Implementation of a Synthesizable MIPS core in SystemVerilog

Υλοποίηση και σύνθεση του πυρήνα του επεξεργαστή MIPS σε SystemVerilog

by
Stefanos Pistopoulos

A Thesis submitted to the Department of Electrical and Computer Engineering in partial Fulfillment of the Requirements for the

Diploma of Science in Computer and Communication Engineering

Supervisors
Dr. Nikolaos Bellas, Associate Professor
Dr. Christos Sotiriou, Associate Professor

Volos, Greece
September 2016

Acknowledgments

After completing this project, I could not begin this diploma thesis report without thanking my supervisor, Dr. Nikolaos Bellas. His helpful advices, trust and abiding support were more than valuable for me in order to fulfill my goal. I would also like to thank my family and especially Anastasia for being patient and always right there for me. Finally, I have to thank my colleagues and friends, Dimitris and Sakis, for their sincere friendship.

Abstract

MIPS (Microprocessor without Interlocked Pipeline Stages) is a microprocessor that was designed years ago but still remains one the most popular and ideal examples of RISC architecture. The majority of universities and higher education colleges instruct its instruction set architecture to students, introducing them into computer design and organization.

A lot of tools and hardware description languages were created allowing engineers to implement their designs in an easier and more efficient way. SystemVerilog is a hardware description language that is mainly used for design verification. However, it might as well be used for implementation of synthesizable complex designs.

The aim of this project is the design and implementation of a synthesizable MIPS core that can be used for integration in Field-Programmable Gate Array based systems. This thesis describes MIPS architecture, including memory hierarchy and floating-point unit, and presents a way of implementation using SystemVerilog hardware description language.

Περίληψη

Ο μικροεπεξεργαστής MIPS σχεδιάστηκε πριν πολλά χρόνια, εντούτοις αποτελεί ένα από τα πλέον δημοφιλή και ιδανικά παραδείγματα της αρχιτεκτονικής επεξεργαστών RISC. Η πλειοψηφία των διεθνών πανεπιστημίων και των ανωτάτων εκπαιδευτηρίων διδάσκουν την αρχιτεκτονική του, εισάγοντας τους φοιτητές στην οργάνωση και σχεδίαση υπολογιστών.

Πολλά εργαλεία και γλώσσες περιγραφής υλικού έχουν δημιουργηθεί επιτρέποντας στους μηχανικούς την υλοποίηση των σχεδίων τους με εύκολο και αποδοτικό τρόπο. Η γλώσσα περιγραφής υλικού SystemVerilog χρησιμοποιείται κυρίως για την επαλήθευση ψηφιακών κυκλωμάτων. Παρ' όλα αυτά μπορεί κάλλιστα να χρησιμοποιηθεί και για τη σχεδίαση και υλοποίηση συνθέσιμων πολύπλοκων σχεδίων.

Σκοπός αυτής της εργασίας είναι η σχεδίαση, υλοποίηση και σύνθεση του πυρήνα του μικροεπεξεργαστή MIPS, το οποίο μπορεί να χρησιμοποιηθεί σε συστήματα που βασίζονται στη χρήση μιας πλατφόρμας FPGA. Στην παρούσα διπλωματική εργασία θα περιγράψουμε την αρχιτεκτονική MIPS, περιλαμβάνοντας την ιεραρχία μνήμης και τη μονάδα κινητής υποδιαστολής, και θα παρουσιάσουμε τον τρόπο υλοποίησης του με τη χρήση της γλώσσας περιγραφής υλικού SystemVerilog.

Author's declaration

I hereby declare that all information and work in this thesis, titled 'Implementation of a Synthesizable MIPS core in SystemVerilog', have been obtained and presented by me in accordance with the regulations of University of Thessaly, academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have cited all material and results that are not original to this work.

Table of contents

List of figures

List of tables

List of abbreviations

MIPS	Microprocessor without Interlocked Pipeline Stages
RISC	Reduced Instruction Set Computer
ISA	Instruction Set Architecture
CPI	Clocks Per Instruction
IoT	Internet of Things
OS	Operating System
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
CPU	Central Processing Unit
FPU	Floating-Point Unit
CC	Clock-Cycles
PC	Program Counter
ALU	Arithmetic Logic Unit
RAM	Random Access Memory
MSB	Most Significant Bit
LSB	Least Significant Bit
RAW	Read After Write
WAR	Write After Read
WAW	Write After Write
NOP	No Operation
FSM	Finite-State Machine
IEEE	Institute of Electrical and Electronics Engineers
NaN	Not a Number
LRU	Least Recently Used
MRU	Most Recently Used
LFSR	Linear Feedback Shift Register
TRNG	True Random Number Generator
FIFO	First In First Out
EDA	Electronic Design Automation

Chapter 1

Introduction

This thesis presents the way that theory of MIPS instruction set architecture can be applied into practice by implementing a MIPS core. It can be divided in six major parts. The first part describes MIPS microprocessor. This part begins with a quick description of the instruction set architecture (Section 2.1) by referring the instruction types that MIPS supports and the way they are encoded. Section 2.2 continues with a presentation of pipelining technique and how it is applied and controlled in MIPS. In section 2.2.3 specifically we analyze all hazards that may arise by given examples and describe how we can confront these problems.

We continue in third chapter of this thesis by analyzing the way that CPU of MIPS was implemented and illustrate an abstract way that code was written. Section 3.1 describes how every pipeline stage was implemented and section 3.2 analyzes how we solved the structural and the control hazards, and how we implemented the data hazard unit to deal with all data hazards including the way that pipeline is stalled.

After discussing all the theory background of MIPS architecture and presenting the way that execution pipeline was implemented, we discuss about memory hierarch and caching technique. We begin chapter 4 by describing the basic organization of cache memories, what their purpose is, how memory is mapped to cache and how cache lines are replaced and written. In next section we describe how we implemented a parameterized set-associative cache and the way it interacts with CPU and main memory. Moreover, in section 4.2.3 we explain how cache is controlled and which cache line replacement policy was chosen to be implemented and why.

In next chapter we begin with a short description of floating-point numbers' representation formats including the IEEE-754 standard. In section 5.4 we discuss

about the floating-point instruction set and in section 5.5 we explain how we designed and implemented floating-point unit. In this section we present the way that floating-point arithmetic and data transfer instructions were implemented, about how we control FPU and what problems and hazards were faced and solved after integrating FPU into the core system.

Finally, in last chapter (Ch. 6) we come in conclusion, presenting the summary report of synthesis and discussing about future work that can be done.

Chapter 2

MIPS Microprocessor

MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC (Reduced Instruction Set Computer) instruction set architecture which was designed by John L. Hennessy at Stanford University. As a RISC instruction set architecture, it consists of a small non-complex instruction set, illustrating four underlying principles of hardware design; simplicity favors regularity, smaller is faster, good design demands compromise and make the common case fast [1]. It was developed by MIPS Technologies (formerly MIPS Computer Systems, Inc.) and currently is in possession of Imagination Technologies Group plc, a British-based company [3] [4]. MIPS instruction set is very popular because it is widely used for academic purposes, introducing students of Computer Engineering in Computer Design and Architecture. MIPS implementations were widely used on early commercial RISC CPUs and nowadays are mainly used in embedded systems, network, Internet of Things (IoT), digital home and mobile applications [3].

Early MIPS architectures were 32-bits and later versions were 64-bits following the progress of Computer Science. The first MIPS instruction set was MIPS I CPU instruction set which was introduced in 1985. It has been extended in a backward-compatible way; latest architecture versions include former ones. This allows processors that implement the latest architecture versions to run binary programs that are produced by previous processors [3]. The different revisions which have been introduced are MIPS I (1985), MIPS II (1990), MIPS III (1992), MIPS IV (1994), MIPS V (1996), MIPS32/MIPS64 (1999) and recent releases of MIPS32/MIPS64. There are also plenty of application-specific extensions.

2.1 MIPS Instruction Set Architecture

MIPS architecture supports up to 4 coprocessors. Coprocessor 0 is the system control coprocessor, coprocessor 1 is an optional floating point unit and coprocessors 2 and 3 are undefined optional coprocessors [3]. It has 32 general-purpose registers (Table 2.1) and another 32 floating-point registers each one of 32 bits. Usually a 32-bits data group is called word, representing the fundamental data unit in a computer.

Register Name	Register Number	Use	Preserved across a call?
$zero	0	The constant value 0	N.A.
$at	1	Assembler temporary	No
$v0-$v1	2-3	Values for function results and expression evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS kernel	No
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

Table 2.1 MIPS general-purpose registers.

2.1.1 Basic Instruction Set

The basic instruction set consists of arithmetic and logical instructions, control flow instructions, memory access instructions, coprocessor and other miscellaneous instructions [2] [3] [4].

Arithmetic and Logical instructions

The main purpose of using registers is to store data that processor needs in order to complete an operation. Processor uses data from two registers and produces the expected outcome storing it in the destination register. For instance, processor can

add the data from two registers and store the result in another register, implementing the operation of addition. All arithmetic and logical operations are executed by a unit called Arithmetic Logic Unit (ALU). MIPS instruction set architecture also supports operations between a register and a 16-bit immediate. This immediate value is treated as signed integer for arithmetic and control flow instructions and as unsigned for logical instructions.

Multiplication and division are more complicated arithmetic operations and require many steps to be completed. Hence, processors have independent multiplication and division units. Multiplying an m digit number by an n digit number results in an m + n digit, at most. Dividing two numbers leads to the production of a quotient and a remainder. Therefore, in MIPS architecture two 32-bit registers are used to contain the result. These registers are not general-purpose and are called HI and LO. Multiplication instruction produces a 64-bit result which is stored in these registers; low half is stored in LO register and high half is stored in HI register. Division instruction produces a 32-bit quotient and a 32-bit remainder that are stored in LO and HI registers respectively.

Data Transfer instructions

General-purpose registers are used for simple and small operations satisfying the first two principles of hardware design. This limits the amount and restricts the structure of the data that can be processed. That's why there is a crucial need for more data space which can be found in the memory system of the processor. Processor can use registers for computations and memory system for storing the results and the processed data. This design rule is implemented with data transfers between registers and memory. MIPS instruction set includes load and store instructions that transfer data between coprocessor 0 and memory system and between all coprocessors.

There are data transfer instructions for ranging size data transfers such as byte, half word and word, for treating loaded data as signed or unsigned and for aligned or unaligned memory access. This kind of data transfers use the base addressing mode which in particular means that an offset is added to a base register in order to get the memory access address. This offset must be sign extended before it is added to base register. MIPS architecture uses byte addressing, thus words must be aligned (multiples of 4) since MIPS addresses each byte. If a word memory access address is not a multiple of 4 then an address access exception will be thrown [3]. Data transfers can also be made between coprocessors and between coprocessors and memory system. Moreover, special data transfer instructions allow access to HI/LO registers.

Conditional branches and jumps

In computer science, programs are not executed in a straight-line way. Different parts of a code need to be executed depending on the input data and on the data created during the computation. The address of the instruction that is currently executed is maintained in a special 32-bit register named Program-Counter (PC). There are specific control flow instructions that guide the computer to execute another part of the instruction space such as branches and jumps. Branch instructions compare the data of two general-purpose registers and jump to an address depending on the comparison result whereas jump instructions unconditionally jump to that address. MIPS architecture supports PC-relative conditional branches and unconditional jumps. Jump addresses can be given in a pseudo-direct way using 26 bits for the jump target or in an absolute way via a register that contains the jump target. There is also support for saving a return link address in a general purpose register for subroutines ($ra).

2.1.2 MIPS instruction representation

Basic MIPS instruction set was discussed above, and now we must analyze how processor distinguishes and handles these instructions. Every instruction is a 32-bit binary representation. This word is separated in a few bit fields. These fields define the instruction category, the immediate values, the branch and jump offsets, the source and destination registers, the instruction operations and the binary shift bit amount (for constant shift operation instructions). There are three specific binary instruction formats: the R-type, the I-type and the J-type format.

R-type format

In R-type instructions all data values that are used by these instructions are maintained and stored in general-purpose registers. R-type instructions do not require immediate value, jump target offset, memory address displacement or memory address to specify an operand. R-type format includes all ALU instructions except of immediate ALU instructions, register-direct jump instructions and HI/LO data transfer instructions. Figure 2.1 illustrates the R-type instruction format.

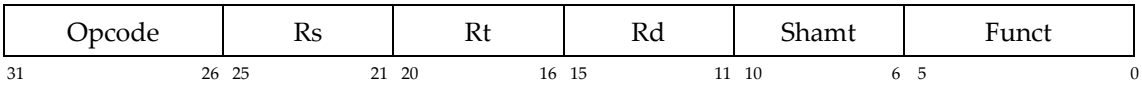

Figure 2.1 R-Type format

We shall now describe what these fields represent:

- **Opcode** is the operation code of the instructions
- **Rs** indicates the first source register
- **Rt** indicates the second source register
- **Rd** indicates the destination register
- **Shamt** indicates the shift amount for constant shift instructions
- **Funct** specifies the ALU and shift operation

Every type of instruction has its own opcode distinguishing it from others. All R-type instructions have opcode 000000. The funct field specifies which operation must be made. In all constant shift instructions, Rt field indicates the register that contains the word which will be shifted and specifically in variable shift instructions Rs field indicates the shift amount value. Rs field also specifies the register that holds the jump address in register-direct jump instructions and the data that is going to be stored into HI/LO registers in HI/Lo data transfer instructions. Examples of how R-type instructions are binary represented are given in Figure 2.2.

	Opcode	Rs	Rt	Rd	Shamt	Funct
ADD $8, $7, $3	000000	00111	00011	01000	00000	100000
SRL $14, $8, 9	000000	00000	01000	01110	01001	000010
SRLV $2, $1, $3	000000	00001	00011	00010	00000	000110
JR $13	000000	01101	00000	00000	00000	001000
JALR $6	000000	00110	00000	11111	00000	001001
MFHI $9	000000	00000	00000	01001	00000	010000
MTHI $2	000000	00010	00000	00000	00000	010001
	31 26	25 21	20 16	15 11	10 6	5 0

Figure 2.2 Binary representation examples of R-type instructions.

I-type format

I-type instruction format (Figure 2.3) includes all the immediate arithmetic and logical instructions, all branch instructions and all coprocessors-memory data transfer instructions. All opcodes except 000000, 00001x, and 0100xx are used for I-type instructions.

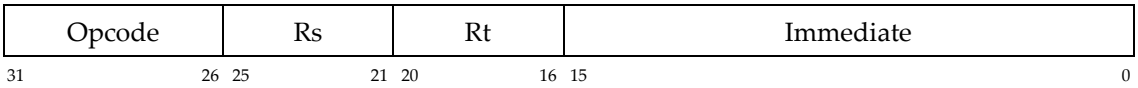

Figure 2.3 I-Type format

The immediate value is represented by a 16-bit field, thus immediate value can be defined in the signed values range of -2^{15} up to $+2^{15}- 1$. So, immediate arithmetic and logical instructions can have a minimum -32.768 and a maximum 32.767 immediate value. Furthermore, that means that we can't address ranges of memory larger than 32.768 bytes (or 8192 words). Branches use a PC-relative addressing mode because of the fact that branch target address is always near to the current program counter. Branch target address is computed in the following way:

$$PC = PC + 4 + \text{sign extend (immediate} \leftarrow 00)$$

This means that two zeros are inserted to the LSB end of the immediate value (similar to shift left logical by two times) in order to create a value divisible by 4. It is then sign extended to 32-bits and finally added to the next instruction address (PC + 4). Thus, the range of possible addresses is $PC - 2^{17}$ up to $PC + [2^{17} – 4]$. That means that we can branch 128 KB backward and almost 128 KB forward to the current PC.

Rs and Rt fields indicate the source and destination registers and immediate field contains the 16-bit immediate value. In coprocessor's 0 LW instructions the data contents of Rs register are added with the sign extended immediate value to produce the memory access address and the memory load data are stored in Rt register. In SW instructions the data of Rt register is going to be stored in memory. Binary representation examples of I-type instructions are shown below in Figure 2.4.

	Opcode	Rs	Rt	Immediate
ADDI $21, $3, 88	001000	00011	10101	0000000001011000
BEQ $17, $8, 2007	000100	10001	01000	0000011111010111
LW $14, 8 ($9)	100011	01110	01001	0000000000001000
SW $13, 92 ($6)	101011	01101	00110	0000000001011100
	31 26	25 21	20 16	15 0

Figure 2.4 Binary representation examples of I-type instructions.

J-type format

J-type format (Figure 2.5) consists of jump and jump-and-link instructions. In MIPS instruction set architecture designers have made a compromise that all instructions would have a word size and a 6-bit opcode field, satisfying the third principle of hardware design. Therefore jump instructions can have a 26-bit field for the jump target address. All J-type instructions use the opcode values 00001x.

Opcode	Address
31 26	25 0

Figure 2.5 J-Type format

Jump target address is computed by concatenating the 4 MSB bits of PC with the word-aligned 26-bit immediate value; hence the maximum address value can be 2^{26} - 1. In order to make the 26-bit word aligned we follow the same way as in branches, therefore the range of possible addresses is PC - 2^{28} up to PC + [2^{28} – 4], so we can have maximum jumps of 256 MB (or 64 million instructions).

$$PC = \{PC\ [31:28]\ ,\ (address \leftarrow 00)\}$$

This addressing mode is called pseudo-direct addressing because 4 bits of the PC are used to compute the address. For larger jumps we should use jump register instructions where the jump address is 32-bits. Jump register instructions can jump anywhere in the 4GB address space. We can see two examples of J-type instructions in Figure 2.6.

	Opcode	Immediate
J 19081966	000010	01001000110010101011101110
JAL 210388	000011	00000000110011010111010100
	31 26	25 0

Figure 2.6 Binary representation examples of J-type instructions.

2.2 MIPS Pipeline

Single cycle processors execute every instruction in one cycle, achieving an optimal CPI (Clocks Per Instruction) equal to 1. The clock period must be as large as the most time-consuming instruction. In basic MIPS architecture this instruction is LW instruction. Single cycle processors are easy to implement but they lack of performance because clock cycle is too long. Moreover, memory and every functional unit are not utilized efficiently. Therefore, multi cycle processors were implemented for a better overall performance. The idea was to make simple and small operations in a small clock cycle. Every instruction now is completed in more than one clock cycles and each instruction's individual operation is executed in one cycle. In addition, every functional unit can be used more than once per instruction in different clock cycles. Multi cycle processors can achieve better throughput but they have higher CPI measure.

2.2.1 Instruction pipeline datapath

Instruction pipelining is a technique implemented in order to increase the performance of multi cycle processors while maintaining a small clock period and a CPI measure close to 1. Pipelining implements a parallel execution of instructions' individual operations; multiple instructions are overlapped in execution [1].

MIPS instructions can take up to five steps in order to be executed, however as we will see not all instructions really need five steps to be completed. These steps also separate MIPS datapath (Figure 2.7) into 5 fundamental stages which have their own functional units and need one cycle to perform all the individual operations needed.

