CS 3650 Final Project:
Pipelined Mini-MIPS Processor

Aron Bauer

Sam Paul

Auraiporn Auksorn

Osman Halwani

(1) Introduction

Generic implementation of Pipelined Mini-MIPS Processor uses the program counter (PC) to supply instruction address, gets the instruction from memory, reads registers, and uses the instruction to determine what to do in the Pipelined Mini-MIPS Processor. The design of this implementation supports both memory-reference instructions such as lw and sw and arithmetic-logical instructions such as add, sub, and, or, slt.

There are 14 different instructions implemented, which then are classified into 8 different operations according to what is being performed in the ALU.

The 14 instructions supported are: (1)Addi, (2)Andi, (3)Ori, (4)Slti, (5)Add, (6)Sub, (7)And, (8)Or, (9)Slt, (10)Xor, (11)Sll, (12)Srl, (13)LW, (14)SW.

The ALU performs 8 different operations:

1. add: addi, add, lw, or sw (instruction 1, 5, 13, and 14)
2. or: ori or or (3 and 8)
3. set less than: slti or slt (4 and 9)
4. subtract: sub (6)
5. and: andi or and (2 and 7)
6. xor: xor (10)
7. shift left: sll (11)
8. shift right: srl (12)

The Pipelined Mini-MIPS Processor does not include PC Update as well as ShiftLeft2 because the implementation does not support Jump and Branch instructions.

In a regular MIPS processor, the PC counter is incremented by 4, but in this project the PC counter is incremented by 1 since it simply reads from an array of 32-bits.

(2) Overview

This project implements a five-stage pipelined MIPS processor using VHDL. The design contains major components needed for MIPS processor, along with the instruction decode and control circuits used to connect them all together. The pipeline fetch instructions to execute from the program RAM and increment the program counter by 1, decode an instruction, execute, do memory access for SW and LW, and write the results back to the register file. The project has the implementation of forwarding unit and hazard detection unit even though they were not successfully being included in a five-stage pipelined MIPS processor.

(3) Main Components

3.1 Components in the processor:

- ALU.vhd
    - The ALU performs 8 different operations: add, or, set less than, subtract, and, xor, shift left, and shift right.
- ControlUnit.vhd
    - The Control Unit takes 14 different instructions determined by the opcode.
      (1)Addi, (2)Andi, (3)Ori, (4)Slti, (5)Add, (6)Sub, (7)And, (8)Or, (9)Slt, (10)Xor, (11)Sll, (12)Srl, (13)LW, (14)SW.
- DataMem.vhd
    - The Data Memory implements the functionality for reading and writing data to/from memory.
- InstructionMem.vhd
    - The Instruction Memory provides read access to the instructions of a program, and it takes an address as input and supplies the corresponding instruction at that address. .
- Multiplexer.vdh
    - The Multiplexer Component can be used for all 2-1, 32-bit Mux components in the processor.

