Final Report Group 1

Mu-I Chen 260410774 Kornpat Choy 260454385 Andrew Jesson 260196340 Meng Yin Tao 260480207

***Abstract*—The project implements a 5-stage pipelined processor and an assembler. The assembler turns the MIPS assembly code into 32-bit binary machine code instructions. The instructions propagate through the pipeline. Hazards are detected along the pipeline and dependent data are forwarded when possible.**

I. Problem Statement

The processor implemented in this project translates MIPS assembly code into binary machine code. The instructions are executed using a 5-stage pipeline as shown in Figure 1 [1]. Control signals (in green), data lines (in blue) and register information (in yellow) propagate through the pipeline. Stall signals (in red) are used to prevent instructions (in purple) from entering the pipeline when hazards are detected.

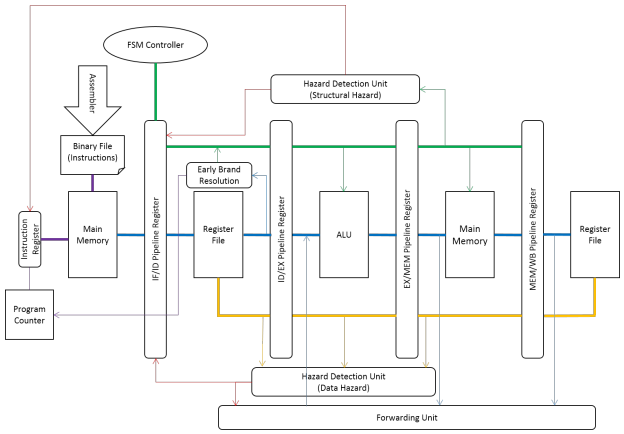

Fig. 1. Pipeline Processor Block Diagram

II. Components

The system consists of two components: an assembler and a processor. The assembler translates the MIPS assembly code into binary instructions. The instructions are stored in the main memory along with the data. The processor accesses the main memory to fetch instructions, load data and store data. The processor is pipelined into five stages and uses hazard detection, a forwarding unit and early branch detection to reduce stalls.

A. Assembler

The assembler functions by first parsing the assembly code into a structured instruction table and then converting each row of the table into binary machine language. We have implemented the assembler in MATLAB.

1) Parsing Assembly Language: To parse the assembly language the input file is first opened and read line by line. For each line comments are removed and labels are checked for. If the line is still not empty the mnemonic of the instruction and arguments are parsed and stored. After this first iteration the assembler then searches for labels in the argument fields and calculates their offset to the label position. The resulting instruction table for fib.asm is shown in Table I. Aside from the arguments and names of each instruction, an array is stored identifying the basic block that the argument belongs to. Inner blocks are stored after Arg 3 and more general blocks are stored proceeding from left to right.

TABLE I
Instruction Table Result of Parsing fib.asm

Mnemonic	Type	Arg 1	Arg 2	Arg 3	Block 1	...	Block N
addi	I	$10	$0	4			
addi	I	$1	$0	1			
addi	I	$2	$0	1			
addi	I	$11	$0	2000			
addi	I	$15	$0	4			
addi	I	$3	$2	0	loop		
add	R	$2	$2	$1	loop		
addi	I	$1	$3	0	loop		
mult	R	$10	$15		loop		
mflo	R	$12			loop		
add	R	$13	$11	$12	loop		
sw	I	$2	$13	0	loop		
addi	I	$10	$10	-1	loop		
bne	I	$10	$0	-9	loop		
beq	I	$11	$11	-1			