These stages are defined as:

1. Instruction Fetch (IF): An instruction is read from memory and PC is incremented.
2. Instruction Decode (ID): Fetched instruction is decoded into specific format fields and opcode is translated into control signals and read registers.
3. Execution (EX): An ALU operation is performed and jump/branch addresses are computed.
4. Memory (MEM): Read or Write data memory.
5. Write Back (WB): Result is stored in the destination register

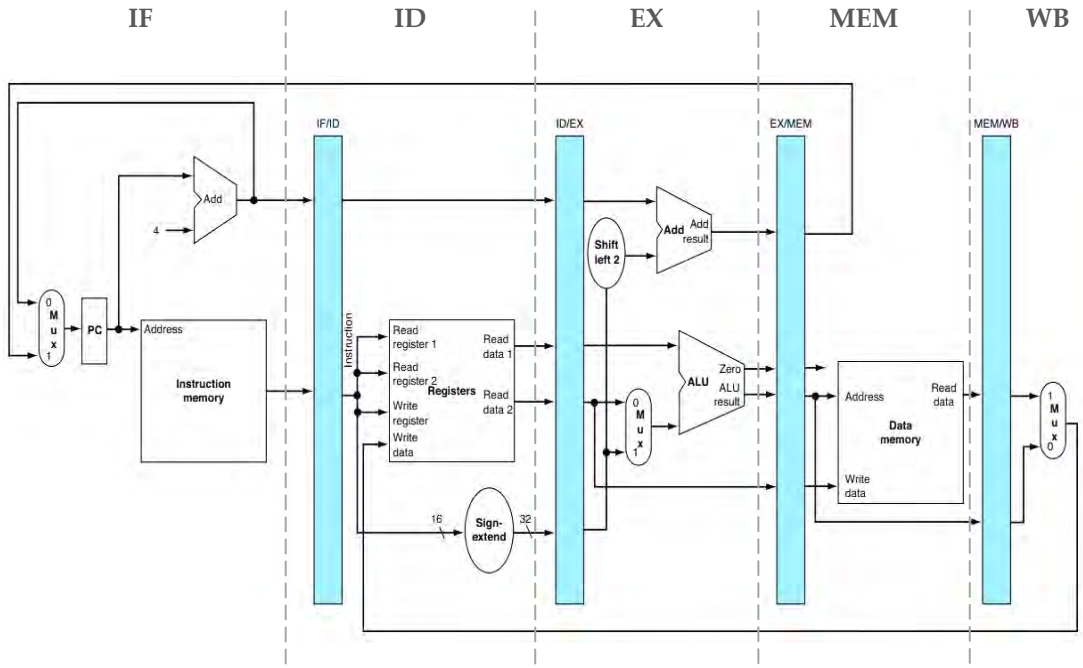

Figure 2.7 Basic Pipelined MIPS datapath. Pipeline stages are separated by the pipeline registers (in colour). Each pipeline register contains data useful to the next stage. The name of a pipeline register defines the stage transition.

We can see that in IF stage, the instruction that is read from instruction memory (indexed from the current PC) and the next instruction address (PC+4) are stored in IF/ID pipeline register. In ID stage IF/ID pipeline register is read to get the fetched instruction. This instruction is decoded into the corresponding format fields and register file is read in order to get the ALU execution operands. All necessary control signals (we will discuss about control signal later) are generated in the meanwhile. All the decoded information, the next instruction address, the control signals and the register file read data are stored in ID/EX pipeline register for usage in next stage. In EX stage, ID/EX pipeline register is read in order to get all useful data and control

signals. This stage executes the ALU operation for ALU instructions, computes the branch and jump target addresses and decides whether a branch is taken or not. The ALU result, the jump and branch target addresses, the second register file data read operand, the comparison result (Zero bit) of branch instructions and the control information are stored in EX/MEM pipeline register for usage in next stages.

In MEM stage the data memory is accessed, hence we need the memory address and the memory write data which are read from the EX/MEM pipeline register. Zero bit and jump/branch target addresses are also read in order to complete the jump/branch and transfer control to another part of the instruction space. The only operation left is to write back the data loaded from memory or the ALU result into register file, therefore these data and control information are stored in MEM/WB pipeline register from the EX stage and read in WB stage. We can record every instruction's stage at every clock cycle in a diagram like the one shown in Figure 2.8.

	1 cc	2 cc	3 cc	4 cc	5 cc	6 cc	7 cc	8 cc	9 cc
LW $8, 0 ($9)	IF	ID	EX	MEM	WB				
ORI $16, $17, 28		IF	ID	EX	MEM	WB			
ADD $14, $8, $9			IF	ID	EX	MEM	WB		
SW $21, 88 ($3)				IF	ID	EX	MEM	WB	
SUB $21, $21, $14					IF	ID	EX	MEM	WB

Figure 2.8 Multiple Clock-Cycle Pipeline diagram. This diagram illustrates the cycle by cycle execution flow of five instructions. Instructions are executed in a top-down way and clock cycle moves from left to right. MEM stage of ORI, ADD and SUB instructions is just for data propagation to WB stage, as these instructions do not access data memory.

2.2.2 Controlling pipeline

In ID stage the fetched instruction is decoded, generating all the necessary control signals. These control signals enable the functional units, control their operation, enable or disable their access and control all multiplexors preventing from pipeline flow errors. Control signals are generated depending on the opcode of the instruction; opcode value defines the operation and the format of the instruction. Control signals are propagated to the stage needed through the pipeline registers. There is a list of control signals in Figure 2.9 with a short description of their operation.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 2.9 Pipeline datapath control signals. The control unit generates three 1-bit signals (RegDst, ALUSrc and PCSrc) to control multiplexors, three 1-bit signals (RegWrite, MemRead and MemWrite) to control register file and data memory, and a 2-bit signal (ALUOp) for the ALU operation control.

The ALU is controlled by a 2-bit control signal named ALUOp which is also generated in ID stage. Depending on this signal, a 4-bit signal is generated, defining the ALU operation in EX stage. Figure 2.10 shows the way that ALU is controlled and Figure 2.11 shows the pipelined MIPS datapath including the control signals.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Figure 2.10 ALU operation control signals. In LW and SW instructions ALU computes the memory access address by adding the offset and the base register. In branch instructions, comparison is made by subtracting the values of the source registers. Hence, in LW, SW and BEQ instructions the generated ALU control signal does not depend on the function code. In all R-type instructions the ALU control input depends on the function code.

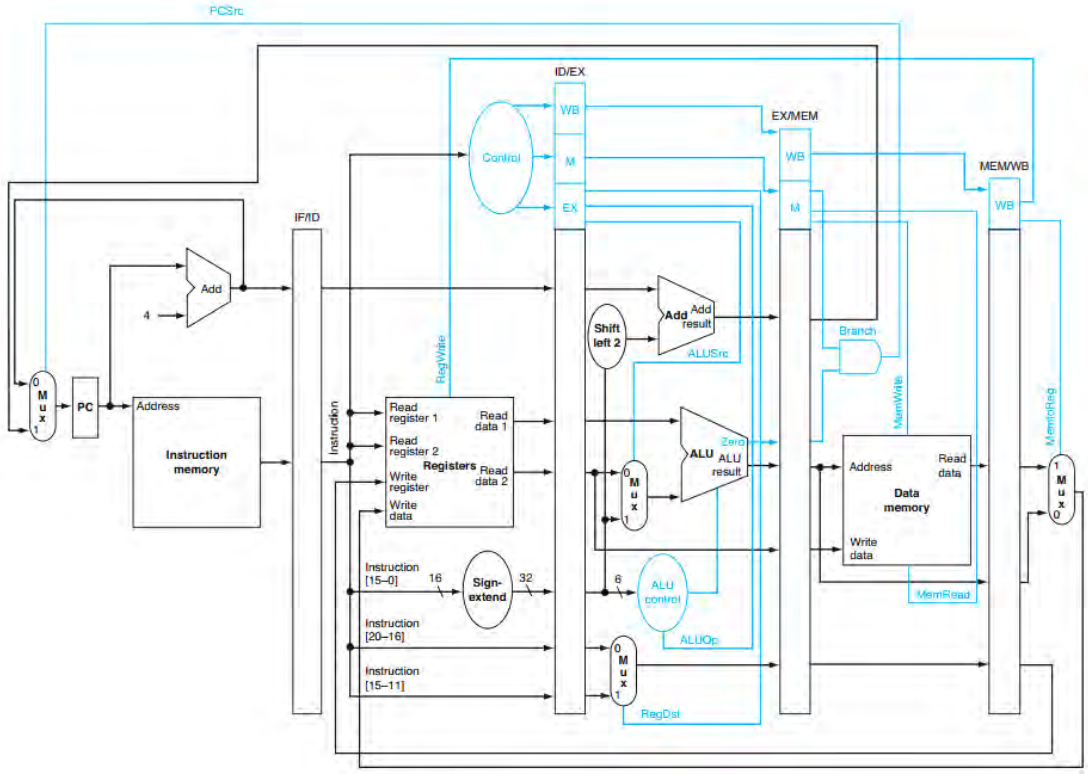

Figure 2.11 Pipelined MIPS datapath with control signals. Control signals are generated in ID stage by control unit and propagated to the next pipeline stages. PCSrc is now generated by performing the AND logic operation on Zero bit and Branch control signal.

PCSrc signal is always set to 1 throughout all stages, except in jump or branch cases, so that next instruction address is stored. In EX stage the RegDst, ALUOp and ALUSrc signals must have proper values; in immediate operand instructions ALUSrc must be 1 so that the 16-bit immediate operand is chosen. In MEM stage Branch signal must be 1 in case of branch instructions and when the branch is taken (Zero signal is set to 1) PCSrc sends the branch target address to the PC register. In addition, MemRead and MemWrite signals should be set properly for a memory read or write access. Finally, in WB stage RegWrite allows writes in register file when is set, and MemtoReg signal chooses between the memory read data or the ALU result.

2.2.3 Pipeline hazards

Despite the fact that every instruction is executed in order, there are cases that some instructions cannot continue with their execution because they need data that previous instructions produce. There are also cases that two or more instructions need the same functional unit at the same clock cycle and cases that an instruction is improperly fetched because of wrong control flow decisions; this means that the

correct instruction cannot be executed in the proper clock cycle. These cases lead to an imperfect pipelined instruction execution and are called structural, data and control hazards respectively [1].

Structural hazards

In MIPS microprocessor all functional units can be used once in a clock cycle. What if two instructions need to use a functional unit in the same clock cycle? This problem is called structural hazard and refer to the conflict of instructions for a functional unit usage. MIPS instruction set was designed to be pipelined so that structural hazards can be easily avoided. In addition, data memory and instruction memory are separated so that there is no memory structural hazard.

Data hazards

These hazards arise from the data dependency between two instructions. More specifically, an instruction cannot continue its execution because a data value that is produced by a previous instruction is not available in the right clock cycle. Consider the following examples with their pipeline diagrams:

	1 cc	2 cc	3 cc	4 cc	5 cc	6 cc
ADD $8, $7 ,$3	IF	ID	EX	MEM	WB	
MUL $16, $8, $9		IF	ID	EX	MEM	WB
SLL $14, $8, 9	IF	ID	EX	MEM	WB	
LW $20, 8 ($14)		IF	ID	EX	MEM	WB
LW $20, 21388 ($9)	IF	ID	EX	MEM	WB	
ADD $21, $13, $20		IF	ID	EX	MEM	WB
LW $20, 21388 ($9)	IF	ID	EX	MEM	WB	
SW $20, 1048($7)		IF	ID	EX	MEM	WB

We can easily notice that in the 4th clock cycle MUL instruction is going to use wrong data because the $8 register is written in 5th clock cycle. Therefore, we have a data dependency between ADD and MUL instructions. In second example, the memory access address is computed in the EX stage (4th clock cycle) but the correct data of $14 register are available in the 5th clock cycle. In third example, LW writes the memory

read data in $20 register in the 5th clock cycle while ADD instruction uses the $20 contents in the 4th clock cycle leading to wrong result. In last example we have a data dependency between MEM stages. This happens when a SW instruction is depended on a LW instruction. In this example, the write data of SW instruction are available in the end of the 5th clock. That means that wrong data are going to be stored in memory.

Then how are we going to deal with this problem? There are three ways to eliminate this problem. We can design compilers that re-order the instructions so that the depended instruction is executed at least two clock cycles later; register file is written in the first half of the clock cycle and read in second half, thus a register can be written and then read in the same clock cycle. Compilers re-order the instructions creating delay slots between the depended instruction and the instruction that produces the wanted result, and fill them with other instructions. The main problem of this approach is that there are much more data dependencies in a usual code than compilers can handle.

The second approach is just to stall the depended instruction for 2 clock cycles suspending the execution of the pending instruction and those that follow. This is an easy solution but the overall delay is very long. We can imagine these stalls like "bubbles". Pipeline bubbles practically mean that there is not work to do. These bubbles are implemented by inserting NOP (No Operation) instructions that simply do not do anything.

The best solution is to send the desired data, right after they are produced, to the stage that they are needed. The technique that data is send from one pipeline stage to another is called forwarding or bypassing. In our examples, we can send the result of the ADD instruction from the EX stage to the EX stage of the depended MUL instruction. In the second example, we can forward the result data from the EX stage of SLL instruction to the EX stage of the LW instruction. In the third example though, the desired data are available in the MEM stage of LW instruction. This means that we cannot only forward them to the EX stage of ADD instruction because they are available in the next clock cycle. Therefore, a stall is unavoidable. Data dependency of R-type instruction after a LW instruction is settled by stalling the pipeline and forwarding the desired data from the MEM stage of LW instruction to the EX stage of R-type instruction. In the last example, we can send the memory read data from the MEM stage of LW instruction to MEM stage of SW instruction so that the correct data are written in memory. We can see the data forwards in the Figure 2.12.

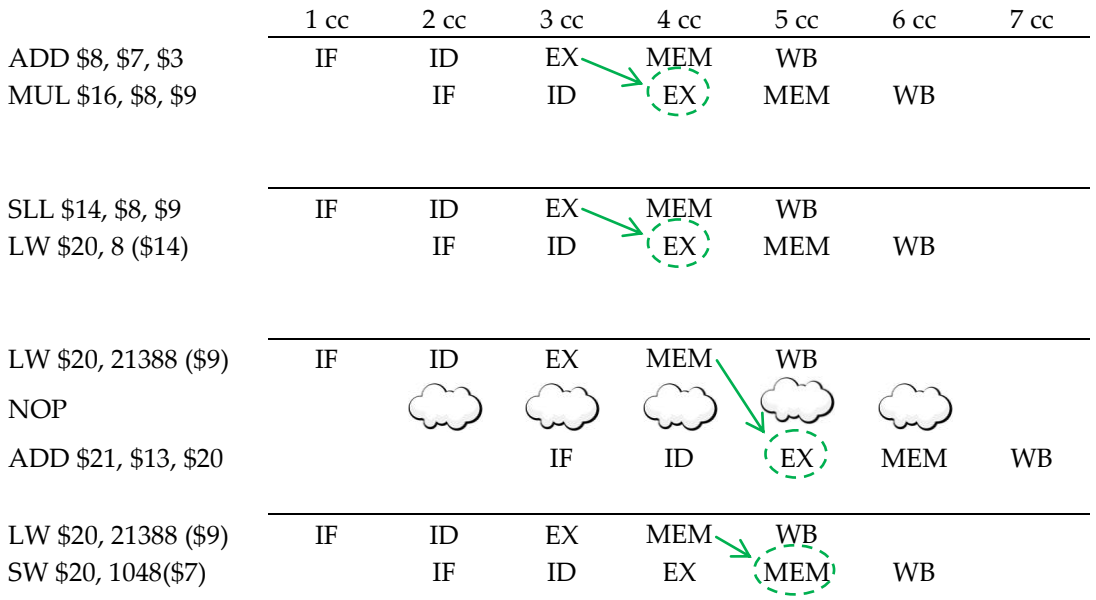

Figure 2.12 Forward technique in MIPS pipeline. Data forward is represented with a green arrow from the stage that forwarded data is produced to the stage that is received. We can forward data between EX stages, MEM stages and between MEM and EX stage. In LW – R-type data dependency, bubbles are inserted for a single cycle stall.

Control hazards refer to the hazards that arise from control flow instructions and the fact that branch decision is made in the EX stage. More specifically, until branch decision is made, two more instructions are fetched. That means that pipeline cannot always fetch the right instruction. Thus, we need to settle this problem. There are two main solutions; pipeline stall on every branch instruction and branch prediction. In the first solution (see Figure 2.13) we just have to wait until branch outcome is determined before fetching next instruction. In order to decrease pipeline stalls, the branch decision is taken in the ID stage. This enhancement demands more hardware for the branch condition check, the computation of the branch target address and for the PC update. The main problem of this approach is that stalling the pipeline on every conditional branch becomes unacceptable.

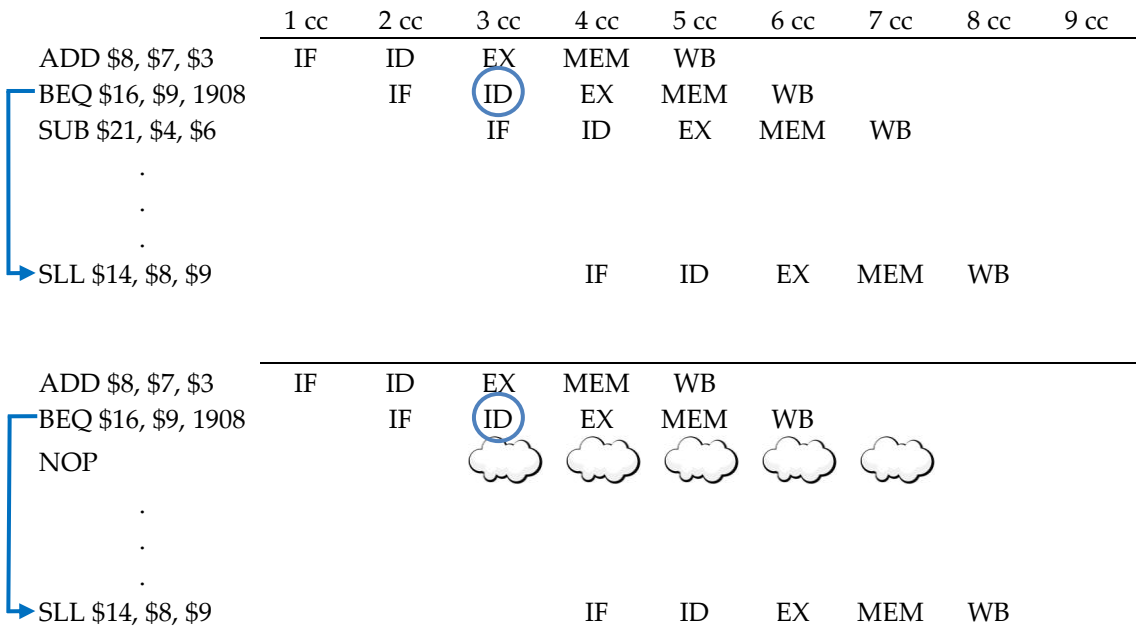

Figure 2.13 MIPS pipeline control hazard. The branch decision is made in the ID stage of BEQ instruction in 3rd clock cycle. We can see that SUB instruction is incorrectly fetched in the first example. In the second example pipeline is stalled preventing from fetching wrong instructions. Instruction is fetched right after branch outcome is determined.

In real life every human can make decisions depending on some predictions he makes. For example, someone decides to take an umbrella if the weather is cloudy because it is possible to rain. Prediction can also be applied in branch instruction. In branch predictions, we assume that every branch is always untaken. In this case if the prediction is correct, there is no penalty in the execution time and pipeline proceeds at full speed. With this approach we can noticeably decrease the pipeline stalls. For a more realistic approach, because not all branches are untaken, we can predict some branches as taken and some others as untaken. There are two methods for more realistic branch prediction, the static and the dynamic methods. Static branch prediction is based on typical branch behavior while dynamic records recent history of a small amount of branches assuming that future behavior will continue the trend. In all cases, when prediction is wrong, pipeline is stalled while fetched instructions are flushed. The correct instruction is then fetched. Figure 2.14 shows the pipelined MIPS datapath including the hazard detection and forwarding data units. For further information about branch prediction and a more detailed description of MIPS microprocessor you can refer to Computer Organization and Design 4th edition, written by David A. Patterson and John L. Hennesy. A list of MIPS instructions and their opcode values can be seen in Appendix.

