ECSE 425 Final Project - MIPS Processor

Nicolas Grenier 260742552 | Paul Hooley 260727150 | Yanis Jallouli 260854580 | Andrew Lin 260776705

I. Introduction

In this project, we develop a 32-bit MIPS processor with a standard five-stage pipeline using VHDL. As is necessary in such pipelines, intermediate registers store control signals and results after each stage. Given there are no stalls, each stage takes one clock cycle to modify the incoming data and set the outgoing data. Our own implementation successfully performs branch operations, hazard detection, and forwarding.

The input of our processor is a text file named `program.txt`, and the outputs are two files (`memory.txt` and `register_file.txt`) listing the data held in the memory and register files respectively. As per assignment instructions, our main memory holds 32,768 bytes grouped into 8192 lines of 4-byte words. 32 registers were used, each holding 1 word (4 bytes) of data. `program.txt` is assumed to be a list of 32-bit binary instructions (one per line). An assembler was provided to translate assembly instructions to such a format.

In order to simplify the assignment, floating point operations are not supported in this processor, and many other instructions have been removed from the supported Instruction Set Architecture (ISA). A list of supported instructions can be found in Table I.

TABLE I
Supported Instructions

Instruction	Mnemonic
Add	add
Add Immediate	addi
Subtract	sub
Multiply	mult
Divide	div
Set Less Than	slt
Set Less Than Immediate	slti
And	and
And Immediate	andi
Or	or
Or Immediate	ori
Nor	nor
Xor	xor
Xor Immediate	xori
Move From HI	mfhi
Move From LO	mflo
Load Upper Immediate	lui
Shift Left Logical	sll
Shift Right Logical	srl
Shift Right Arithmetic	sra
Load Word	lw
Store Word	sw
Branch On Equal	beq
Branch On Not Equal	bne
Jump	j
Jump Register	jr
Jump And Link	jal

II. Implementation of the Parts

As this processor uses the standard five stages, it was broken into component VHDL files `fetch.vhd`, `decode.vhd`, `execute.vhd`, and `memory.vhd`. Each of these stages performs operations on the rise (or fall for decode) of a global clock. Writeback, while still a stage, was integrated into decode, as the register file is implemented in that stage .

Each component was instantiated and connected (with appropriate registers) in the file `processor.vhd`.

A. Fetch

The first stage, fetch, contains a personal instruction memory instantiated with the data found in `program.txt`. Every clock cycle, it outputs the next `instruction` to be executed, and updates the `program_counter`. If the `branch_taken` flag is set, the program counter is updated to hold

`branch_address`, from which the next instruction will be fetched.

TABLE II
PORTS OF THE FETCH STAGE

Name	Type	Width
clock	in	1
branch_address	in	32
branch_taken	in	1
instruction	out	32
program_counter	out	32

Fetch is responsible for clearing the pipeline when jumps or branches occur. To keep things simple, when a branch instruction is fetched the stage will output two 0x0 values (empty instructions). At the end of the second cycle the execute stage will have computed the target address as well as the branch condition.

B. Decode

Decode is responsible for analyzing the incoming instruction, reading the registers, managing the execution of later stages, and writing the result of the instructions to the appropriate register, being merged with writeback in that regard.

It keeps track of the target of the three previous instructions using `execute_target`, `memory_target` and `writeback_target`.

At each cycle, if `writeback_target` is not zero, it writes the value being outputted by the memory stage to the register.

Hazards are detected by checking if an instruction will read a register which will be written by an instruction in the execute or decode stage.

Instructions in the execute stage can either produce their results immediately or in the memory stage. In the former case an internal flag `execute_result_available_execute` is set to 1. If an instruction being processed by decode reads a register whose value will only be available in the memory stage (a load operation most notably), then the stage will insert one stall (by setting all of the appropriate signals to 0).

