accesses, it is enough to access a single DTLB way to obtain the tag. The remaining 59% of memory accesses needs to perform a full tag check against all the ways.

Table 7.3b shows the percentage of load instructions which only had to read a single data array in the L1 DC. We see that almost 80% of the data accesses only had to access a single data array.

Chapter 8

Discussion

The results achieved in this thesis show that the DCAM technique provides significant reduction in energy usage during data access in the DTLB and L1 DC. For our test case we saw a 30% reduction in power. We also found that the technique can be implemented without altering the critical path, which means we can still run at the same clock frequency as a baseline implementation. The DCAM technique uses a significant amount of extra resources, $2\ 052$ extra LUTs and 958 extra flip flops. However, it is able to recoup the energy cost of the extra logic by preventing a large amount of tag checks and unnecessary data array reads.

This result is similar to the results from Stokes et al. (2019), which used CACTI to estimate the energy usage of the DCAM technique. They found that the data access energy is reduced by 51% on average compared to a baseline implementation. For the dijkstra benchmark they only observed a 22.9% reduction. The 30% reduction we observed is for the whole system, not just data access. Considering that almost two-thirds of the LUTs and flip flops in our implementation was spent on the L1 DC and DTLB, we expect our number to be slightly higher.

We also found that for almost 80% of the L1 DC data reads only a single data array had to be read. This number is unusually high, but can be explained by examining the benchmark we used. We only estimated the power and observed memory accesses for the first $10\ 000$ instructions in the dijkstra benchmark. This benchmark starts by reading in the nodes and costs from a file before the algorithm starts. The algorithm itself contains a lot of pointer dereferencing which is hard to memoise, but the loading of data from a file is very easy to memoise since it copies data sequentially from one location to another. Our result is therefore skewed towards an access pattern which the DCAM technique is very good at handling.

One important goal of this research was to verify that the DCAM technique could be implemented without increasing the critical path. We were able to do this due to optimisations performed by Vivado. In our own design there is still one multiplexer that has to be added to the critical path. The reason for this could be that Vivado was able to use a larger LUT with no extra delay or the optimisation moved the multiplexer around so it was no

longer on the critical path.

Chapter 9

Conclusions

In this thesis we have described how to implement a basic pipeline, supporting the MIPS I instruction set, a DTLB and L1 caches on an FPGA. We have then improved upon this baseline implementation and introduced the DCAM technique and successfully executed the dijkstra benchmark on it. The DCAM method is an approach to reduce the energy used by the L1 DC and DTLB by associating their tag check information with the base register of upcoming memory operations. This allows us to often avoid accessing the DTLB altogether or only access a single way to read the tag. We are also able to reduce the number of L1 DC tag checks and often access only a single data array in a set associative L1 DC. We are also able to dynamically detect which instructions are later used by memory operations, which means this technique will work without changes to ISA and does not require recompilation of binaries. We found that DCAM does not alter the critical path and provides a 30% reduction in power for the dijkstra benchmark compared to the baseline implementation.

Bibliography

Al-Zoubi, H., Milenkovic, A., Milenkovic, M., 2004. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In: Proceedings of the 42Nd Annual Southeast Regional Conference. ACM-SE 42. ACM, New York, NY, USA, pp. 267–272.
URL http://doi.acm.org/10.1145/986537.986601

Bardizbanyan, A., Själander, M., Whalley, D., Larsson-Edefors, P., 2014. Reducing set-associative l1 data cache energy by early load data dependence detection (eld3). In: Proceedings of the Conference on Design, Automation & Test in Europe. DATE '14. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 82:1–82:4.
URL http://dl.acm.org/citation.cfm?id=2616606.2616707

Dally, W. J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R. C., Parikh, V., Park, J., Sheffield, D., July 2008. Efficient embedded computing. Computer 41 (7), 27–32.

Digilent, 2019. Pynq z1 reference manual. Accessed: 2019-06-09.
URL https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual

Fatemi, O., Idris, F., Panchanathan, S., Aug 1994. Fpga implementation of the lru algorithm for video compression. IEEE Transactions on Consumer Electronics 40 (3), 337–344.

Gille, D., 2007. Study of different cache line replacement algorithms in embedded systems. Master's thesis, KTH Royal Institute of Technology.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., Brown, R. B., Dec 2001. Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538). pp. 3–14.

Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B. C., Richardson, S., Kozyrakis, C., Horowitz, M., Jun. 2010. Understanding sources of inefficiency in

general-purpose chips. SIGARCH Comput. Archit. News 38 (3), 37–47.
URL http://doi.acm.org/10.1145/1816038.1815968

Horowitz, M., Alon, E., Patil, D., Naffziger, S., , Bernstein, K., Dec 2005. Scaling, power, and the future of cmos. In: IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest. pp. 7 pp.–15.

Huang, W., Rajamani, K., Stan, M. R., Skadron, K., July 2011. Scaling with design constraints: Predicting the future of big chips. IEEE Micro 31 (4), 16–29.

Johnsson, L., Netzer, G., 10 2016. The impact of moore's law and loss of dennard scaling: Are dsp socs an energy efficient alternative to x86 socs? Journal of Physics: Conference Series 762, 012022.

Megalingam, R. K., Deepu, K. B., Joseph, I. P., Vikram, V., Aug 2009. Phased set associative cache design for reduced power consumption. In: 2009 2nd IEEE International Conference on Computer Science and Information Technology. pp. 551–556.

MIPS Technologies, 2016. Mips® architecture for programmers volume ii-a: The mips32® instruction set manual. Revision 6.06.

MIPS Technologies, 2019. Mips32 architecture. https://www.mips.com/products/architectures/mips32-2/, online; accessed: 2019-03-01.

Moreau, D., Bardizbanyan, A., Själander, M., Whalley, D., Larsson-Edefors, P., March 2016. Practical way halting by speculatively accessing halt tags. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1375–1380.

Price, C., September 1995. MIPS IV Instruction Set: Revision 3.2. MIPS Technologies.

Sembrant, A., Hagersten, E., Black-Shaffer, D., 2013. Tlc: A tag-less cache for reducing dynamic first level cache energy. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-46. ACM, New York, NY, USA, pp. 49–61.
URL http://doi.acm.org/10.1145/2540708.2540714

Stokes, M., Baird, R., Jin, Z., Whalley, D., Onder, S., 2018. Decoupling address generation from loads and stores to improve data access energy efficiency. In: Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems. LCTES 2018. ACM, New York, NY, USA, pp. 65–75.
URL http://doi.acm.org/10.1145/3211332.3211340

Stokes, M., Baird, R., Jin, Z., Whalley, D., Onder, S., 2019. Improving energy efficiency by memoizing data access information. In: Proceedings of the 2019 International Symposium on Low Power Electronics and Design. ISLPED '19. ACM, New York, NY, USA.

Xilinx, jan 2013. Vivado design suite useg guide: Power analysis and optimization.
URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_4/ug907-vivado-power-analysis-optimization.pdf

Xilinx, nov 2015. Multiplier v12.0: Logicore ip product guide.
URL https://www.xilinx.com/support/documentation/ip_documentation/mult_gen/v12_0/pg108-mult-gen.pdf

Xilinx, Sep 2016a. 7 series fpgas configurable logic block: User guide.
URL https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

Xilinx, oct 2016b. Divider generator v5.1: Logicore ip product guide.
URL https://www.xilinx.com/support/documentation/ip_documentation/div_gen/v5_1/pg151-div-gen.pdf

Xilinx, apr 2017. Block memory generator v8.3: Logicore ip product guide.
URL https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf

Xilinx, mar 2018a. 7 series dsp48e1 slice: User guide.
URL https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

Xilinx, dec 2018b. Vivado design suite user guide: Implementation.
URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug904-vivado-implementation.pdf

Xilinx, Feb 2019. 7 series fpgas memory resources: User guide.
URL https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

Zhang, C., Vahid, F., Yang, J., Najjar, W., Mar. 2005. A way-halting cache for low-energy high-performance systems. ACM Trans. Archit. Code Optim. 2 (1), 34–54.
URL http://doi.acm.org/10.1145/1061267.1061270

Zheng, Z., Wang, Z., Lipasti, M., 2014. Tag check elision. In: Proceedings of the 2014 International Symposium on Low Power Electronics and Design. ISLPED ’14. ACM, New York, NY, USA, pp. 351–356.
URL http://doi.acm.org/10.1145/2627369.2627606