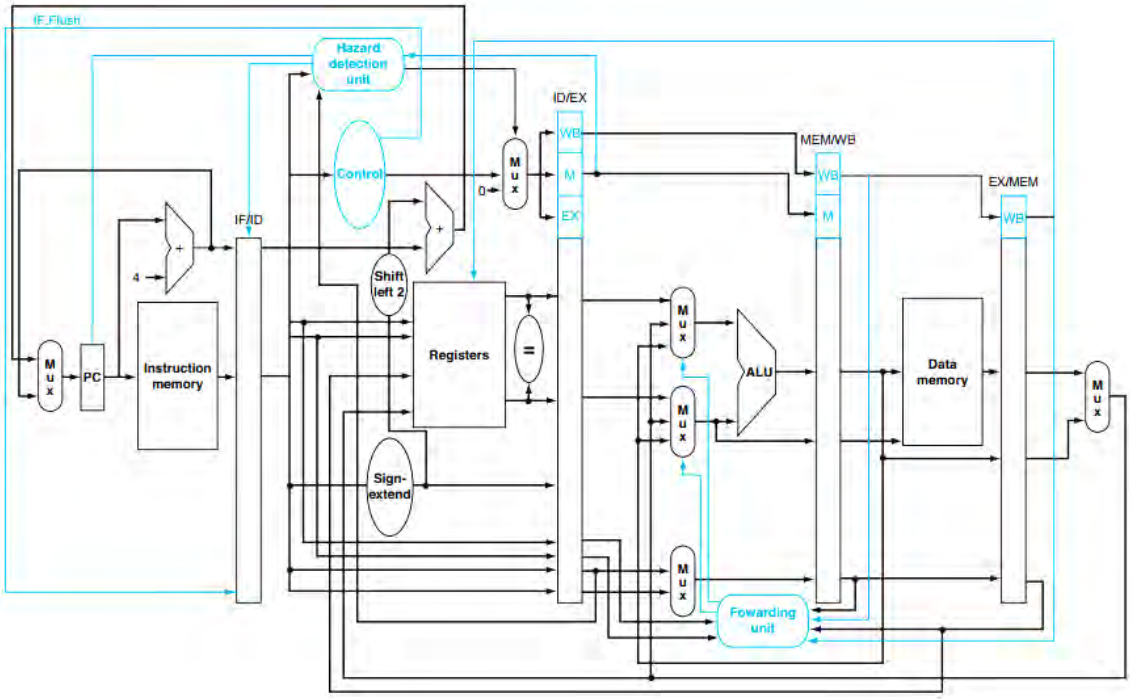

Figure 2.14 Pipelined MIPS datapath with hazard detection and forwarding units. This figure does not include figure's 1.15 control unit and connections for simplicity reasons.

Chapter 3

Implementation of MIPS coprocessor 0

We have discussed about MIPS coprocessor 0 in chapter 2 and now it is time to describe the way we implemented its architecture. The first thing we have to do is to implement the pipeline stages. Therefore, we need to implement all four pipeline registers and all the functional units such as instruction and data memories and the ALU. The WB stage is just a selection of write back data that are sent to decode unit; hence we do not implement it as a separate unit.

3.1 Implementing pipeline stages

MIPS coprocessor 0 begins an instruction execution by fetching it from instruction memory. We consider instruction memory as a black box for now and we will discuss about it in details later in memory hierarchy chapter (Ch. 4). Instruction memory has a 32-bit memory address input and provides a 32-bit memory data read output depending on this address.

3.1.1 Instruction Fetch unit

The fetch unit implements the instruction fetch logic. It has to read an instruction in every clock cycle from instruction memory, increment PC by 4 and set the new PC. The fetched instruction and the PC+4 values are stored in the IF/ID pipeline register. New PC can be the next instruction address (PC+4), the branch target address or the jump target address. Hence, the fetch unit should be implemented as shown below in Figure 3.1. We can also see the pseudocode of this unit in Figure 3.2.

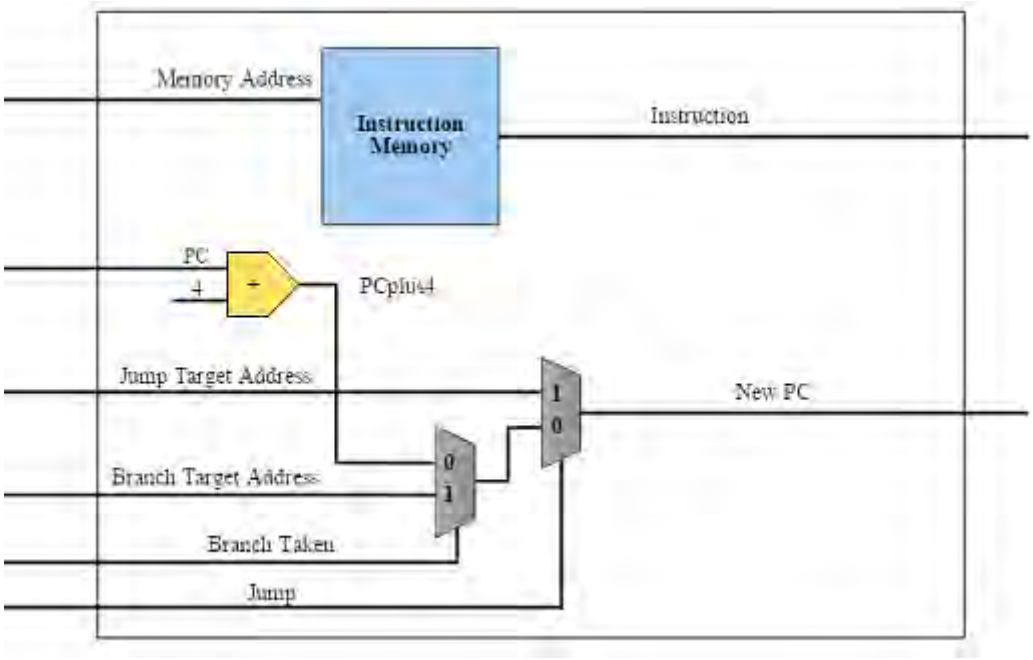

Figure 3.1 Fetch unit diagram.

```
1   Ports declare
2       in:
3           clock
4           reset
5           PC
6           jump target address
7           branch target address
8           branch taken
9           jump
10      out: instruction, new PC
11  end
12
13  Unit description
14      instantiate instruction memory;
15      read instruction memory at memory address;
16      select new PC;
17  end
```

Figure 3.2 Fetch unit implementation pseudocode

3.1.2 Instruction Decode unit

Decode unit is in charge of the instruction decode, the control signals generation and reading and writing the register file. This unit reads the fetched instruction from the IF/ID pipeline register, decodes it, and generates the pipeline and ALU control signals depending on the decoded instruction's opcode field. The decoded instruction's fields and the pipeline control signals are maintained in SystemVerilog

structures. The sign-extended immediate value is also stored as a decoded instruction's field. These structures are propagated to all pipeline stages through the pipeline registers.

Register file is written in the WB stage and must be written only when it is permitted. Therefore, decode unit should have a register file write enable input which is practically a control signal read from MEM/WB pipeline register. Register file can be written and read in the same cycle. That means that we must enable writing and reading a register in the same clock cycle. This can be implemented by checking the decoded source destination registers and the destination register. If one of the source registers is equal to the write back destination register that means we need to write and read in the same cycle, so we just have to send the write back data to the next stage. Furthermore, $0 register has always a zero value; we must take care of not writing data into $0 register. In link jumps we must write in register file the address of next instruction. Do not forget that the destination register of link jump instructions is register $31. A control signal that indicates a link jump will select $31 destination register. We can see the decode unit diagram and the implementation pseudocode in Figures 3.3 and 3.4.

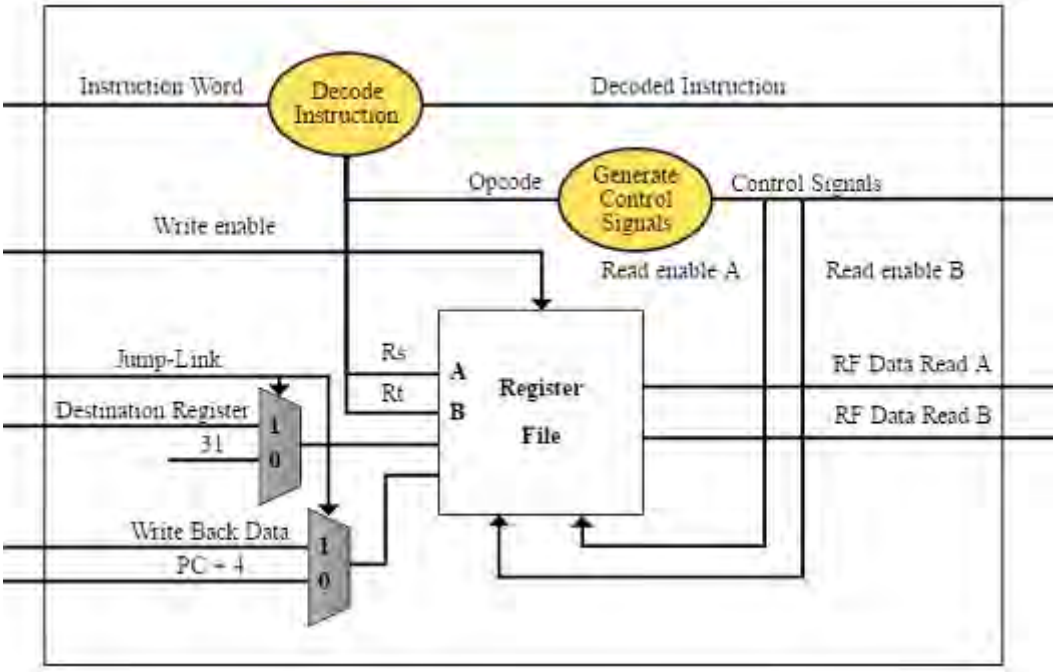

Figure 3.3 Decode unit diagram.

```
Ports declare
    in:
        clock, reset,instruction word
        RF write back data, destination register
        RF write enable
        jal, jalr
    out:
        RF read_data A, RF read_data B
        decoded instruction, control signals
    signal:
end

Unit description
    instantiate register file;

    decode instruction word;
    generate control signals;

    select RF write destination;
    select RF write data;

    on rst
        initialize register file;
        initialize RF read_data A, RF read_data B;
    rst:end

    on clk
        store RF write data in RF;

        RF read_data A = RF read Rs;
        RF read_data B = RF read Rt;

        if ( Rs read & store ) RF read_data A = RF write back data;
        if ( Rt read & store ) RF read_data B = RF write back data;
    clk:end
end
```

Figure 3.4 Decode unit implementation pseudocode

3.1.3 Instruction Execution unit

The execution's unit main purpose is to perform the ALU operation. The ALU operation is based on the ALUOp control signal and on the Funct field of the decoded instruction. The ALU operation operands and control signals are read from the ID/EX pipeline register. ALU operand B can be also the sign extended immediate value and it is controlled by the ALUSrc control signal. Execution unit also computes the branch and jump target addresses and decides whether the branch is taken or not. Branch is taken for BEQ instruction when branch on equal signal is true and Zero bit is set, whereas Branch is taken for BNE instruction when branch not equal signal is true and zero bit is not set. For these computations, PC+4 (read from ID/EX pipeline register) and PC values are needed. Based on the RegDst control signal, execution unit decides whether destination register is Rd or Rt. Figure 3.5 illustrates the diagram and Figure 3.6 the pseudocode of the execution unit.

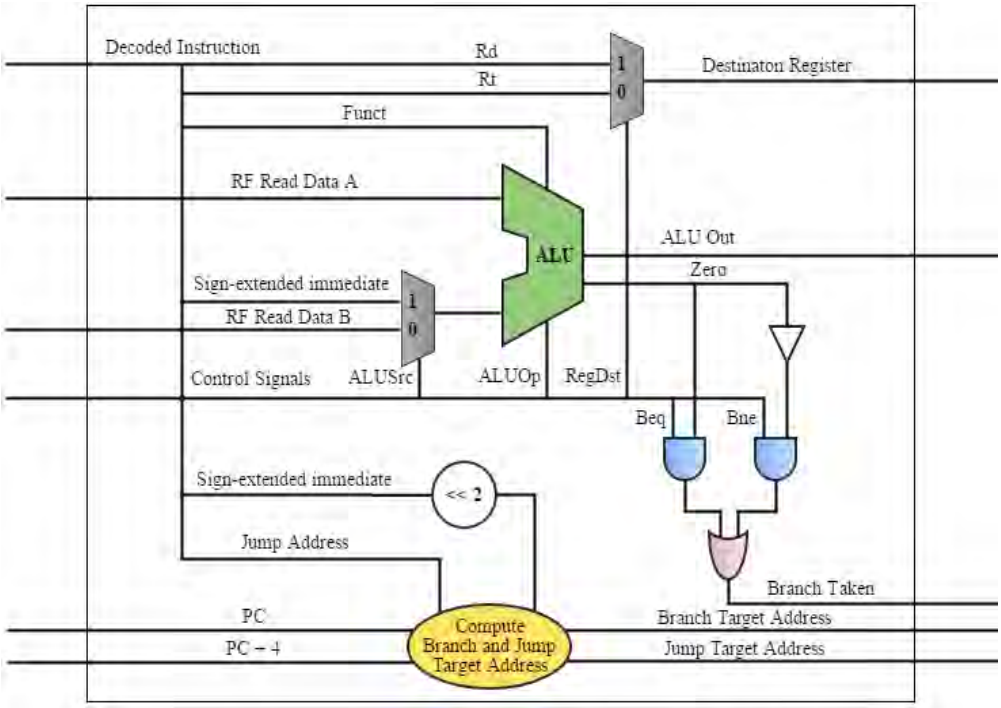

Figure 3.5 Execution unit diagram.

```
1   Ports declare
2       in:
3           PC
4           PC+4 from ID/EX
5           RF read_data A, RF read_data B
6           control signals
7           decoded instruction
8       out:
9           ALU out
10          destination register
11          branch taken
12  end
13
14  Unit description
15      select destination register;
16      select ALU operand B data;
17      set ALU Zero;
18
19      perform ALU operation;
20
21      compute jump target address;
22      compute branch target address;
23
24      decide whether branch or not;
25  end
```

Figure 3.6 Execution unit implementation pseudocode

3.1.4 Instruction Memory Access unit

This unit implements the MEM stage logic of the pipeline. In this stage we need only to access data memory. Data memory is read or written at a specified 32-bit address. The memory data object that is read or written varies in size which is controlled by control signals generated in the ID stage. This data size can be a byte, half-word or a word. Simultaneous memory read and write are not allowed; an error signal is generated otherwise. Memory write data is the register file operand B that is read from EX/MEM pipeline register; Rt register of SW instruction contains the store data. We consider data memory as a black box for the purposes of this chapter as we did for the instruction memory. In chapter 4 we will describe in details the memory hierarchy. Figures 3.7 and 3.8 show the diagram and the pseudocode of memory access unit.

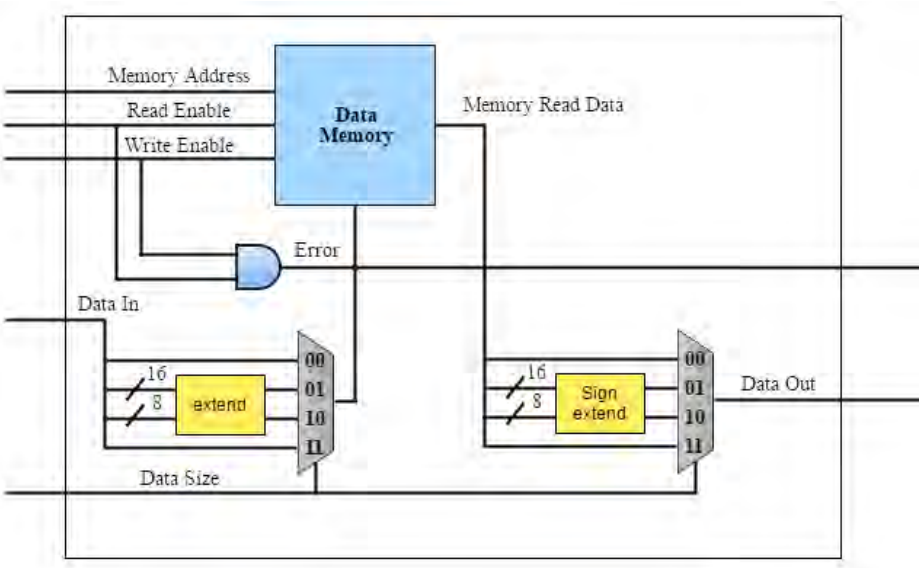

Figure 3.7 Memory Access unit diagram.

```
1   Ports declare
2       in:
3           clock,reset
4           read_en
5           write_en
6           data size
7           data_in
8           memory address
9       out:
10          data_out
11          error
12  end
13
14  Unit description
15      instantiate data memory;
16      check simultaneous read & write error;
17
18      on clk
19          if ( not error )
20              read memory when read_en;
21              write memory when write_en;
22          if:end
23      clk:end
24
25      select data out;
26  end
```

Figure 3.8 Memory Access unit implementation pseudocode

We now have to integrate those units in a bigger system that implements the pipeline. We also insert pipeline registers between all stages and PC register. These registers are just entities of SystemVerilog register data type.

3.2 Confronting Pipeline hazards

This datapath supports R-type signed and unsigned instructions, memory access instructions, I-type signed and unsigned instructions, branches and all jumps including register direct jumps. The next step is to take pipeline hazards into consideration. At first, we will deal with the control hazards and then with the data hazards. Branch instruction's decision is taken in EX stage. When a branch is detected, pipeline continues its execution, fetching the following instructions. When the branch is taken, two instructions have been fetched and need to be flushed. In this implementation, when the branch taken signal is set, we have to flush the IF/ID and the ID/EX pipeline registers. This can be made easily by setting all their fields to zero value. This action generates bubbles preventing from executing the instructions that were incorrectly fetched. As regards to the structural hazards, we follow the separate instruction and data memory model and take care that every functional unit is used by only one instruction in a clock cycle, therefore structural hazards are eliminated.

3.2.1 Data Forward unit

Data forward unit detects the data hazards and forwards the right data where they are needed. We must first deal with the data dependencies after an ALU instruction. ALU instructions produce their result in the EX stage. Therefore, we do not have to wait for the result to be written in register file; we can forward it right after it is produced instead. Moreover, ALU result is forwarded from the EX/MEM pipeline register, where it is stored, to the EX stage of instructions that need it to continue their execution. ALU data dependency can also exist between ALU instructions where the ALU result is produced two cycles before the execution of the second instruction, thus we need to forward ALU result data from MEM/WB to the EX stage of the depended instruction. Execution stage data dependency is detected by checking for register number equality. The first instruction's destination register must be at least equal to one source register of the depended instruction. We still have to consider some restrictions of MIPS architecture; we know that register $0 has always a zero value and that not all instructions write back data into register file. Register file write back is controlled by a write enable control signal. Hence, we need to forward only when the register file write enable signal and the destination

register, both read from EX/MEM or MEM/WB pipeline registers, are 1 and not the $0 register respectively.