Since memory addresses are computed in the execute stage, if the memory address of a load or store depends on the target register of an instruction currently in the execute stage but whose output will only be available in the memory stage (*i.e. a load instruction*) then the decode stage will insert one stall.

Other than that, no stalls are required between instructions having data dependencies since forwarding can take care of bringing unwritten results to any previous stage.

Forwarding is done by sending control signals to subsequent stages. The execute stage can be instructed to use the previous result of the execute stage as one of its inputs using the `execute_1_use_execute` and `execute_2_use_execute` signals. The signals `execute_1_use_memory` and `execute_2_use_memory` instruct it to use the result of the memory stage.

The memory stage can be instructed to use its previous result by the `memory_use_memory` signal. Depending on whether the operation is a load or a store, the signals `memory_read` or `memory_write` are set. If neither are set, the memory stage simply outputs the result of the execute stage. Whether these signals are to be set is decided internally before the execute stage. They are written to an internal delayed buffer so that they reach the memory stage one cycle later (*i.e.* when the instruction has reached that stage).

TABLE III
PORTS OF THE DECODE STAGE

Name	Type	Width
clock	in	1
instruction	in	32
write_data	in	32
stall_fetch	out	1
read_data_1	out	32
read_data_2	out	32
immediate	out	32
opcode	out	6
funct	out	6
shamt	out	5
memory_read	out	1
memory_write	out	1
execute_1_use_execute	out	1
execute_2_use_execute	out	1
execute_1_use_memory	out	1
execute_2_use_memory	out	1
memory_use_memory	out	1

C. Execute

The execute stage is responsible for the computational portion of instructions. The bulk is an ALU composed of adders, multipliers, bit shifters, and more. The operation and operands are defined by control signals from the decode stage, and the outgoing signals are a result, a PC address, and a control signal for whether the PC address should be used in place of a sequential update.

TABLE IV
PORTS OF THE EXECUTE STAGE

Name	Type	Width
clock	in	1
opcode	in	6
shamt	in	5
funct	in	6
read_data_1	in	32
read_data_2	in	32
immediate	in	32
execute_1_use_execute	in	1
execute_2_use_execute	in	1
execute_1_use_memory	in	1
execute_2_use_memory	in	1
memory_result	in	32
program_counter	in	32
result	out	32
memory_write_data	out	32
branch_address	out	32
branch_taken	out	1

D. Memory

Memory is where all interactions with data memory take place.

If neither `memory_read` or `memory_write` are set, this stage simply output the value of its `address` input, which is the result of non-memory operations from execute (*i.e.* arithmetic operations).

The data to be written is taken from the execute stage's output `memory_write_data` or this stage's own previous result if the `memory_use_memory` signal is set.

TABLE V
PORTS OF THE MEMORY STAGE

Name	Type	Width
clock	in	1
memory_read	in	1
memory_write	in	1
memory_use_memory	in	1
address	in	32
write_data	in	32
result	out	32

E. Hazard Detection/Forwarding

As mentioned, forwarding and hazard detection are done in the decode stage. Each stage has a target register, and decode analyzes whether the incoming instruction requires data from these targets. If so, a MUX within the execute and memory stages can choose to use the results of the current (or future) stage as an incoming operand.

F. Processor

Finally, all of the stages were combined in `processor.vhd`, the final 5-stage MIPS processor component. The entity itself merely acts as an interconnection between instantiations of each stage. The processor component, once instantiated, will begin reading from `program.txt` on every clock cycle. Computation outputs after terminating the program can be observed in `register_file.txt` and `memory.txt`.

TABLE VI
PORTS OF THE PROCESSOR

Name	Type	Width
clock	in	1
reset	in	1

III. UNIT TESTING

For the unit tests the goal was to ensure that each function, branch and switch condition across each VHDL file was tested. In order to do this a testbench file was created which corresponds to the pipeline section that it was testing. Although getting code coverage statistics for the each .vhd file is not possible with the version of ModelSim that we

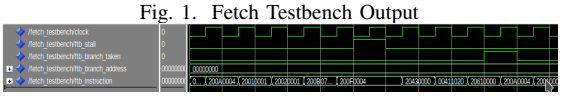
are using, our group still attempted to reach every statement through our tests.