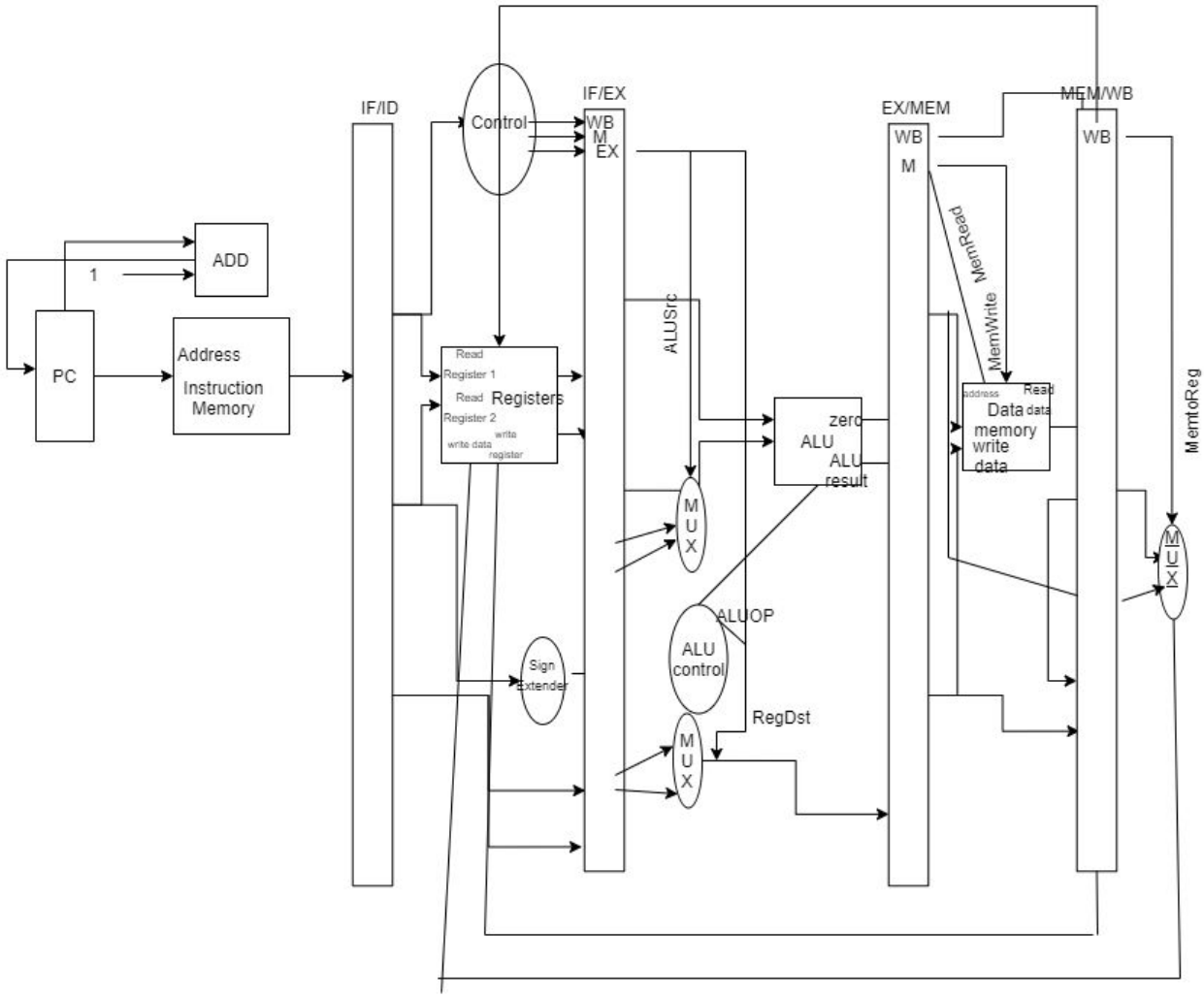
- PCAdder.vhd
    - The regular MIPS processor, the PC Adder increments the program counter by 4. However, PCAdder.vhd only reads from an array of 32 bits in the memory component, so the PC Adder in this implementation PC Adder increments the program counter by 1.
- ProgramCounter.vdh
    - ProgramCounter.vdh is a wrapper for the Program Counter
- Reg_Dest_Mux.vhd
    - 5 Bit 2-1 Multiplexer Component is used for determining the destination register.
- RegisterFile.vhd
    - RegisterFile.vhd contains the MIPS registers. It is a collection of readable/ writable registers.
- SignExtender.vhd
    - SignExtender.vhd performs sign-extending with respect to MSB, and it takes sign extension as input a 16-bit wide value to be extended to 32-bits..
- ALUControl.vhd
    - The ALU Control component determines one of the following eight operations from the opcode.
        1. addi, add, lw, or sw (instruction 1, 5, 13, and 14)
        2. ori or or (3 and 8)
        3. slti or slt (4 and 9)
        4. sub (6)
        5. andi or and (2 and 7)
        6. xor (10)
        7. sll (11)
        8. srl (12)

3.2 Datapath:

- MIPS.vdh
    - MIPS.vdh implements all the components in the processor with five-stage pipelined.
- IFToID.vhd
    - IFToID.vhd gets a 32-bit instruction from memory and takes OpCode, Read Register 1 (rs), Read Register 2 (rt),Immediate Value,Write Register for "I" instruction, and Write Register for "R" Instruction (rd) as the outputs.
- IDToEx.vhd
    - IDToEx.vhd translates opcode into control signals and read registers. It takes ALUop, regDst, memRead, memToRegister, memWrite, ALUsrc, regWrite, Read data 1, Read data 2, Sign extend, Write Register for I instruction, and Write Register for R Instruction (rd) as inputs. The outputs will be the same as inputs
- EXToMEM.vhd
    - EXToMEM.vhd performs ALU operation. It takes memRead, memToRegister, memWrite, regWrite, Read data 2 (write data for memory), ALU result, and value from multiplexer between IF/ID Out 11 and 12 as inputs. The outputs will also be the same as inputs.
- MEMToWB.vhd
    - MEMToWB.vhd updates register file, and it takes memToRegister, regWrite, ReadData, ALUResult, and Value from multiplexer between IF/ID Out 11 and 12 as inputs. The outputs will be the same as inputs. readData and ALUResult will be multiplexed via memToRegister determining which value will be written back to register
- testbench.vhd
    - Testbench.vhd tests the functionality of Pipelined Mini-MIPS Processor implementation. It initializes each clock cycle to be 20 nanoseconds, then it connects the signals in the testbench to the inputs and output in the MIPS entity in Unit under test (UUT) by the test bench. At the beginning

of the process, it resets all the values in the processor. Then, the test itself goes through 18 clock cycles needed for the processor to demonstrate the instructions.