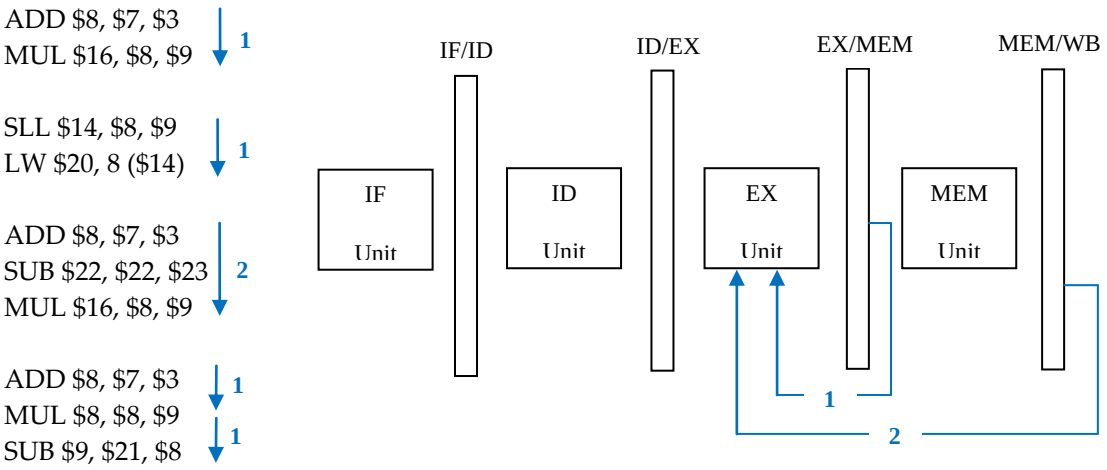

Figure 3.9 Data Forwarding paths Path 1 and 2 show the data forwarding to EX stage from EX/MEM and MEM/WB pipeline registers. Notice in last example that the most recent dependency has the biggest priority, therefore data is forwarded from path 1.

The second forward connection also resolves the load data dependency after a LW instruction. In this case though, we forward the memory read data from the MEM/WB pipeline stage to the EX stage of the depended instruction, instead of the ALU result. If there is a load data dependency between LW and the right-following instruction (load-use data dependency) we have to stall the depended instruction for one cycle and then forward the memory read data from MEM/WB pipeline register to the EX stage. Load-use data dependency is detected when the LW instruction is decoded. In the ID stage we have to check if the generated memory read signal is set, indicating a LW instruction. If it is set, we have to check then if there is a load-use data dependency. This can be achieved by checking for equality between the destination register of the decoded LW instruction and the source registers of IF/ID pipeline registers.

If the instruction that immediately follows the LW instruction is SW and the depended register is not the source register but the destination register of SW, we have to forward the load data from MEM/WB pipeline register to the MEM stage of SW instruction. We can detect this data hazard by checking the memory read control signal of MEM/WB pipeline register (indicates a LW instruction), the memory write control signal of EX/MEM pipeline register (indicates a SW instruction) and then check if the destination register numbers of MEM/WB and EX/MEM pipeline registers are equal which means that we have a dependency. The data forward

diagram contains one more connection from MEM/WB pipeline register to MEM stage as shown in Figure 3.10.

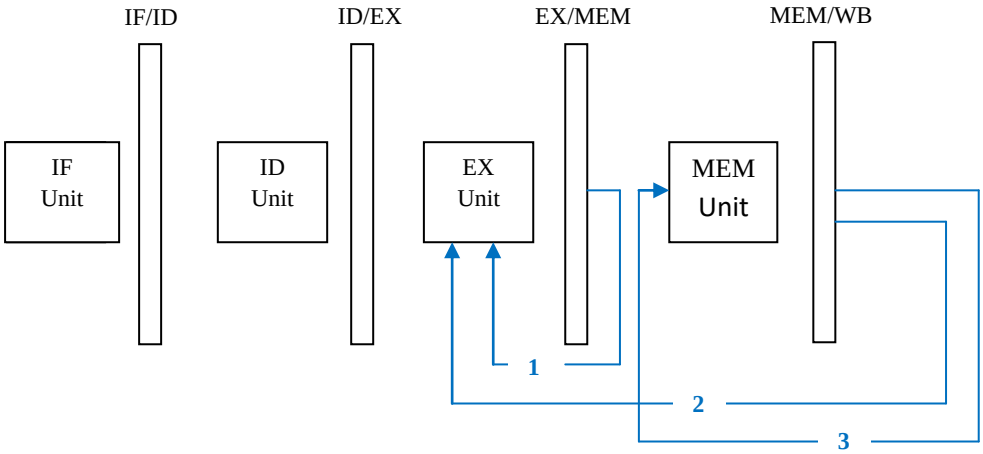

Figure 3.10 Data Forwarding paths including MEM to MEM data

We now have to implement the forward data unit. The inputs of this unit are the source registers read from ID/EX pipeline register, the destination register and the control signals read from EX/MEM and MEM/WB pipeline registers. It checks for the forwarding conditions and generates forward signals that control multiplexors. These multiplexors select the forwarded data in EX and MEM stage when there is a data dependency. The diagram of the forward unit and the forward conditions can be seen in Figures 3.11 and 3.12. Pseudocode is shown in Figure 3.13.

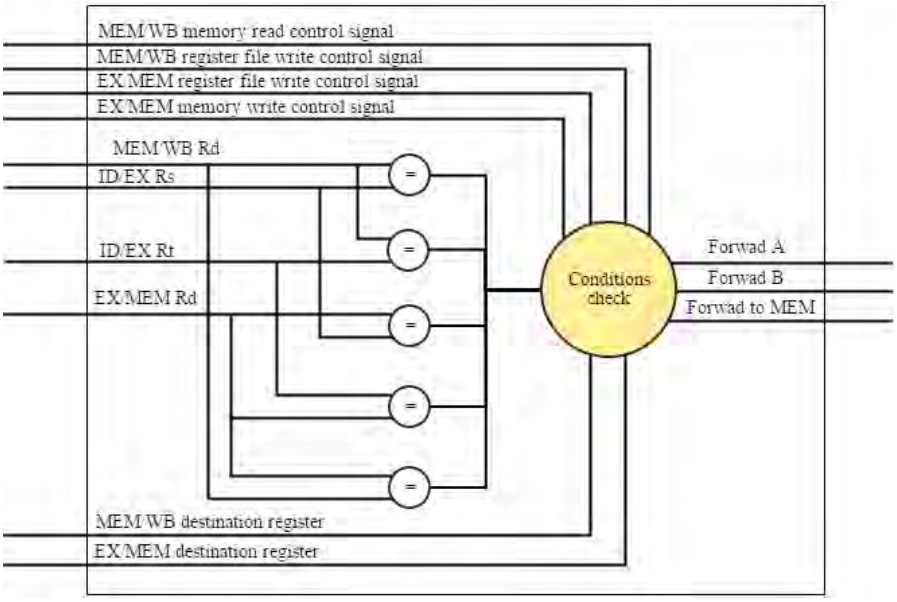

Figure 3.11 Data Forward unit diagram.

Forward to EX

EX hazard

- **if(** EX/MEM.RegWrite and (EX/MEM.RegRd ≠ 0) and (EX/MEM.RegRd = ID/EX.RegRs) **)**
  then ForwardA = 10
- **if(** EX/MEM.RegWrite and (EX/MEM.RegRd ≠ 0) and (EX/MEM.RegRd = ID/EX.RegRt) **)**
  then ForwardB = 10

MEM hazard

- **if(** MEM/WB.RegWrite and (MEM/WB.RegRd ≠ 0)
  and (MEM/WB.RegRd) = ID/EX.RegRs**)**
  and not (EX/MEM.RegWrite and (EX/MEM.RegRd ≠ 0)
  and (EX/MEM.RegRd = ID/EX.RegRs)) **)**
  then ForwardA = 01
- **if(** MEM/WB.RegWrite and (MEM/WB.RegRd ≠ 0)
  and (MEM/WB.RegRd) = ID/EX.RegRt**)**
  and not (EX/MEM.RegWrite and (EX/MEM.RegRd ≠ 0)
  and (EX/MEM.RegRd = ID/EX.RegRt)) **)**
  then ForwardB = 01

Forward to MEM

- **if(** MEM/WB.RegWrite and (MEM/WB.RegRd ≠ 0)
  and (MEM/WB.RegRd) = EX/MEM.RegRd**)**
  and (not MEM/WB.ReadMem and EX/MEM.WriteMem) **)**
  then ForwardToMem = 01
- **if(** MEM/WB.RegWrite and (MEM/WB.RegRd ≠ 0)
  and (MEM/WB.RegRd) = EX/MEM.RegRd**)**
  and (MEM/WB.ReadMem and EX/MEM.WriteMem) **)**
  then ForwardToMem = 10

Figure 3.12 Forward Conditions Forward to EX conditions control the data forward to EX stage and Forward to MEM conditions control the data forward to MEM stage. Forwarding to MEM stage is needed for data dependency to SW instructions. Notice that the second case of forwarding to MEM is a SW after LW destination register dependency. Data is forwarded from MEM stage of LW to MEM stage of SW. There is no need for pipeline stall.

```
Ports declare
    in:
        MEM.WB memory read_en
        MEM.WB register file write_en
        MEM.WB destination register
        EX.MEM memory write_en
        EX.MEM register file write_en
        EX.MEM destination register
        ID.EX Rs
        ID.EX Rt
    out:
        Forward A
        Forward B
        Forward MEM
end

Unit description
    initialize Forward A, Forward B;
    initialize Forward MEM;

    if EX(i) to EX(i+2) data dependency
        set Forward A, Forward B;
        forward MEM.WB ALU result to EX stage;
    if:end

    if EX(i) to EX(i+1) data dependency
        set Forward A, Forward B;
        forward EX.MEM ALU result to EX stage;
    if:end

    if EX(i+1) to MEM(i) data dependency
        set Forward MEM;
        forward MEM.WB ALU result to MEM stage;
    if:end

    if MEM(i+1) to MEM(i) data dependency
        set Forward MEM;
        forward MEM.WB mem read data to MEM stage;
    if:end
end
```

Figure 3.13 Data Forward unit implementation pseudocode

We now have to modify the execution and memory access units in order to use these forwarded data correctly. Note that execution unit can use the forwarded data as ALU operand A or ALU operand B. Hence, we have to insert multiplexors that select data for ALU operand A and ALU operand B based on the ForwardA and ForwardB signals. The same tactic is followed in the memory access unit; we insert multiplexor to select the memory write data, based on the ForwardToMem signal.

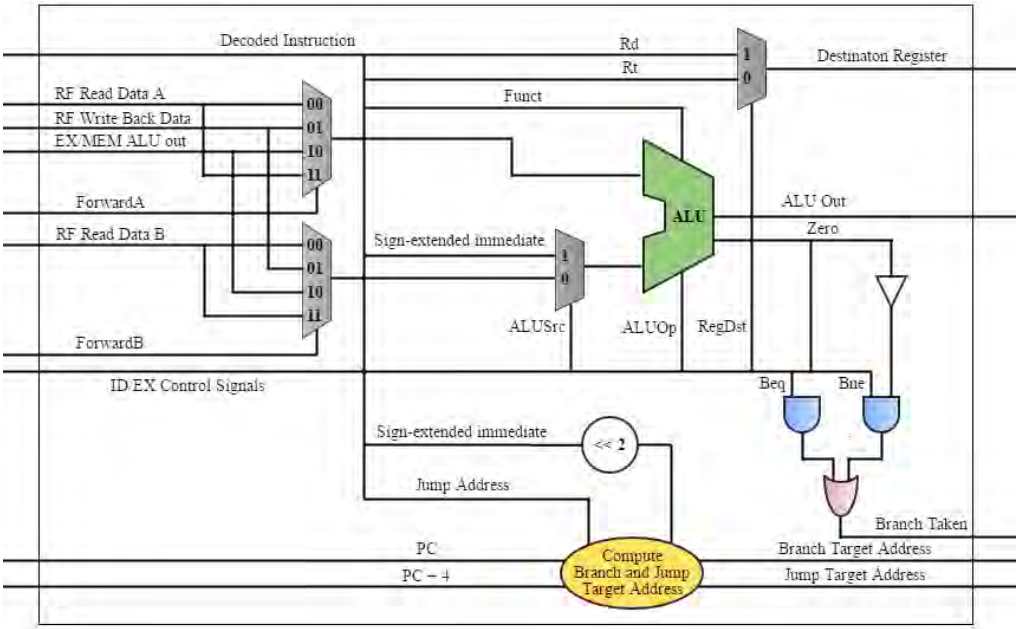

Figure 3.14 Modified Execution unit diagram. Data Forwarding to EX stage is now supported. RF Write Back Data can be MEM/WB ALU out or MEM/WB memory read data.

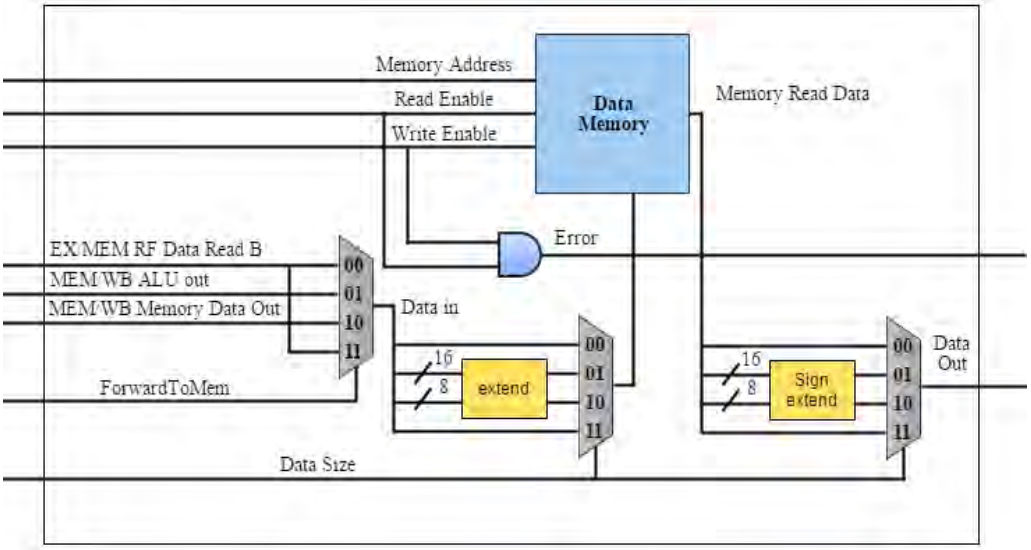

Figure 3.15 Modified Memory Access unit diagram. Data Forwarding to MEM stage is now supported.

3.2.2 Data hazard stall unit

We have implemented all the necessary data forwards but we also have to implement the stall unit so that load-use data dependency is resolved. Data is correctly forwarded to EX stage but we have to delay the depended instruction by 1 clock cycle. As mentioned before, the check for load-use data dependency is made during the ID stage of LW instruction. Stall unit needs the Rs and Rt fields of the fetched instruction that is read from IF/ID pipeline stage, the destination register of the decoded instruction which is read from ID/EX pipeline register and the memory read enable control signal read from ID/EX pipeline register. We also have to check if the depended instruction is SW, because in that case we don't have to stall the pipeline; data is forwarded from MEM stage of LW to MEM stage of SW. Therefore, the just generated memory write control signal is needed.

This unit generates an output stall signal when the ID/EX read memory control signal is 1 (indicating a LW instruction) and the currently generated write memory control signal is 0 (indicating that the depended instruction is not SW). At the same time, ID/EX destination register and at least one of the source fields of the instruction word must be equal. When stall signal is generated, pipeline must wait for one cycle. We implement pipeline stall by setting the control signals of ID/EX pipeline register to zero and preventing from PC update while the stall signal is set. The current instruction and the following instruction are decoded and fetched again. Instruction fetch unit must be also modified in order to read the instruction memory only when there is no stall signal. So, line 15 of the instruction fetch unit implementation pseudocode (Figure 3.2) is modified to:

```
read instruction memory at memory address when no stall;
```

Chapter 4

Memory System hierarchy

Till now we have assumed instruction and data memories as black boxes. In this chapter we will discuss about memory hierarchy and memory implementation. There are forms of data storage that vary in access time and size such as registers and RAM that CPU can directly access. These are processor components that enable the instruction's execution.

Registers are very small and very expensive components because accessing them must be very fast and multi-directional. On the contrary, RAM is a very big and at the same time very slow data storage component. Registers are used as data storage components in cases that fast access is needed and RAM memory in other cases. RAM is the main memory component of CPU; it is a very big and slower than registers memory component. The data produced by a program's execution is extremely larger than the data storage space that registers can provide, hence continuously writing and reading RAM has a drastic impact on processor's performance. Therefore, a problem of an effectively fast and large enough data storage component has arisen.

Engineers have come to a solution by designing a memory hierarchy model (Figure 4.1) which consists of several data storage levels; small and fast memory components are combined and utilized with big and slow memory components giving the impression of a single, very large and very fast memory. This solution is based on the observation that programs tend to use the same set of data items or nearby sets over and over again. The trend of using the same data item is known as temporal locality and the trend of using nearby data items is known as spatial locality. These types of data item reference locality form the principle of locality, a very crucial principle in computer science. In practice, memory system hierarchy is a set of data storage

components with different storage capacities, costs and access times. Memory hierarchy levels differ in speed and size; every level closer to CPU is smaller and faster. In order to improve processor's performance, one more memory level is placed between CPU registers and RAM. This type of memory is called Cache memory.

4.1 Cache memory

Cache memory is a smaller and faster than RAM memory component and is located in CPU. It is practically a very fast copy of RAM. Cache memory is designed to contain the data from frequently used memory locations, taking the advantages of locality principles. The size of cache memory (usually 4, 8 and 16 KB or MB) is much smaller than the main memory because it must be included in the processor chip and because cache memories are really expensive. The minimum amount of information that can be stored in cache is called block or line. A cache request means that processor needs data that are very likely stored in cache. If the cache memory contains indeed the requested data, we have a cache hit and request is quickly served. If the cache memory does not contain the requested data then we have a cache miss. When a cache miss occurs, processor requests the lower level of memory hierarchy (main memory) for retrieving the line containing the requested data. The critical problem of cache misses is that the lower memory level is much slower. That means that there is a penalty in memory access time because the requested data must be read from main memory. After lower memory level access, cache must be updated with the requested data and finally data must be delivered to processor.

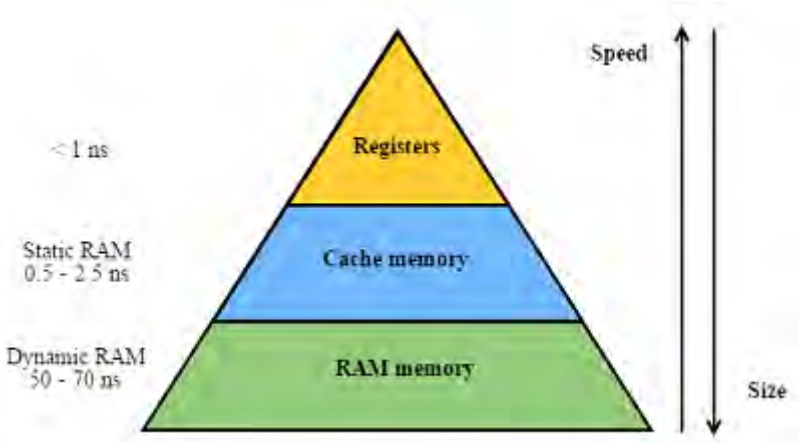

Figure 4.1 Memory System Hierarchy. Access time and cache size increase as we go to lower levels. Lower levels are also cheaper to implement. Data is copied between only two adjacent levels at a time [1].