A. Fetch

This testbench involved letting the fetch stage go on normally at first. Then after a few cycles the `stall` signal was raised to observe if the stage stopped fetching new instructions (as was expected). Subsequently, the target branch was set to 0x0 and the branch taken signal was raised. Under all of these conditions, the component behaved as expected.

This can be seen in figure 1.

Fig. 1. Fetch Testbench Output



B. Decode

As the most complex stage, testing decode was incredibly taxing. Not only did various data hazards have to be input to determine if control signals reacted as they should, but it was important to ensure that all operations were handled in this stage. While not every test case could be run, a select few were performed, tweaked as we went along in an attempt to uncover new errors.

C. Execute

The purpose of this unit test was to ensure that each instruction would generate the appropriate output. In order to do this we tested each opcode with sample inputs and reported the results.

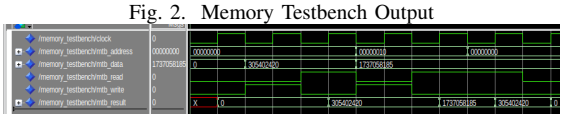
D. Memory

To test the memory, the value 0x12341234 was first written to address 0x0. That same address was then read to see if the value had been properly stored.

Then, the value 0x67896789 was written to address 0x10.

Finally, the address 0x0 was re-read to see if the intially-written value was still present. It was. This can all be seen in 2

Fig. 2. Memory Testbench Output

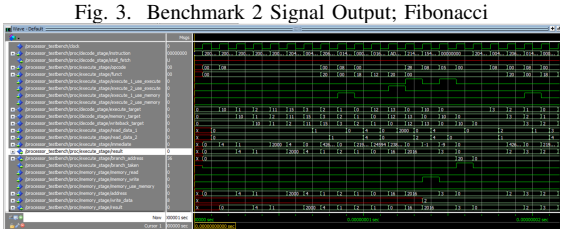


IV. End to End testing

Once individual components were thoroguhly tested, many programs were created in assembly to test `processor.vhd`. Each program was assembled into a binary `program.txt` file by the provided assembler. Within each program, the expected results (of the register files and memory) were noted, and compared with the calculated results after running a testbench of the processor entity.

The first test program used was a basic one to ensure the pipeline was operational. It simply looped repeatedly, writing the address of each memory location into its contents. So 4 would be written to memory address 4, 8 would be written to address 8, etc. After debugging the processor to properly execute this program, a provided benchmark program was used to test the pipeline was functional on a larger scale. This program computed the fibonacci series and stored it in memory. Again, at this point tests were meant to prove the pipeline was operational in standard situations.

Fig. 3. Benchmark 2 Signal Output; Fibonacci



After these tests, a series of hazard detection tests were run. Individual programs tested execute using execute results, execute using memory results, memory using memory results, and memory using writeback results. Finally, a large test file was created to test every possible stall we could think of, including forwarding to different ports, detecting hazards on branch instructions, and creating hazards on both the write data and the register used for addressing in memory operations.

With forwarding signals resolved, the final test completed was a large, but simple, test to ensure that every instruction implemented in the processor functioned properly. Each operation was conducted at least once, and the results were compared to those calculated by hand. After each operation, the result was stored in memory to make comparing answers simple.

A list of expected results for each self-made test program can be found at the bottom of the assembly (asm) files.

V. Conclusion

Many issues arose while completing this assignment, but in the end, we were successfully able to create a standard five-stage MIPS pipeline that supported hazard detection and forwarding. There are many areas for improvement that were not explored during this project, such as branch-prediction schemes or dynamic scheduling. The processor, however, functions as expected.