Datapath:



3.3 Test instructions:

There are 5 different instructions that are used as an example to test the MIPS.vhd which contains the implementation of MIPS pipelining. Currently, this project did have the implementation of the hazard detection unit and forwarding unit, but we did not integrate them to a five-stage pipelined MIPS processor.

addi $0, $1, 2 (let $1 = 1, and the answer would be 1+2 = 3)

machine code of addi $0, $1, 2 is: 000001 00001 00000 0000000000000010

In hexadecimal is: 04200002

and $3, $4 $5 (let $4 =1, $5 = 1, and the answer is 1)
machine code of and $3, $4 $5 is: 000111 00100 00101 00011 00000000000
In hexadecimal is: 1C851800

sll $6, $7, $8 (let $7 = 2, $8 = 2, and the answer is 8)
machine code of sll $6, $7, $8 is: 001011 00111 01000 00110 00000000000
In hexadecimal is: 2CE83000

lw $9, 0($1) ($1 = 1, and data will be 5. The answer = 1 with data as 5 being loaded)
machine code of lw $9, 0($1) is: 001101 00001 01001 0000000000000000
In hexadecimal is: 34290000

sw $11, 1($1) ($11 will hold 6. The answer = 2 with data being 6 written)
machine code of sw $11, 1($1) is: 001110 00001 01011 0000000000000001
In hexadecimal is: 382B0001
A table displays multi cycle processor in the MIPS processor implementation of this project.

Instructions	**Clock Cycle**								
	1	2	3	4	5	6	7	8	9
addi $0, $1, 2	IF	ID	EX	MEM	WB				
and $3, $4 $5		IF	ID	EX	MEM	WB			
sll $6, $7, $8			IF	ID	EX	MEM	WB		
lw $9, 0($1)				IF	ID	EX	MEM	WB	
sw $11, 1($1)					IF	ID	EX	MEM	WB

Control and Instruction Decoding Logic

(a) A brief description of a functionality of signal, the values of control signal, and its purpose.

The signal defines the datapath in MIPS processor, and the control signals are group according to the execution activity that they affect. Since in this implementation does not have the activity of PC Update, it contains the activities of source operand fetch, ALU operation, memory access, and register write.

ALUSrc is the signal in source operand fetch activity, and it selects the second source operand for the ALU (rt or sign-extended immediate field).

0: for second operand to be a register

1: for second operand is immediate value (direct value offset for a memory address)

ALUOp specifies the ALU operation to be performed, it will be same as opcode.

MemRead and MemWrite are the signals in the memory access activity as MemRead will enables a memory read for load instructions, and MemWrite will enables a memory write for store instructions.

memRead → 1: for reading memory

memWrite → 1: for writing data

regWrite, RegDst, and MemtoReg are the signals in Register Write.

regWrite → 1 for computed result is written to destination register, in this project only 0 for SW.

RegDst determines how the destination register is specified which could be rt or rd

0: for immediate and LW/SW

1: for register arithmetic

MemtoReg determines where the value to be written comes from which could be from the result of ALU or memory.

1: for computed result is written to destination register

0: for SW in this project

(b) The truth tables

Table 1: Showing the value of the signal for each possible opcode. The control logic signal will take 6 bits of opcode as the input and determines the values of the control signal according to the give instruction type. ALU opcode is the output which is the same as the opcode. Also, the values of regDst, memRead, memToRegister, memWrite, ALUsrc, and regWrite are determined according to the given instruction.

14 different instructions	Opcode	regDst	memRead	memTo Register	memWrite	ALUsrc	regWrite
Immediate Arithmetic: (1)Addi, (2)Andi, (3)Ori, (4)Slti	[000000, 000101]	0	0	0	0	1	1
		0	0	0	0	1	1
Register Arithmetic (Shifts will be register arithmetic): (5)Add, (6)Sub, (7)And, (8)Or, (9)Slt, (10)Xor, (11)Sll, (12)Srl	[000100, 001101]	1	0	0	0	0	1
		1	0	0	0	0	1
Instruction: (13) LW	001101	0	1	1	0	1	1
Instruction: (14) SW	001110	0	0	n/a	1	1	0
No instruction	None	0	0	0	0	0	0

Table 2: Display the value of ALU opcode of each instruction which is the input in the ALU Control that determines one of eight operations from the opcode.

14 different instructions	ALUop	operation
addi	000001	0001
add	000101	
lw	001101	
sw	001110	
ori	000011	0010
or	001000	
slti	000100	0011
slt	001001	
sub	000110	0100
andi	000010	0101
and	000111	
xor	001010	0110
sll	001011	0111
srl	001100	1000
No instruction	None	0000

(4) Challenges

4.1 Challenge #1

Lacking the knowledge of VHDL code. This is our first time learning how to code in VHDL. Fortunately, we had a lecture about VHDL code, but we also had to spend time doing research and learning the functionality of VHDL code.

4.2 Challenge #2

Due to the level of difficulty and complexity of implementing the MIPS processor required a lot of time spent to accomplish this project on top of our responsibilities that we have for other classes. However, we managed to do our best on this project.

4.3 Challenge #3

Due to the complexity of a five-stage pipelined MIPS processor makes the integration of data hazard detection and forwarding unit becomes challenging. However, our group was unable to properly implement them by using the information from the lecture slides even though we currently have them included in the deliverables.