4.1.1 Multi-level cache

In order to decrease the miss penalty, engineers have inserted additional levels of caching. This technique uses a multi-level cache so that the requested data is found in the next cache level avoiding the search in RAM. It is practically a cheap expansion to the cache with slower, cheaper and bigger additional lower cache levels, but still faster than the main memory. The multi-level cache can consist of three levels; L1, L2 and L3 levels. If a cache miss is occurred in L1 cache then the requested data is searched in L2 cache. If there is a hit in L2 cache, the requested data is stored in L1 cache and delivered to processor. If there is a miss, the requested data will be searched in L3 cache. If there is a miss in all three cache levels, the requested data will be retrieved from main memory with a bigger penalty though.

4.1.2 Cache memory mapping

There are three techniques used to map memory lines to cache lines. These techniques are direct mapping, fully associative mapping and set-associative mapping [1] [2] [11].

Direct mapping

In direct mapping each memory line can be mapped directly to one and only location in cache. This means that each location in RAM has one specific place in cache where the data will be held. Memory lines are mapped to cache lines using the following formula:

(Block address) modulo (Number of blocks in cache)

That means that memory line j is mapped to cache line j modulo 256 for a cache with 256 lines. In Figure 4.2 we can see the organization of a direct-mapped cache and the way memory address is translated.

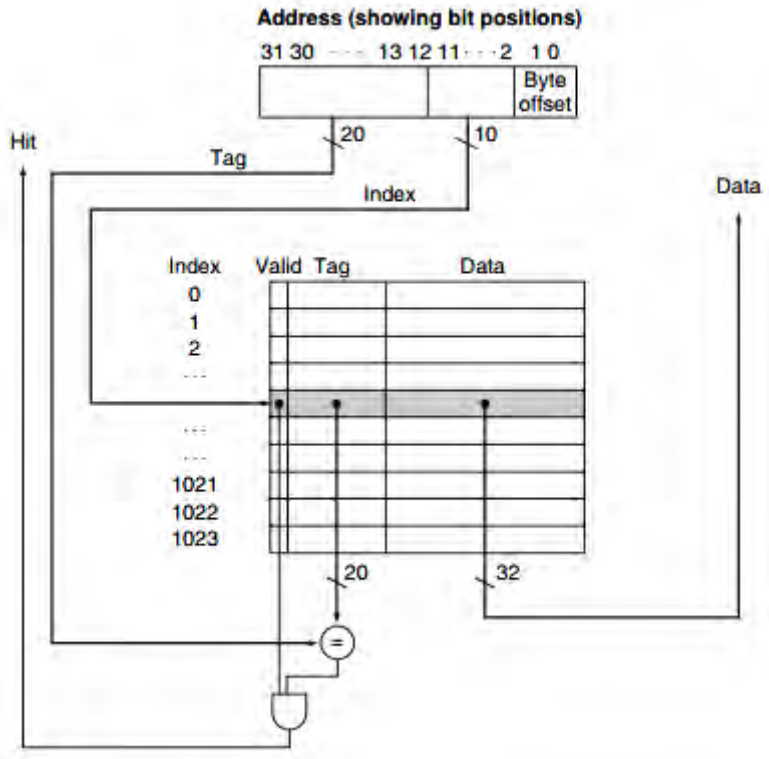

Figure 4.2 Direct-mapped Cache This cache has 1024 lines, which means that $\log_2 1024 = 10$ bits are used for index field. The line size is 32 bits, so we have 4 bytes in a cache line and byte offset field is $\log_2 4 = 2$ bits. The remaining 32 – 10 – 2 = 20 bits are the tag field bits.

The memory address is divided in three fields; the tag, the index and the byte offset field. The byte field defines the byte in the line. If the line size is M then the bits needed to specify the byte offset are $\log_2 M$. The index field defines the cache line. If cache has N lines then the index field has $\log_2 N$ bits. Tag field contains the rest part of the address and is used to check whether a word in the cache corresponds to the requested word. If a memory address is L bits then tag bits are L – M – N. There is also a valid bit entry. This flag indicates whether the cache line contains valid data or not.

A cache entry search is made by comparing the tag field with the tag entry in the line that is indexed by the index field. If valid bit is set and data is found request is served and hit signal reports a cache hit. It is the simplest way of mapping but it has a high miss rate. One solution to the high miss rates is to increase the cache line size [12] [13]. Increasing the cache size though, results in less cache line entries. Therefore, the competition for cache line entries is getting bigger. This can result in a very quick replacement of cache lines, which is practically opposed to the locality principles,

limiting the benefits in the miss rate. Hence, reasonable cache line sizes are chosen. Figure 4.3 shows the relation between cache miss rate and cache line size.

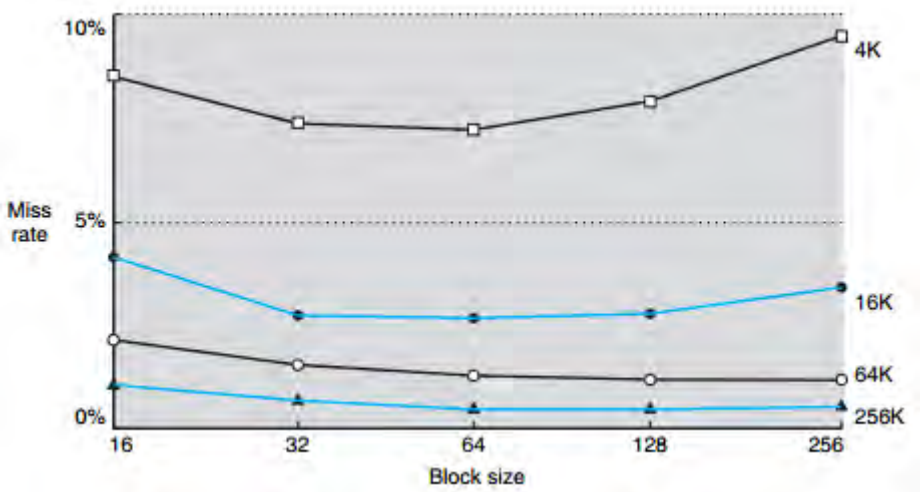

Figure 4.3 Cache miss rate and cache line size relation. This diagram is based on the results of SPEC92 benchmark. Note that increasing the cache line to a size nearly equal to the cache size increases the miss rate.

Fully associative mapping

Direct mapping has a major contention problem. Two memory lines can be mapped in the same cache line, so one must take the place of the other even if cache is really empty. Associative mapping technique deals with this problem; any memory line can be mapped anywhere in cache. This technique has the best hit rate because there is less competition for cache entries but another problem arises. How can we locate the cache line where the data is stored since it can be stored in any cache line? This means that we have to check all cache entries in order to locate the requested data. Therefore, fully associative mapping improves cache utilization, as all cache lines will be used, but at the expense of speed. Parallel check of all cache tags can be a reasonable tactic to speed up the check but it is expensive and requires more circuit complexity. This technique is feasible only for very small cache memories [10]. The index field is no more useful, therefore memory address is divided into tag and byte offset fields.

Set-associative mapping

The set-associative mapping is practically a compromise technique. It is a combination of the direct mapping and the fully associative techniques. The cache is now divided into sets and every set contains multiple cache lines. In this technique,

every memory line is mapped to one and only set in cache (Direct mapping) but it can be mapped to any line in the set (Fully associative). The cache lines that a set consists vary from 2 to 16. This technique is surely cheaper and faster than the fully associative mapping, has a lower miss rate, but it is yet slower than direct mapping. The set that contains a requested memory line is given by the formula:

(Block address) modulo (Number of sets in the cache)

A cache that contains M lines in total is called N-way set associative cache for some N that divides M. Every set consists of N lines and is indexed by the index field of memory address. Direct and fully associate mapping techniques are extreme versions of set-associative technique. Direct mapping is the 1-way set associative case; every set has one way, hence one cache line, so every memory line is mapped to one cache line. On the other hand, fully associative is a set-associative case where N is equal to the lines that cache contains in total. In that case cache consists of one set and every memory line can be mapped anywhere in this set. Even though direct-mapped caches have more conflict misses due to their lack of associativity, their performance is still better than set-associative caches when the access time costs for hits are considered [13]. Figure 4.4 illustrates a 4-way set-associative cache.

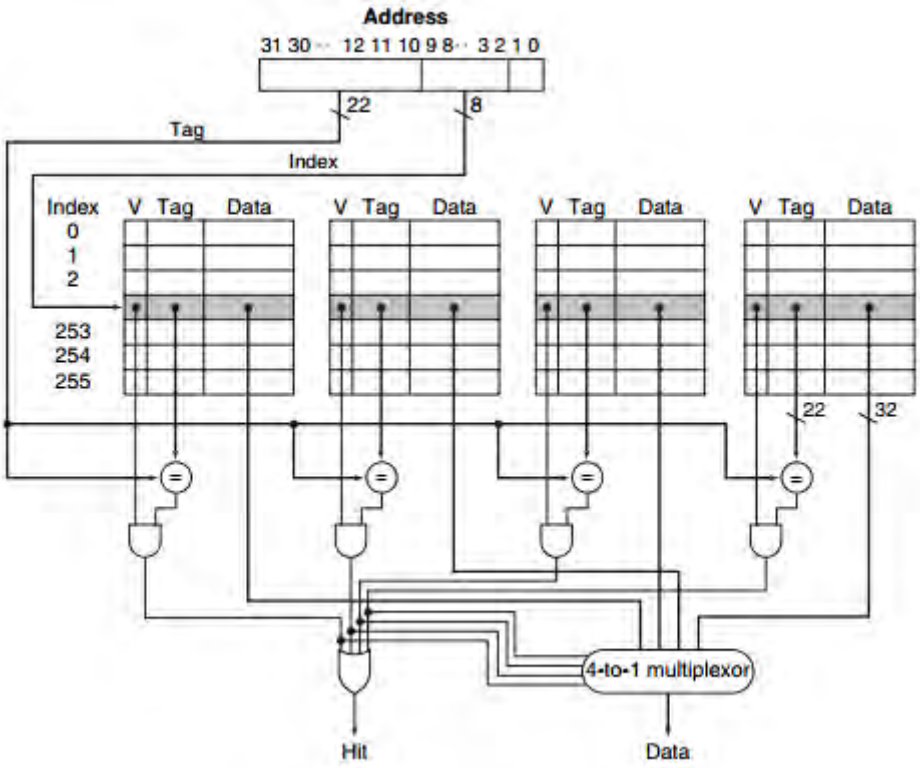

Figure 4.4 Organization of a 4-way set-associative cache. Index field is used to select the set. Every tag in a set is compared to the given tag.

4.1.3 Cache line replacement algorithms

When a cache miss occurs, the requested data must be stored in cache after it is retrieved from the lower level. In direct mapped caches, there is only one possible cache line each time that will be replaced if occupied. In set-associative caches if all the possible cache lines in the set are occupied, where will the requested data be stored? The answer is given by an algorithm which defines the way that cache lines are replaced. There are several cache line replacement algorithms because there is no common wisdom about the best one [14]. We will discuss about Bélády's algorithm, LRU, MRU, FIFO and random replacement algorithms. All these algorithms except Bélády's and random replacement utilize a history of the references in cache. Note that in fully associative caches, all cache lines are candidates for replacement.

Bélády's algorithm

Bélády's algorithm is an optimal replacement algorithm. In this algorithm the cache line that is chosen to be bumped out of cache is the one that will not be used for the longest time in the future. This algorithm is practically impossible to be implemented because we are not able to know when a cache line will be needed in the future. In some cases we can predict the behavior of a program but still it is not enough. Bélády's algorithm is a very good metric though; we can compare the results of other replacement algorithms and retrieve important information about the effectiveness of a replacement algorithm.

Least Recently Used algorithm (LRU)

This algorithm is the most commonly used. The cache line that will be replaced is the one not used for the longest time. In order to implement this algorithm, additional complexity is added to the cache circuitry; we need flags that indicate when a cache line was last referenced. The implementation of LRU is getting much more difficult to implement as the associativity increases. Because of this fact, a lot of algorithms such as PLRU (Pseudo LRU) were proposed to reduce the hardware cost of LRU by approximating the LRU algorithm [14].

FIFO algorithm (First-In – First-Out)

FIFO algorithm just replaces the cache lines in a sequential order. The one that was stored first in set (also the one that is the longest time in set) will be replaced.

Most Recently Used algorithm (MRU)

This algorithm is equivalent to the LRU with the difference that the cache line that was last referenced will be replaced. The implementation is also similar to LRU. MRU is the best algorithm for cycle references (e.g. loop) [15].

Random replacement algorithm

It is the simplest to implement algorithm because it does not take cache reference history into consideration. In this algorithm the cache line that will be replaced is chosen randomly. This algorithm has tolerable results in most cases.

4.1.4 Cache Write strategies

There are two main strategies for writing data in cache: write through and write back strategies [1] [16].

Write through strategy

In write through strategy, data must be written in cache and in the lower memory hierarchy level. This strategy is easy to implement, offers safety because the lower memory hierarchy always contains a copy of the updated data, but it is quite slow because two writes must be made each time.

Write back strategy

In write back strategy, data is written only in cache. The modified cache line is written in lower memory hierarchy level only when the cache line is replaced. But how hardware can know when cache line is modified? For this purpose, a flag bit called dirty bit is inserted in cache. Dirty bit indicates, when set, that the cache line is modified. If dirty bit is set, this strategy directs to write data in the lower memory hierarchy level. This strategy offers very fast writes and low overall writes latency because multiple writes are gathered and then performed together in one write. However, it is very hard to implement and there is a risk because the lower memory hierarchy is not always consistent with the cache.

Write policies in case of write miss

In case of write miss, there are two policies that we can follow. Write allocate which writes data in cache and then a write-hit operation follows and no-write allocate which does not write data in cache but only in the lower memory hierarchy. Therefore, write through strategy uses no-write allocate policy in order to avoid back-to-back writes as write through strategy writes data in the lower memory hierarchy level. Furthermore, write back strategy uses write allocate policy hoping that there will be subsequent read or write hits [16].

4.2 L1 cache implementation

4.2.1 Setting up cache

The memory system consists of a L1 cache and the main memory. We are now ready to implement the L1 cache which will be used in instruction and data memories. The cache will be parameterized allowing us to choose the cache size, the line size and the associativity. Cache size and cache line must be given in Bytes and the association ways in integer value. In our design cache size is 4096 Bytes (4KBytes), block size is 8 Bytes (2 words) and set associativity ways are 4.That means that we need 3 bits for the byte offset field. We have 4KBytes/8Bytes = 512 cache lines, thus we have 512 cache lines / 4 ways = 128 cache lines in each way; hence index field size is 7 bits. Consequently, tag field size is 32 – 7 – 3 = 22 bits. In order to compute these values a unit that computes a $\log_2$ value was implemented. This unit gets a 32-bit vector, and outputs the $\log_2$ value of this vector. Cache is implemented with registers. For the tag comparison, tag comparator units are dynamically generated depending on the associativity. Every tag comparator compares the tag of one way to the tag value of the given address. Hit signal is generated by checking all tag comparators output and the valid bit of the requested cache line. Main memory is 1MB size (parameterizable) and is implemented as a two dimensional array.

4.2.2 Implementing the interface

The next step is to implement the way CPU, Cache and RAM interact with each other. For this purpose we implement an interface that consists of a few request and response structures. When CPU requests to read or write cache we need the memory address that will be translated, the write data and a read or write bit informing cache for the operation service. In case of a read cache request, if cache contains the requested cache line, a hit occurs and cache responses with the read data, hit signal

and a bit that informs CPU that the request is served. For a cache read miss, cache informs CPU by setting a miss signal. When a cache read miss occurs, then the requested data must be retrieved from main memory and then written back in cache. Cache generally must be able to read or write data in main memory. Therefore, cache to main memory request contains two bits that define if we have to read or write the main memory, a chip select signal, an output enable signal, a vector for the memory access address and a vector for the write data. The main memory's line is of two words size. The main memory responses with the requested data, a valid bit that indicates the validation of read data and a signal that informs that the memory response is done. Table 4.1 shows the interface between CPU, cache and main memory.

Request	Structure Fields
CPU to Cache	32-bits Access address 64-bits Data to write in Cache Read Signal Write Signal
Cache to RAM	32-bits Access address 64-bits Data to write in RAM Chip Select Signal Write Enable Signal Read Enable Signal Output Enable Signal
Response	**Structure Fields**
Cache to CPU	64-bits Data returned from Cache Hit Signal Miss Signal Done Signal
RAM to Cache	64-bits Data returned from RAM Valid Signal Done Signal

Table 4.1 Interface between CPU, Cache and RAM. All vectors are parameterizable.

4.2.3 Implementing the cache controller

Cache behavior is controlled by a controller. Controller is implemented as an FSM determining about the operations of the cache. This FSM consists of five stages. On reset signal cache is in an idle mode which means that there is no operation to be made. After this idle stage, cache controller directs cache to be in a read or write mode. Cache now is ready to accept read or write requests from CPU. If there is a

read hit, requested data is outputted while the done bit is set indicating that cache read is done successfully. Cache continues to be in read/write mode. If a read miss occurs, cache controller directs cache to request the data in main memory.

Cache is now in a ram request mode, requesting for a read service. In this stage ram is requested to return the data to cache and when data is ready (ram data is ready when ram response valid and done bits are set), they are written in cache. Cache controller searches for an empty cache line in the set by checking the valid bit in every way. If there is not free cache line, cache controller writes the data in a random chosen cache line in the set. We will describe later how random replacement policy is implemented. When the requested data is successfully written in cache, cache is again directed in the read/write mode and cache response done signal is set. In cases that cache is in a read/write mode but neither hit nor miss signal is set, then cache controller directs cache in an error stage informing CPU than a cache read/write error occurred.

The requested data is now found, so there is read hit and cache read data is delivered to CPU. Remember that cache line has a two-word size which means that it contains two words while CPU utilizes 32-bit instruction or data vectors. Therefore, we must select which word is needed from the data that was just read from the cache line. This is implemented by selecting the needed bytes depending on the byte offset value. If the MSB of the byte offset is 0 then we select the first word of the cache line (Byte 0 – 3). If it is 1 we select the second word (Byte 4 – 7) instead.

The write strategy that we are going to follow is write through with write allocate policy because our future goal is to support write back with write allocate strategy. Therefore, when a write hit occurs, cache controller updates the cache line and directs cache to request RAM for a write service while cache remains in a read/write mode waiting for the next requests. When a write miss occurs, controller directs cache to request RAM for a write service and brings the cache line in cache. When writing data in RAM is done, controller directs cache to request a read service from RAM. When RAM has served the request, data is written in cache by controller and then cache returns again in the read/write mode waiting for the next request while cache response done signal is set. In case that there is a write request but none of hit or miss signals are set, cache is directed to an error stage. Note that simultaneous read and write are not allowed. Figure 4.5 illustrates the cache controller FSM.

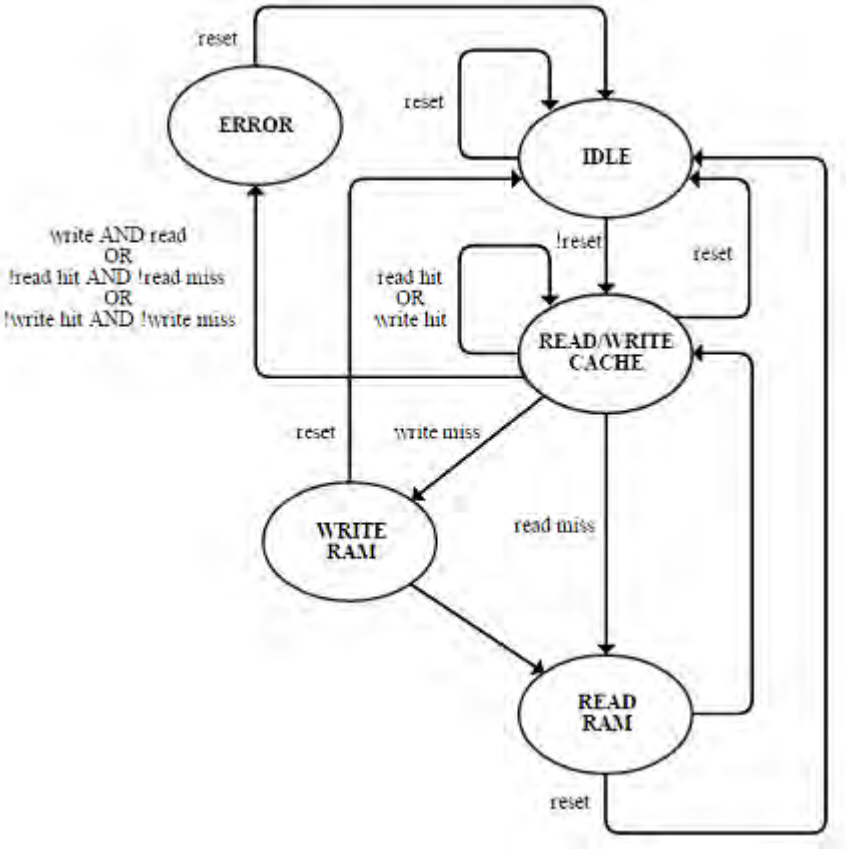

Figure 4.5 Cache controller FSM

4.2.4 Implementing random replacement policy

Random replacement policy is very easy to implement and has moderate results. In our 4-way associative cache random policy works fine well. We don't need to change hardware in order to maintain a history of cache line references but we only need to implement a random way to choose one of the four cache lines in a set to replace. There are true random number generators (TRNG) that generate random numbers from a physical process such as thermal or atmospheric noise. In our case, we will implement a pseudo-random number generator; the same sequence of numbers is repeated after time. In digital design, an entity called left feedback shift register (LFSR) is used for pseudo-random number generation. This entity is just a shift register whose input is a linear function of its previous state. There are two commonly used LFSR formats; the Fibonacci and the Galois LFSR [17]. We will implement a 4-bit Fibonacci LFSR (Figure 4.6). The initial state (also called seed) of this LFSR is 1001 (9 in decimal system) and is set on reset signal. Note that generated numbers are in the range of 1-15 so we need to restrict this range in 0-3, hence a modulo operation is applied to the current state of LFSR.

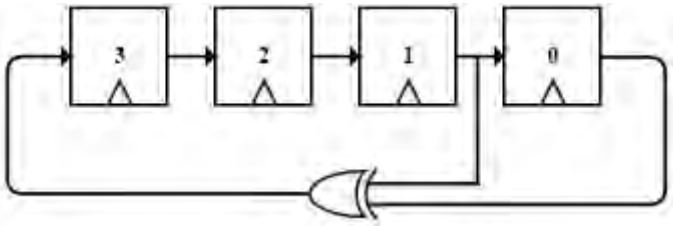

Figure 4.6 Fibonacci LFSR diagram. The feedback input is generated by an XOR gate.

4.3 Integrating cache into system

After implementation, we have to integrate cache in our system. An instruction cache and a data cache are instantiated in fetch unit and memory access unit. Of course we have to take care about connections. A ram response structure and a cache request structure are respectively defined as input and output in fetch unit. We also have to assign PC to the CPU request address and set the CPU request read signal in order to read a cache line. Pipeline execution is continued only when instruction cache has served the request, hence when cache response done signal is set. After reading a cache line, we must select the right word depending on the MSB of the Byte offset. Figure 4.7 shows how we should modify fetch unit pseudocode to support memory cache.

```
1   Ports declare
2       in:
3           clock,reset,stall,PC
4           jump target address,branch target address
5           branch taken,jump,ram response
6       out:
7           instruction,new PC,cache_to_ram request
8   end
9
10  Unit description
11      instantiate instruction cache;
12      assign PC,cache_read to cpu_request;
13
14      send request to instruction cache when no stall;
15      select word from the cache line when cache response done;
16      assign word to instruction;
17      compute PC+4 when no stall and cache response done;
18      select new PC;
19  end
```

Figure 4.7 Modified Instruction Fetch unit pseudocode to support caching.

4.3. Integrating cache into system

Integrating data cache in our system is a similar process. We instantiate data cache in memory access unit taking care for the connections. A CPU to data cache request structure is defined and assigned with the correct values. A RAM response structure is defined as an input, containing the RAM read data and the done signal, and a cache response structure is defined as an output. We have to select again the right data word. When a data cache miss occurs, pipeline must be stalled. Therefore, a stall signal is generated for all the time that data cache miss signal is set. In Figure 4.8 we can see the modified memory access unit pseudocode.

```
1   Ports declare
2       in:
3           clock,reset,read_en,write_en,data size,data_in,memory address
4           ram response
5       out:
6           data_out,error,cache_to_ram request,data cache miss stall
7   end
8
9   Unit description
10      instantiate data cache;
11      check simultaneous read & write error;
12      assign cache_read,cache_write,memory address,data_in to cpu_request;
13
14      on clk
15          if ( not error )
16              send request to instruction cache when no stall;
17          if:end
18      clk:end
19
20      generate data cache stall when data cache miss;
21      select word from the cache line when cache repsonse done;
22      assign word to data_out;
23  end
```

Figure 4.8 Modified Memory Access unit pseudocode to support caching.

Chapter 5

Implementation of MIPS coprocessor 1

As mentioned, MIPS coprocessor 0 is in charge of the system control and coprocessor 1 is a Floating-Point Unit (FPU) also called Floating-Point Accelerator (FPA). The FPU instruction set includes all floating-point operations defined by the Floating-Point Arithmetic IEEE Standard (IEEE-754). All coprocessor instructions use opcode 0100xx where last two bits specify the coprocessor number. Thus all floating-point instructions use opcode 010001.

MIPS FPU has two 32-bits control registers for controlling the FPU and additional 32 registers of 32-bits each, called single precision registers. These registers are used only for FPU operational purposes and are notated as $f0-$f31. $f0 register is not a special register as $zero of coprocessor 0; it can hold any value, not just zero. The IEEE-754 standard defines also an instruction group for double precision floating-point arithmetic which needs 64-bit operands. Hence, in order to support double precision arithmetic, recent MIPS architectures include 32 64-bits floating-point registers however it is not backward-compatible [3]. Older versions can support double-precision arithmetic by grouping single precision floating-point registers in a pair of two. This technique results in 16 pairs of 64-bits in total, each one named by the first register name; $f0 is the first double precision pair, $f2 is the second double precision pair, $f4 is the third double precision pair, etc.

5.1 Floating Point formats in IEEE 754 standard

The IEEE-754 standard defines floating-point number formats, floating-point arithmetic operations, conversions between other number formats, and floating-point exceptions [3] [9]. It is the most common representation for real numbers on computers. Floating-point numbers in IEEE-754 define single and double precision

floating-point numbers. These numbers are stored following a scientific floating-point number notation. Each floating-number can be formed by the below formula.

$$(-1)^{sign} \times mantissa \times 2^{exponent}$$

In this formula, sign indicates the sign, mantissa (also known as significand) represents the precision and exponent represents the exponent of the real number. Real numbers can have positive and negative exponents. Thus, a bias is added to the actual exponent before it is stored. Bias value is 127 for single precision and 1023 for double precision floating-point numbers. In IEEE-754 significand field has an implicit MSB 1 for normalized floating-point numbers and now significand field can define a 24-bit value. Hence, the formula is transformed into the below form:

$$(-1)^{sign} \times (1 + fraction) \times 2^{exponent}$$

5.1.1 Single Precision format

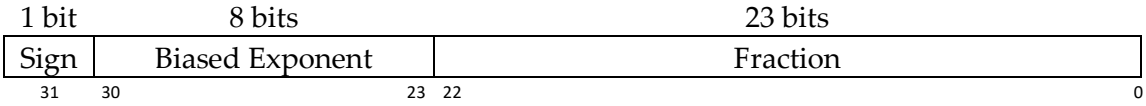

Figure 5.1 Single Precision IEEE-754 format

In Figure 5.1 we can see the IEEE-754 binary representation format of single precision floating-numbers. Single precision format can represent floating-point numbers in the range of 2.0×10^{-38} up to 2.0×10^{38}. The sign of the floating-number is stored in the 1-bit sign field. Bit 0 indicates a positive number and bit 1 indicates a negative one. The biased exponent is stored in the biased exponent field. The fraction field contains the fraction value of floating-point number. Biased exponent is computed easily by adding a value of 127 to the actual exponent. The 0 and 255 biased exponents are used to indicate the floating-point underflow and overflow special cases. Underflow refers to very small floating-point numbers and it means that the non-zero floating-number cannot be represented because the exponent value is very small to fit in the 8-bits field. Overflow on the other hand refers to very big floating-point numbers and it means that the floating-number cannot be represented because exponent is very big to fit in the 8-bits field. Therefore, the minimum and maximum actual exponents that can be supported by single precision format are -126 and +127. A floating-point underflow or overflow exception is thrown whenever an actual exponent is smaller than -126 or bigger than +127.

5.1.2 Double Precision format

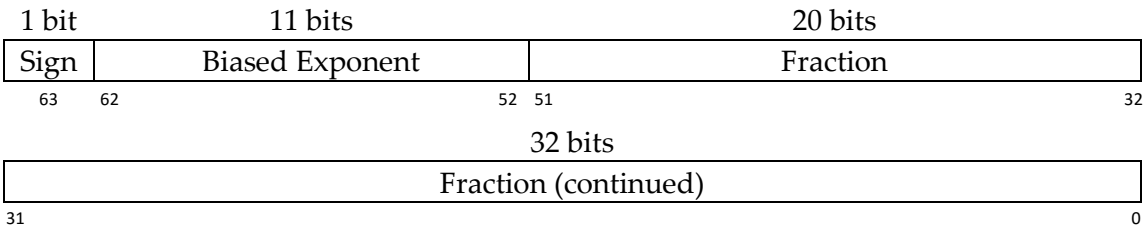

Figure 5.2 **Double Precision IEEE-754 format**

Double precision floating-point format is the solution to the underflow and overflow problems of single precision format. The main idea was to enlarge the exponent and fraction fields; larger and smaller values than single's precision format exponent and fraction can now fit. Figure 5.2 shows the double precision format where we can see that the required bits are doubled to 64-bits (or 2 words). Double precision format can represent floating-point numbers in the range of 2.0×10^{-308} up to 2.0×10^{308}. The sign bit field remains 1 bit whereas biased exponent and fraction fields have turned into 11 and 52 bits, respectively. Despite the fact that exponent and fraction fields have been enlarged, underflow and overflow problems cannot be eliminated. Biased exponent values from 0 up to 2047 now, indicating the floating-point underflow and overflow special cases. Because bias value for double precision format is 1023, the minimum and maximum actual exponents that can be supported are -1022 and +1023.

In mathematics some values such as infinity and some special operations such as division by zero are very crucial especially for approximate computation. IEEE-754 encodes and handles these special cases in a certain way. IEEE-754 reserves exponent field values of all zeros and all ones to denote special values. These special values and their IEEE-754 encoding are shown in table 5.1 below [2].

	Single Precision				Double Precision			
	Sign	Biased Exponent	Fraction	Special Value	Sign	Biased Exponent	Fraction	Special Value
Positive Zero	0	0	0	0	0	0	0	0
Negative Zero	1	0	0	- 0	1	0	0	- 0
Positive Infinity	0	255	0	∞	0	2047	0	∞
Negative Infinity	1	255	0	$-\infty$	1	2047	0	$-\infty$
Quiet NaN	0 or 1	255	$\neq 0$	NaN	0 or 1	2047	$\neq 0$	NaN
Signaling NaN	0 or 1	255	$\neq 0$	NaN	0 or 1	2047	$\neq 0$	NaN
Positive Normalized Non Zero	0	$0 < e < 255$	f	$2^{e-127}(1.f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
Negative Normalized Non Zero	1	$0 < e < 255$	f	$-2^{e-127}(1.f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1.f)$
Positive Denormalized	0	0	$f \neq 0$	$2^{e-126}(0.f)$	0	0	$f \neq 0$	$2^{e-1022}(0.f)$
Negative Denormalized	1	0	$f \neq 0$	$-2^{e-126}(0.f)$	1	0	$f \neq 0$	$-2^{e-1022}(0.f)$

Table 5.1 IEEE-754 encoding of special floating-point values. A NaN (Not-a-Number) can be produced by one of the following operations: $\infty - \infty$, $-\infty + \infty$, $0 \times \infty$, $0 \div 0$, $\infty \div \infty$. Signaling NaN signals an invalid operation exception whereas quiet NaN does not.

5.2 Fixed Point format

MIPS FPU also supports fixed point arithmetic. Fixed-point representation is an easy way to represent fractional numbers. Fixed-point values have the same format with signed integers of coprocessor 0. To define a fixed-point number we need the width of the number representation and the position of the binary point. Fixed-point format is shown in Figure 5.3 below.

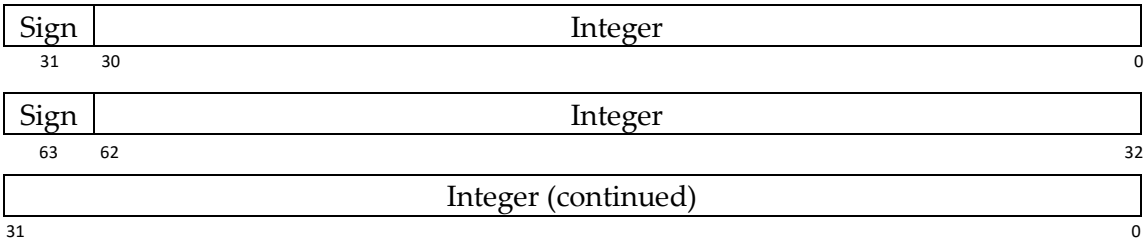

Figure 5.3 32-bit and 64-bit Fixed-Point Format. Also called as Word type (or W- type) and Longword type (or L type) format.

5.3 FP instruction format

There are two FP instruction formats, the FR-type and FI-type format. These types of format follow the same logic as in coprocessor's 0 R-type and I-type formats but they are reserved for use with floating-point numbers. FR-type format is used for floating-point arithmetic instructions whereas FI-type is used for floating-point branches.

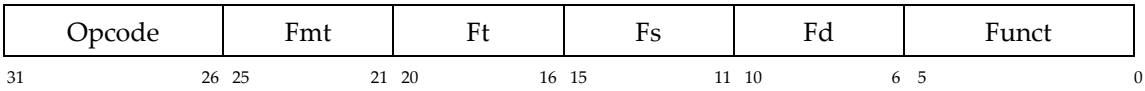

Figure 5.4 FR-Type instruction format

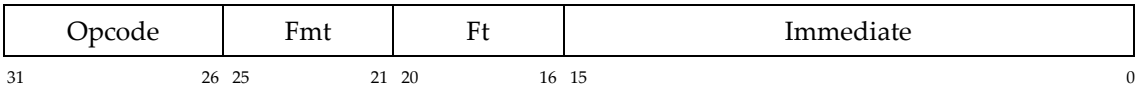

Figure 5.5 FI-Type instruction format

The main difference is the Fmt field. Fmt field is used for specifying numerical data type binary encoding. It specifies whether data type is single or double precision or fixed-point. There are also reserved values of Fmt for certain instructions, such as data transfer instructions between coprocessor 0 and FPU. These instructions use the Fmt field as an extension to Funct field.

5.4 FP instruction set

FPU instruction set consists of arithmetic (including compare), conversion, data transfer and conditional branch instructions. Arithmetic instructions use the FR-type format and Fmt field specifies the operands' and result's data encoding. Fmt can be s (Single precision), d (Double precision), w (Word fixed-point) and l (Longword fixed-point) data binary encoding. Conversion instructions also use the FR-type format and convert one data type to another. Hence, two operand fields of the FR-type format are needed; the Fd and Fs fields. Fmt specifies the data type format of the source register Fs. The unused Ft field is set to 0. Conversion can be made between all formats.

FPU has the ability to send and receive data to the system. Data can be transferred:

- Between FPU and coprocessor 0
- Between FPU and memory system
- Between all the floating-point registers (including the FPU control register)

Data transfer instructions for data transfer between FPU and memory system are I-type instructions. Instructions for data transfer between FPU and coprocessor 0 are FR-type instructions and they use two operand fields of the FR-type format; the Fd and Fs fields. Data transfers between floating-point registers are also FR-type instructions and take data type format of the transferred data into consideration.

Conditional floating-point branch instructions are equivalent to conditional branches described before. The difference is that in floating-point conditional branches the compare operands are floating-point values and instruction format is FI-type. These branch instructions are also using a PC-relative addressing mode.

5.5 Implementing Floating Point Unit

We are now ready to describe the way that floating-point unit was designed and implemented. The main idea was to find a generator that would create synthesizable code for the floating-point arithmetic operations and to concentrate on the way that all these operators would be combined and utilized efficiently. The next step was to create all the data transfer instructions and finally to integrate successfully the FPU in the system.

5.5.1 Generating FP arithmetic operators

FloPoCo is an open source generator of floating-point operators for FPGAs [18]. These operators are written in C++. It is a command-line tool and commands follow the syntax: `flopoco <options> <operator specification list>.`

Command-line interface

FloPoCo options are the following [site]:

- `target:` Sets the target hardware family (e.g. target=virtex5).
- `frequency:` Sets the target in MHz (e.g. frequency=300).
- `name:` Replaces the name of the generated entity for the next operator(e.g. name=fp_adder).

- `plainVHDL`: Instructs FloPoCo to output concise and readable VHDL, using only + and * VHDL operators instead of FloPoCo adders and subtractors.
- `useHardMult`: Instructs FloPoCo not to use hard multipliers or DSP block.
- `unusedHardMultThreshold`: Instructs FloPoCo to use a hard multiplier (or DSP block) if less than 30% of this hard multiplier are unused. The ratio is between 0 and 1, such that 0 means: any sub-multiplier that does not fully fill a DSP goes to logic; 1 means: any sub-multiplier, even very small ones, will consume a DSP.
- `pipeline`: Requires the operators to be pipelined. If no, the operator will be combinatorial. If yes, registers may be inserted if needed to reach the target frequency

FloPoCo operators can be seen in this link: http://flopoco.gforge.inria.fr/operators.html

FloPoCo instruction representation format

FloPoCo format is practically an expansion to the IEEE-754 format. There are two more LSB (exception field) bits that indicate whether the floating-point value is a special case or not. The fraction and exponent fields are parameterized by two integers wE and wF indicating the size of the exponent and fraction fields. For instance, if we want to create a single precision FloPoCo formatted number we should use wE=8 and wF=23 values.

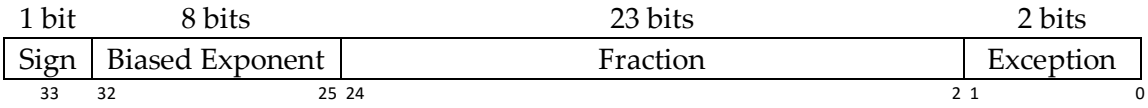

Figure 5.6 Single Precision FloPoCo format

The special value cases that Exception field encodes are the zero (00), normal numbers (01), infinities (10) and NaN (11).

Installation and generating operators

We will use FloPoCo version 4. For the installation process you can visit the following link: http://flopoco.gforge.inria.fr/flopoco_installation.html

We will create the arithmetic operators `FPAddSub`, `FPMult`, `FPDiv`, `FPSqrt`, `FPExp` and `FPLog`. FloPoCo also provides operators for conversions between FloPoCo and IEEE-754 formats, such as InputIEEE and OutputIEEE. All FloPoCo operators use FloPoCo format, therefore we will use the InputIEEE to change the FPU arithmetic operands from the given IEEE-754 format to FloPoCo format. When FPU is done with the computation, the result needs to be converted again in IEEE-

754 format before it is stored in FPU register file. We can see the pipeline depth of the generated entities in Figure 5.7.

```
Final report:
Entity FPAddSub_8_23_uid2
   Pipeline depth = 10
Entity FPMult_8_23_8_23_8_23_uid26
   Pipeline depth = 2
Entity FPDiv_8_23_F400
   Pipeline depth = 12
Entity FPSqrt_8_23
   Pipeline depth = 25
Entity FPExp_8_23_F400
   Pipeline depth = 13
Entity InputIEEE_8_23_to_8_23
   Not pipelined
Entity OutputIEEE_8_23_to_8_23
   Not pipelined
Entity IterativeLog_8_23_10_400
   Pipeline depth = 14
Output file: flopoco.vhdl
vagrant@vagrant-ubuntu-trusty-32:~/flopoco-4.0.beta0$
```

Figure 5.7 FloPoCo final report.

5.5.2 Floating Point arithmetic instructions

We have generated the basic floating-point operators, now we have to design FPU. `FPAddSub`, `FPMult` and `FPDiv` get two operand inputs in FloPoCo format and produce a FloPoCo formatted result. `FPSqrt`, `FPEx` and `FPLog` need one FloPoCo formatted operand input instead of two. We begin with instantiating all generated modules. All floating-point instructions have the opcode 010001; when this opcode is detected CPU sends a request signal, a FPU enable signal and the floating-point instruction to FPU.

FPU then categorizes the incoming instructions. For this task, FPU stores the incoming instruction in FIFO buffers for each type of floating-point instruction operation. We have six operators, so we create six buffers. The size of these buffers is parameterizable. Each buffer is controlled by a controller that gets the categorized instruction and stores it in the corresponding buffer depending on the instruction's operation. This way we can execute floating arithmetic instructions in a parallel way, if there is not data dependency of course. In addition, CPU can continue without delays; it just sends the floating-point instruction without waiting an immediate execution. Note that all FP arithmetic instructions in this design have a 10000 Fmt field indicating single precision data type.

The execution of arithmetic floating-point instructions begins by reading an instruction from the buffers. Each buffer is read and every instruction that was just read is executed in a specific pattern; instruction is decoded, executed and the result is written back in the floating-point register file. In the FP decode stage every instruction is split into the floating-point single precision IEEE-754 format fields. The floating-point register file contains 32 registers of 32-bits each and is implemented in a similar way with the CPU register file.

FloPoCo generated entities do not have a signal that reports the end of execution. Therefore, the execution stage of floating-point arithmetic instructions is handled as a stage of waiting FloPoCo operation entities to produce the result. The amount of clock cycles, for the execution to be completed, depends on the pipeline depth of the entity. For example, FPAddSub needs 10 clock cycles and FPExp needs 14 clock cycles to produce their result. Every floating-point arithmetic operation is controlled by a FSM which consists of four stages; idle, decode, wait for execution and write back stages. This FSM can be seen in Figure 5.8.

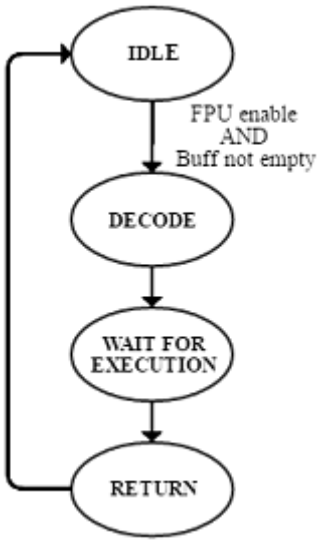

Figure 5.8 Floating-Point Operation control FSM The wait for execution stage depends on the pipeline depth of the operation. In return stage the result is written in floating-point register file. Every floating-point arithmetic instruction completes in pipeline depth + 2 clock cycles from the time it is read from buffer.

What if a buffer cannot receive other instruction? If a buffer is full then we have to inform CPU and wait for free space. We will use a 6-bit vector (equal to the amount of buffers) and every j bit of this vector will indicate whether buffer j is full or not. If a buffer's index exceeds the size of the buffer, controller will set the corresponding bit of the vector. The order of the arithmetic operation that corresponds to the vector bits, beginning from the LSB, is addsub buffer, mult buffer, div buffer, sqrt buffer,

exp buffer and log buffer. When a buffer is full, CPU does not fetch another instruction so that there is not big load. In case that all buffers are empty we have to inform that there is no floating arithmetic instruction to execute. We create similarly a 6-bit vector that indicates whether a buffer is empty or not. When all bits are set, all buffers are empty and CPU will disable FPU. FPU arithmetic operation datapath and control flow can be seen in Figure 5.9 below.

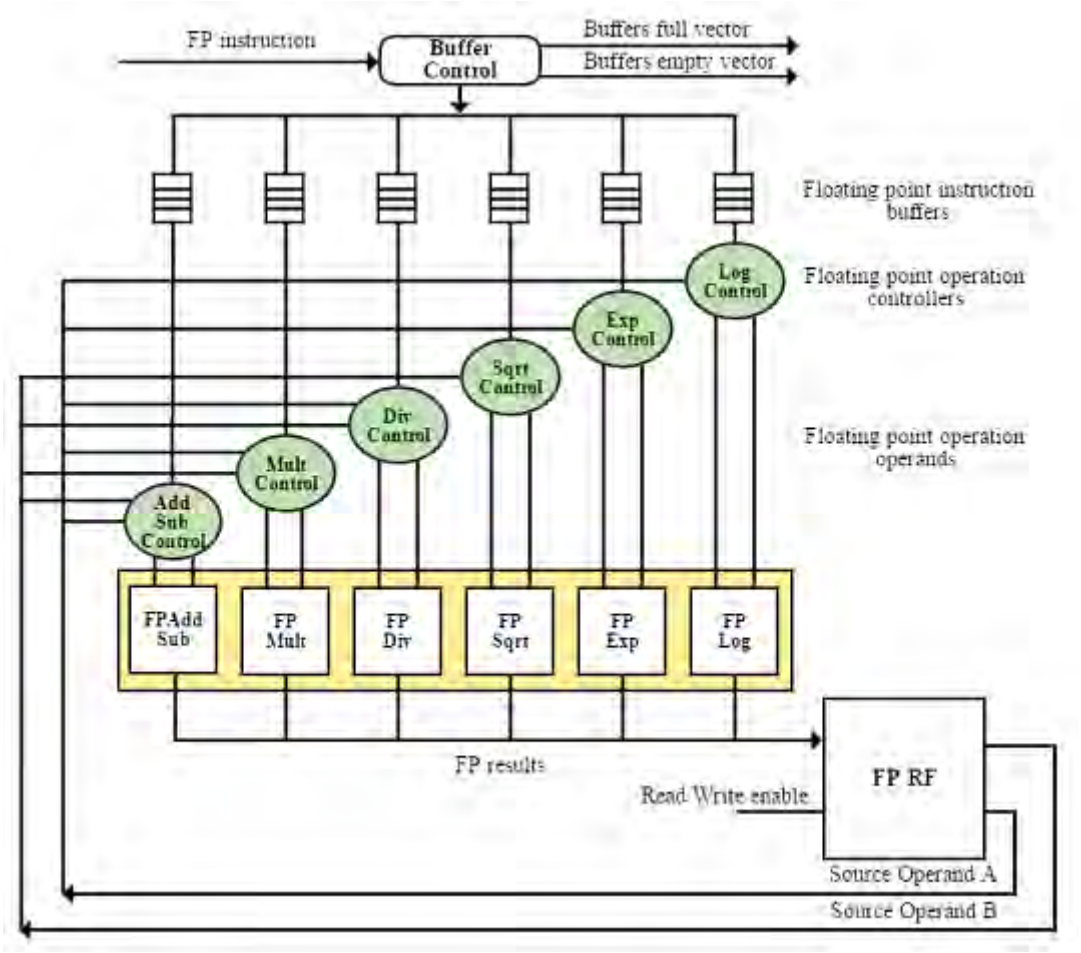

Figure 5.9 FPU Arithmetic Instruction Datapath. FP instruction is categorized by the buffer controller and stored in the corresponding buffer. Buffer controller generates the full and empty buffer vectors. FP instructions are executed in parallel and results are stored in FP register file. The Floating point operation controllers control the FP execution and also enable register file reads and writes.

5.5.3 Confronting data hazards

Floating-point instructions need several and different amount of clock cycles to execute. Therefore, a lot of dependencies need to be solved. Floating-point instructions insert write after write and write after read data dependencies in our system because every instruction has a different pipeline depth. For instance,

consider an FPMul instruction after an FPExp instruction and the same destination register. Because FPMul is completed earlier, this register is written in a wrong sequence. Therefore, incorrect data will be read by a following instruction if FPExp hasn't completed its execution. In another scenario, FPMul instruction's destination register may be the same with one of the source registers of FPExp instruction. In that case, FPExp can read wrong data if FPMul has completed its execution. Thus, we need to deal with write after write (WAW), write after read (WAR) and read after write (RAW) data hazards.

We need to inspect which FP register is written and read. For this reason, we will make use of two vectors which will hold the floating-point registers' usage. We practically inspect when a FP register is written or read. Each vector is 32-bits of size because we have 32 FP registers. Every vector's j bit corresponds to the j floating-point register. Let's call the first vector floating-point read inspection vector and the second one floating-point write inspection vector. We examine the received instruction and all buffers and set the corresponding bits in two vectors. These vectors are monitored by CPU, allowing it to decide if a data hazard between FP arithmetic instructions occurs. Checking for data dependency is done right after a floating-point instruction is detected. Whenever CPU detects a floating-point instruction, it checks the corresponding bits of read and write inspection vectors.

RAW floating- point data hazards are detected by checking the source registers of the currently detected floating-point instruction and the corresponding bit entries in the write inspection vector. If there is a match (a corresponding bit is set) CPU must wait for the pending instruction to complete its execution and not fetch other instructions. When the execution is completed, FPU sets the corresponding bits to zero and CPU is allowed to send the floating-point instruction to FPU for execution.

WAR data hazards are detected in a similar way by checking the destination register of the currently detected floating-point instruction and the corresponding bits in the read inspection vector. In WAW data hazards CPU checks for a match between write destination register and the corresponding bit in write inspect vector. In both cases, if there is a match CPU waits and resumes when the execution is completed and the read registers (WAR case) and write registers (WAW case) are released. Figure 5.10 shows the modified FPU datapath for supporting register inspection and Figure 5.11 shows the implementation pseudocode of FPU.

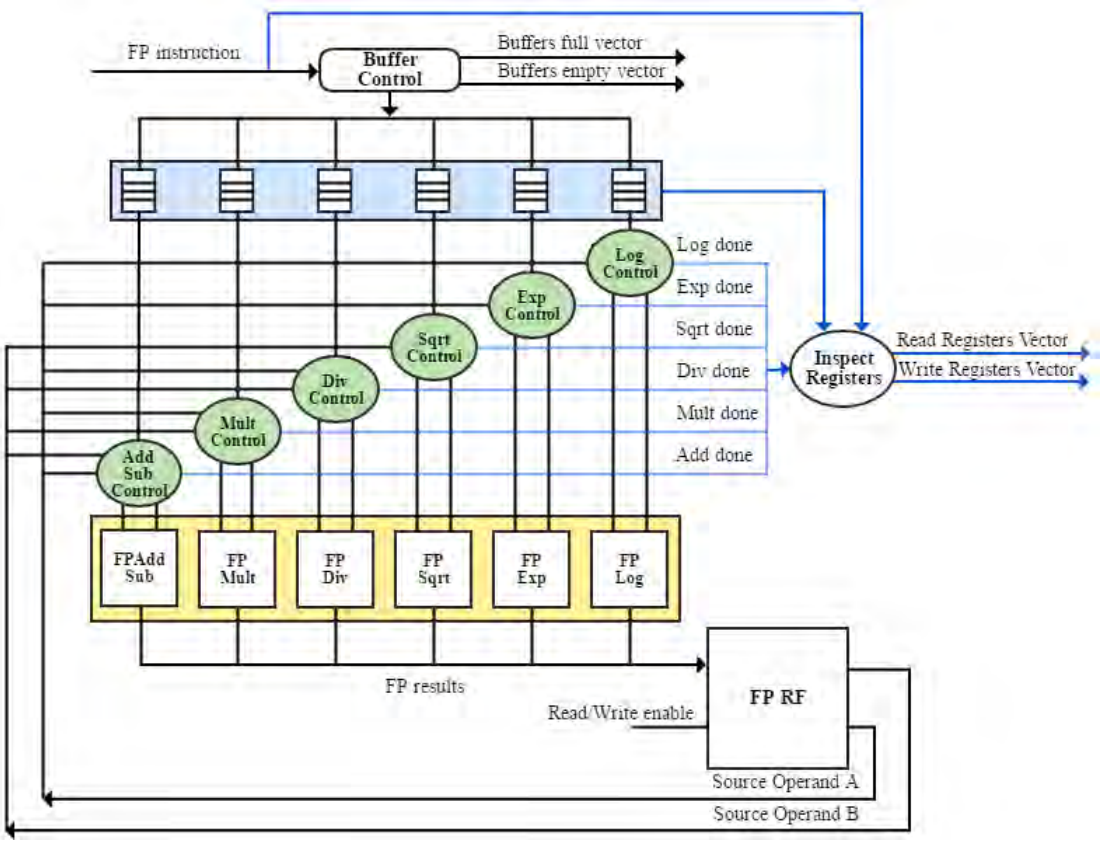

Figure 5.10 FPU Arithmetic Instruction Datapath with register read and write inspection.

```
Ports declare
    in:
        clock,reset,fpu enable,cpu request,
        fp instruction
        fpu stall
    out:
        fp buffers full vector
        fp buffers empty vector
        fp write_inspection vector
        fp read_inspection vector
end

Unit description
    instantiate flopoco entities;
    instantiate fp register file;

    convert operands from ieee754 to flopoco;
    convert flopoco entities results to ieee754 results;

    on rst
        initialize register file;
    rst:end

    on clk
        store or read fp register file;
    clk:end

    determine whether buffers are full or empty;
    when buffer full generate stall signal;

    inspect read_registers(
        set default value 0;
        check incoming fp instruction;
        check all buffers instructions;
        bind registers in fp read_inspection vector;
        release registers of completed instruction;
    );
    inspect destination_registers(
        set default value 0;
        check incoming fp instruction;
        check all buffers instructions;
        bind registers in fp write_inspection vector;
        release registers of completed instruction;
    );

    on clk
        when cpu request and send fp instruction(
            if ( no fpu stall ) categorize incoming fp instruction;
        );

        control add-sub fifo buffer(
            push add-sub instruction in buffer;
            pop add-sub instruction when completed;
        );
```

```
55          control mult fifo buffer(
56              push mult instruction in buffer;
57              pop mult instruction when completed;
58          );
59          control div fifo buffer(
60              push div instruction in buffer;
61              pop div instruction when completed;
62          );
63          control sqrt fifo buffer(
64              push sqrt instruction in buffer;
65              pop sqrt instruction when completed;
66          );
67          control exp fifo buffer(
68              push exp instruction in buffer;
69              pop exp instruction when completed;
70          );
71          control log fifo buffer(
72              push log instruction in buffer;
73              pop log instruction when completed;
74          );
75
76          if (fpu is enable)
77              when no dependency occurs(
78                  if (add-sub buff not empty) execute add-sub;
79                  if (mult buff not empty) execute mult;
80                  if (div buff not empty) execute div;
81                  if (sqrt buff not empty) execute sqrt;
82                  if (exp buff not empty) execute exp;
83                  if (log buff not empty) execute log;
84              );
85          if:end
86      clk:end
87  end
```

Figure 5.11 FPU Arithmetic instruction datapath implementation pseudocode.

5.5.4 Floating Point Unit data transfer instructions

As previously mentioned, data transfers can be made between memory (specifically the data cache) and FPU and between CPU and FPU. The first type of data transfers is CPU instructions which transfer data between data cache and FPU. The second type is FPU instructions which transfer data between CPU and FPU register files.

Implementing data transfers between FPU and Memory

These instructions are LWC1 and SWC1 which are I-type format instructions. LWC1 instruction reads data from data cache and stores it in the FPU register file whereas

SWC1 does the opposite transfer. The main difference of these instructions and other I-type format instructions is that now the destination register field will indicate a FP register (Figure 5.12).

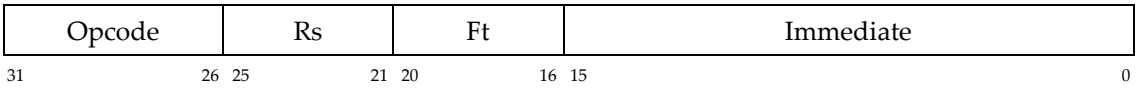

Figure 5.12 I-Type format for LWC1 and SWC1 instructions

These instructions are detected in the decode stage of the pipeline where opcodes 110001 (LWC1) and 111001 (SWC1) are detected. By detecting these instructions, decode unit will generate two new specific control signals which indicate a store in FP register file (LWC1) and a load from FP register file (SWC1). Decode unit also sets the memory read control signal for LWC1 instruction and the memory write control signal for SWC1 instruction. In ID stage, the store in FP and load from register file control signals are stored in ID/EX pipeline register and propagated through pipeline registers to MEM stage. In MEM stage these instructions access the data cache; a read access for LWC1 and a write access for SWC1. In WB stage of LWC1 instruction the store in FP register file forces memory data out, read from MEM/WB pipeline register, to be stored in the FP register file instead CPU register file. In ID stage of SWC1 instruction the load from FP control signal indicates that data will be read from FP register file instead of CPU register file. These data are stored in ID/EX pipeline register and propagated to MEM stage through the pipeline where they are stored in data cache.

After implementing data transfers between memory and FPU we need to check for data dependencies that may occur. First of all, RAW FP data hazards after LWC1 instruction are eliminated due to the FPU design, because all FP instructions are decoded in FR-format fields two clock cycles after they are detected. This means that there is enough time-frame for LWC1 to write back data in FP register file. For instance we can consider the RAW data dependency in Figure 5.13. We notice that the decode of FPMul instruction is done 1 clock cycle after LWC1 has written the data in FP register file ensuring that there is no RAW data hazard between LWC1 and FP arithmetic instructions. Notice that there is not also WAW data hazards between LWC1 instruction and following FP arithmetic instructions because the smallest (in clock cycles amount) FP arithmetic instruction FPMul will write back in register file four clock cycles (two clock cycles to start execution and two clock cycles of the FPMul entity pipeline) after LWC1's write back stage.

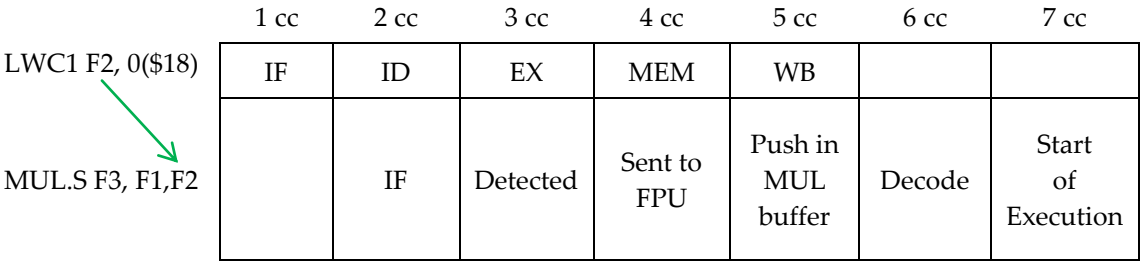

Figure 5.13 RAW data dependency between LWC1 and FP arithmetic instructions. We can see that there is no RAW data dependency because FP Arithmetic instructions are decoded 2 cycles after they are detected.

Let's check now about RAW data dependency between LWC1 and SWC1 instructions. This really means that we have a MEM to MEM data dependency because SWC1 needs to store in memory the data that LWC1 has read from memory. So data forward unit detects the dependency and forwards the memory read data to MEM stage of SWC1 instruction. Remember that both instructions have memory access control signals as LW and SW instructions have.

There is also one more WAW data hazard that we need to check. Consider a FP arithmetic instruction and a LWC1 instruction that write back in the same destination register. We need to find a solution to this hazard. WAW data hazards between FP arithmetic instructions were eliminated by implementing a FP write inspection vector. So why don't we just check this vector? Every time a LWC1 instruction is detected, CPU checks the floating inspection array and determines if there is a WAW hazard. Remember that WAW will occur if there is a match between the destination register of LWC1 and the corresponding bit of the vector. If a WAW hazard is detected, then CPU stalls the pipeline and does not fetch another instruction waiting for the write register release when the execution of the arithmetic instruction is finished.

Implementing data transfers between FPU and CPU register files

There are two FP instructions that allow data transfers between FPU and CPU register files. MFC1 instruction transfers data from FP register file to CPU register file while MTC1 transfers data in the opposite direction. These instructions follow the FR-type format and need two operands; the registers that take part in data transfer. The destination register is always defined by the Fd field and the source register by the Fs field. That means that these fields are handled differently because MFC1 has a CPU destination register and a FPU source register whereas MTC1 has a FPU destination register and a CPU source register.

Because these instructions are FP instructions, CPU sends them to FPU for execution. It is FPU's turn to recognize them and execute them. How can FPU recognize MFC1 and MTC1 instructions? We can distinguish these instructions by the Fmt field; MFC1's Fmt value is 00000 and MTC1's is 00100. We will modify FPU's implementation in order to support these instructions. We will implement these instructions in a 5 clock cycle pattern so that we can avoid WAW dependencies with previous CPU instructions. For instance, if MFC1 instruction needed less clock cycles than the CPU pipeline depth to complete, this would result in a WAW data hazard if the previous CPU instruction (e.g. a SUB instruction) writes back data in the same destination register.

MFC1 instruction is first detected and sent to FPU. FPU has to recognize that the incoming instruction is a MFC1 instruction, enable the FP register file read, output these data and report that job is done. This done signal is sent with the FP register read data to CPU and is used in order to enable a write in CPU register file. The sequence of operations need to be done can be summarized in the Figure 5.14 below.

	1 cc	2 cc	3 cc	4 cc	5 cc
MFC1 $8, F3	IF	Detected	Sent to FPU	Recognize and enable FP RF read	Output done signal and read data

Figure 5.14 Operation sequence of MFC1 instruction. $8 is the destination register. Our MFC1 implementation imitates the flow of a LW instruction; the FP RF read takes place in the 4th clock cycle (similar to memory read in LW instruction) and write back in the 5th clock cycle.

MTC1 is implemented in a similar way with the difference that now we have to write FP register file. We also need to send the write data to FPU and inform when job is done. The MTC1 operation sequence is shown in Figure 5.15.

	1 cc	2 cc	3 cc	4 cc	5 cc
MTC1 $8, F3	IF	Detected	Sent to FPU	Recognize and enable FP RF write	Write Completed Output done signal

Figure 5.15 Operation sequence of MTC1 instruction. F3 is the destination register.

Decode unit and FPU must be modified so that MFC1 and MTC1 can be supported. Decode unit needs the MFC1 read data and done signal as inputs so that when done signal is set, the MFC1 read data will be written in RF. Hence, we will use one more multiplexor that selects between the MFC1 data and all the WB data that we have previously mentioned. Two more simple controllers must be created in FPU to control the operation sequences of these instructions. FPU also needs the CPU register file read data as an input.

5.5.5 MTC1 and MFC1 interaction with other instructions

After implementing MTC1 and MFC1 instructions, the next step is to deal with data hazards that arise. These instructions write and read in register files; hence it is sure that data dependencies will occur. Remember that every incoming FP instruction's source and destination registers will set the corresponding bits in the inspection vectors. Hence, MFC1's source register will set the read inspection vector and MTC1's destination register will set the corresponding bit in the write inspection vector.

Interaction between MTC1 and MFC1

A data dependency can occur if these instructions follow each other. Specifically, consider the following examples:

A1	A2
MFC1 $7, F10	MTC1 $8, F10
MTC1 $7, F11	MFC1 $7, F10

In case A1 there is a RAW data hazard. MFC1 writes into $7 register while MTC1 has to read $7 register. MFC1 instruction will read the FP register file in the 4th stage of its execution before the WB in CPU register file takes place in 5th stage. MTC1 needs to have that data available till the 4th stage of its execution; data must be sent to FPU in 4th stage in order to be written in FP register file in the 5th stage. So we will act as we did before in load-use data dependency; we will stall pipeline and then forward that data from the production stage to the stage that they are needed. Therefore, we need to modify stall unit so that it can detect the dependency. This can be done by detecting the opcode and fmt fields of MFC1 instruction (read from ID/EX pipeline register) and the opcode and fmt fields of MTC1 instruction (read from IF/ID pipeline register). We also need to check for register value equality; MFC1's Fd field must be equal to MTC' Fs field. Now we have to send the FP register read data of MFC1 instruction to FPU after RAW data hazard is detected.

Thus, we need to modify data forward unit in order to detect this hazard, and in order not to confuse this hazard with hazards between CPU instructions. The first thing to do is to distinguish these instructions from others. The hazard is detected in the same way with stall unit but MFC1 fields are now read from MEM/WB pipeline register and MTC1 from EX/MEM pipeline register; opcode and fmt fields of MFC1 are 010001 and 00000 and MTC1's are 010001 and 00100. Now we only have to check if we have equality between destination register of MFC1 and the source register of MTC1 and forward the data to FPU.

In case A2 a RAW hazard is also occurred. In this case, a stall is generated because MTC1 has set the 10th bit in write inspection vector, indicating that F10 will be written by a previous pending instruction. That means that MFC1 and next instructions will have to wait until this FP register is released (when MTC1 writes the data in FP register file) securing that MFC1 will read FP register file after MTC1's write stage.

Interaction with FP Arithmetic instructions

In this interaction all data hazards are avoided due to the FP register inspection. Consider the following examples:

B1	B2	B3	B4
ADD.S F10, F1, F5	MFC1 $7, F10	MTC1 $7, F10	MUL.S F8, F10, F5
MFC1 $7, F10	MUL.S F10, F1, F5	MUL.S F8, F10, F5	MTC1 $7, F10

In B1 example we have a RAW hazard because MFC1 needs to read F10 register and ADD.S writes in F10 register. In B2 example there is a WAR hazard because MFC1 reads and MUL.S writes F10 register. In B3 example there is a RAW hazard because MTC1 writes and MUL.S reads F10 register. In B4 example there is a WAR hazard because MUL.S reads and MTC1 writes in F10 register. There are also WAW hazards when MTC1 and other FP arithmetic instructions have the same destination register.

Every time that MTC1 or MFC1 instruction is sent to FPU, write and read inspection vectors are checked in order to detect possible hazards. A stall signal is generated whenever MTC1 or MFC1 need to access a binded FP register, securing that all hazards between these instructions and FP arithmetic instructions are resolved.

Interaction with CPU instructions

MTC1 and MFC1 instructions' destination and source registers pass through EX/MEM and MEM/WB pipeline registers, hence we can use them for hazard detection of hazards generated by the interaction with CPU instructions. All these hazards are detected in the same way as the hazards between CPU instructions but

we have to forward now data to FPU or from FPU. We also have to insert more control information in forward and stall units in order to distinguish these hazards from hazards between CPU instructions. Consider the following examples:

C1	C2	C3	C4
ADD $7, $20, $21	MFC1 $7, F10	LW $7, 0($18)	MFC1 $7, F10
MTC1 $7, F11	ADD $20, $7, $8	MTC1 $7, F11	SW $7, 0 ($18)

In C1 example we can see that there is a RAW hazard on register $7. ALU instructions produce the result in EX stage; hence we need to forward the result to FPU when the hazard is detected. This is similar to the data dependency where the ALU out is forwarded from EX/MEM pipeline register to the EX stage of the depended instruction. The difference though, is that we don't want to forward data to ALU but instead to FPU. Therefore, we need to distinguish this hazard by checking also if the depended instruction is MTC1. If it is indeed MTC1, we forward to FPU and not to ALU. We can distinguish MTC1 instruction by the opcode (010001) and Fmt (00100) fields. These forwarded data are stored in a specific register in FPU when forwarded, read in the next clock-cycle and stored in FP register file. If there was one more independent instruction between ADD and MTC1 instruction then the data would be forwarded from MEM/WB to FPU; similar to the data dependency where the ALU out is forwarded from MEM/WB pipeline stage to the EX stage of the depended instruction.

In C2 example we have a RAW data hazard on $7 register. Now we have to stall pipeline and then forward data from FPU to ALU. The stall is generated in the same way as in load-use data dependency; we modify stall unit to distinguish the RAW data hazard between MFC1 and ALU instruction by using the opcode and fmt fields of MFC1 instruction. We will also use these fields (now read from MEM/WB pipeline register) in data forward unit to distinguish this hazard from a data hazard between two CPU instructions. The depended instruction is read from EX/MEM pipeline register. When data hazard is detected and when the opcode and Fmt fields of the first instruction are 010001 and 00000 indicating that it is MFC1, we forward the data that was read from FP register file to ALU. In case that there was one independent instruction between MFC1 and ADD then this hazard would be detected as an EX hazard where the depended instruction is read from ID/EX pipeline register. Again, we would forward data from FPU to EX stage.

There are no WAW or WAR data hazards between MFC1/MTC1 and CPU instructions because we have taken care to implement these instructions to be

completed in 5 clock cycles in order to have the same pipeline depth with the CPU instructions.

In case C3 we have a load-use data dependency because MTC1 needs to read the contents of $7 register. This hazard is solved by forwarding the read data of LW instruction from the MEM stage to FPU and stalling the depended instruction for 1 clock cycle as we did before. For this reason, we have to distinguish that the load-use depended instruction is MTC1 by using the opcode and Fmt fields. So we can forward the memory read data from MEM stage to FPU while preventing them to be forwarded to MEM stage (LW-SW data dependency) and to EX stage (load-use data dependency between LW and ALU instructions).

In case C4 there is a RAW data hazard because MFC1 writes and SW then reads $7 register. This hazard can be solved by forwarding the FP read data of MFC1 to MEM stage of SW. In order to achieve this, we modify the control case of data forward unit where destination registers of MEM/WB and EX/MEM pipeline registers are equal. Now we have to check also the case where the instruction is MFC1 (again by checking the opcode and Fmt field) and the depended instruction is SW (by checking if there is a write control signal). When this hazard exists, then data from FPU is forwarded to MEM stage.

Interaction with LWC1 and SWC1

LWC1 and SWC1 are handled by data hazard stall unit as LW and SW because they have the same memory read and memory write control signals. So, in load-use data dependency between LWC1 and MFC1 instructions, MFC1 is correctly stalled. The only thing we have to do is to detect this hazard by distinguishing the MFC1 instruction by its opcode and Fmt field and data will be forwarded to FPU. The MTC1 to SWC1 RAW data hazard is solved in the same way like the MFC1 to SW data hazard.

Chapter 6

Conclusion

6.1 Summary

Nowadays, EDA industry has delivered higher-level tools to academics and engineers so that they can make use of the FPGA technology benefits of reprogrammable silicon. Moreover the industry invests billions in FPGA research and targets to the production of more powerful FPGA platforms. Altera's acquisition by Intel Corporation reveals the significance of powerful reconfigurable platforms and signifies a new promising era in digital and embedded systems.

In this project we used Altera ModelSim Quartus Prime and Xilinx ISE Design Suite tools for the design and simulation of the project. We started implementing the pipeline and the basic instruction set of coprocessor 0. The next step was to implement and integrate cache and the final step was to design, implement and integrate FPU in the core system. The instructions that are supported are listed in Appendix. The most difficult part of this project was the design and implementation of FPU because issues about synchronizing and controlling the FPU entities were difficult to be solved. In addition, a lot of effort was made in order to integrate all these parts of the project. Every part was tested separately before being integrated in the core system. After completing the design and integration all those parts, we tested the functionality of the MIPS core by running several codes which consisted all-types of functions and all data and control dependencies.

After completing all the necessary tests, we used Vivado Design Suite by Xilinx in order to synthesize the core design. The device target was Virtex-7 VC709. We faced a problem in this step because we used SystemVerilog classes in our design but Vivado Design Suite does not support them. Therefore, we had to make a lot of changes before synthesizing the core design.

You can contact me at stpistop(AT)uth(DOT)com for a copy of the source code. The synthesis utilization report can be seen in Figure 6.1.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	262454	0	433200	60.58
LUT as Logic	262368	0	433200	60.57
LUT as Memory	86	0	174200	0.05
LUT as Distributed RAM	0	0		
LUT as Shift Register	86	0		
Slice Registers	128588	0	866400	14.84
Register as Flip Flop	94689	0	866400	10.93
Register as Latch	33899	0	866400	3.91
F7 Muxes	19829	0	216600	9.15
F8 Muxes	9095	0	108300	8.40

Figure 6.1 Synthesis Utilization Report

Completing this project helped me enrich my knowledge in computer architecture and understand how processors are designed and implemented and improve my skills in digital design by analyzing and confronting each arising problem.

6.1 Future Work

Despite all the hard work made on this project, we can continue its implementation to a more complete level. Implementing floating-point branch and convert instructions would complete the FP instruction set. We also like to expand the memory hierarchy with a L2 cache level for more realistic approach and change the write strategy into write back with write allocate. Finally, a branch prediction buffer can be implemented for a better performance.

Appendix

R-type		
Instruction	**Opcode/Function**	**Operation**
add	000000/100000	$d = $s + $t
sub	000000/100010	$d = $s - $t
mult	000000/011000	$d = $s * $t
div	000000/011010	$d = $s / $t
slt	000000/101010	$d = ($s < $t)
and	000000/100100	$d = $s & $t
or	000000/100101	$d = $s | $t
nor	000000/100111	$d = ~($s | $t)
xor	000000/100110	$d = $s ^ $t
sll	000000/000000	$d = $t << shamt
sllv	000000/000100	$d = $t << $s
srl	000000/000010	$d = $t >>> shamt
srlv	000000/000110	$d = $t >> $s
sra	000000/000011	$d = $t >>> shamt
srav	000000/000111	$d = $t >>> $s
jr	000000/001000	PC = $s
jalr	000000/001001	$31 = PC; PC = $s
I-type		
Instruction	**Opcode/Function**	**Operation**
beq	000100	if ($s == $t) PC = PC + 4 + BranchAddr
bne	000101	if ($s != $t) PC = PC + 4 + BranchAddr
addi	001000	$d = $s + SignExtImm
andi	001100	$t = $s & SignExtImm
ori	001101	$t = $s | SignExtImm
xori	001110	$d = $s ^ SignExtImm
slti	001010	$t = ($s < SignExtImm)
lw	100011	$t = MEM [$s + SignExtImm]
sw	101011	MEM [$s + SignExtImm] = $t
lh	100001	$t = (15:0)MEM [$s + SignExtImm]
sh	101001	(15:0)MEM[$s + SignExtImm] = (15:0)$t
lb	100000	$t = (7:0)MEM [$s + SignExtImm]
sb	101000	(7:0)MEM [$s + SignExtImm] = (7:0)$t
lwc1	110001	ft = MEM [$s + SignExtImm]
swc1	111001	MEM [$s + SignExtImm] = ft
J-type		
Instruction	**Opcode/Function**	**Operation**
j	000010	PC = jumpAddr
jal	000011	$31 = PC; PC = JumpAddr

FR-type		
Instruction	**Opcode/Format/Function**	**Operation**
add.s	010001/10000/000000	fd = fs + ft
sub.s	010001/10000/000001	fd = fs - ft
mult.s	010001/10000/000010	fd = fs * ft
div.s	010001/10000/000011	fd = fs / ft
sqrt.s	010001/10000/000100	fd = sqrt(fs)
exp.s	010001/10000/000101	fd = exp(fs)
log.s	010001/10000/000110	fd = log(fs)
mfc1	010001/00000/000000	$d = fs
mtc1	010001/00100/000000	fd = $s

A-1. Instructions supported by this MIPS core implementation

Bibliography

[1] Computer Organization and Design: The Hardware/Software Interface, David A Patterson, John L. Hennessy, 4th Edition, Morgan Kaufmann, 2009

[2] Computer Architecture and Organization, William Stallings, PHI Pvt. Ltd., Eastern Economy Edition, Sixth Edition, 2003

[3] MIPS IV Instruction Set Revision 3.2, Charles Price, MIPS Technologies Inc, September, 1995

[4] MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Revision 6.01, Imagination Technologies LTD , August 20, 2014

[5] MIPS® Architecture For Programmers Volume II-A: Introduction to the MIPS32® Revision 6.05, Imagination Technologies LTD , May 20, 2016

[6] SystemVerilog 3.1a Language Reference Manual Accellera's Extensions to Verilog, Accellera Organization Inc, May 2004

[7] A Proposal for a Standard Synthesizable Subset for SystemVerilog-2005:What the IEEE Failed to Define, Stuart Sutherland, DVCon-2006, San Jose, CA

[8] Synthesizing SystemVerilog Busting the Myth that SystemVerilog is only for Verification, Stuart Sutherland, Don Mills, SNUG Silicon Valley 2013

[9] IEEE Standard for Binary Floating Point Arithmetic. ANSI/IEEE Std. 754-1985, August 1985

[10] https://en.wikipedia.org/wiki/CPU_cache, What is Cache memory?

[11] http://www.pcguide.com/ref/mbsys/cache/funcMapping-c.html, Cache Mapping and Associativity

[12] Cache Performance of the SPEC92 Benchmark Suite, Jeffrey Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, Alan Jay Smith, IEEE Micro, 1993

[13] Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, Norman P. Jouppi, ISCA '90

[14] Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite, H. R. Al-zoubi, Al. Milenkovic, M. Milenkovic, ACM Southeast Conference, 2004

[15] J. M. Thorington, J. D. Irwin, An Adaptive Replacement Algorithm for Paged-Memory Computer Systems, IEEE Transactions on Computers, vol C-21, no. 10, pp. 1053-1061, October 1972

[16] http://web.cs.iastate.edu/~prabhu/Tutorial/CACHE/interac.html, Interaction Policies with Main Memory

[17] Fibonacci and Galois Representations of Feedback-With-Carry Shift Registers, Mark Goresky, IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 48, NO. 11, NOVEMBER 2002

[18] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. IEEE Design & Test of Computers, 28(4):18--27, July 2011