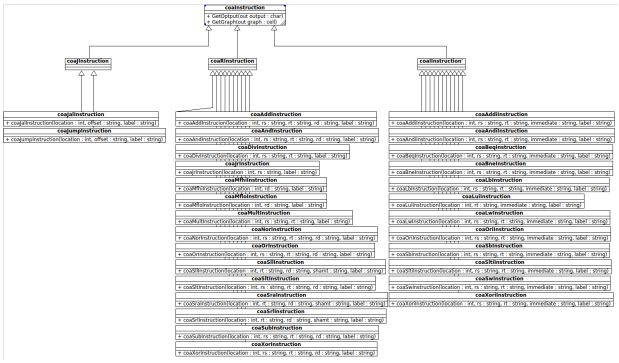

Fig. 2. Class Hierarchy of Assembler Instruction Objects

2) Converting Instruction Table to Binary Machine Language: For this procedure we take an object oriented approach. This design decision was made to support easy extension of instructions not included in the specification. The class hierarchy of the instruction API is shown in

Figure 2. The public class constructors take the form of coa¡mnemonic¿Instruction(args) and return an Instruction object, I. The args for each type of instruction are taken from the above described instruction table. The public I.GetOutput() method returns the line of binary machine language corresponding to the type of instruction and arguments passed. The public I.GetGraph() method returns a structured array of the position of the instruction within the program, the register it writes to, the registers it reads from, and minimum and maximum indices of the basic block it belongs to. This method has been designed to facilitate instruction rescheduling. Table II represents the graph that would correspond to fib.asm after the initializing a static rescheduling routine with the graphs from each instruction. The routine would then maximize the distance between the true data dependencies with constraints determined by the minimum and maximum positions, along with the anti and output dependencies.

TABLE II
FIRST ITERATION DEPENDENCIES GRAPH FOR FIB.ASM

Pos.	Write	Read 1	Read 2	Min Pos.	Max Pos.	True	Anti
1	10	0		1	5		
2	1	0		1	5		
3	2	0		1	5		
4	11	0		1	5		
5	15	0		1	5		
6	3	2		6	13	3	
7	2	2	1	6	13	3	6
8	1	3		6	13	6	
9		10	15	6	12	1,5	
10	12			7	13		9
11	13	11	12	6	13	4,10	
12		13	2	6	13	11,7	
13	10	10		6	13	1	9
14		10	0	14	14	13	
15		11	11	15	15		

B. Instruction Fetch (IF) Stage

The first stage of the pipeline consists of fetching an instruction from memory and updating the program counter (PC). By convention, instructions are stored sequentially and the first instruction starts at memory address 0x00. Every instruction is 32 bits long and memory is address by byte; therefore, instructions are separated by a 4 memory address offset.

1) PC: The PC is programmed to increment by 0x04 at every clock cycle and used as a pointer to the next instruction . When a control flow instruction is detected, such as a branch or a jump, the target address is computed. A select signal is set high to take the target address and the PC is updated to use the new address for its next computation.

C. Instruction Decode (ID) Stage

Once the instruction is fetched from memory, it is decoded in the ID stage, where the OPCODE is sent to the pipeline controller. Depending on the current OPCODE and the last state, the pipeline controller will issue a series of control signals. Depending on the type of MIPS instruction (R-type, I-type or J-type), the register file puts the corresponding data onto the register data line. To reduce stalling after a branch instruction, we implemented an early branch resolution unit. When a branch instruction is detected during the ID stage, the PC counter will first be instructed to update its value to the target address, then the IF/ID pipeline register will be warned that a branch instruction was detected, and finally the last instruction fetched will be flushed. The decoded instruction information and data are passed to the next stage through the ID/EX pipeline register.

1) Pipeline Controller: The pipeline controller is the most crucial component in the processor as it turns the OPCODE into control signals that oversees the execution of the pipeline. All control signals are updated during a single clock cycle and then fed to the IF/ID pipeline register. Control signals propagate through the pipeline and may be changed if a hazard is detected.

2) Register File: This processor implementation uses 32-bit data registers. The register file stores the register values and is accessed depending on the instruction type. The first register (R0) is hardwired to 0x00. The register file reads its inputs on every rising clock edge and updates the corresponding registers. On the falling edge, the register file outputs the data requested. Therefore, during a single clock cycle, data can be written to and read from the register file. This implementation avoids stalling the pipeline when two instructions (one read and one write) need to access the registers.

3) Early Branch Resolution Unit: Branch hazards may cause the pipeline to fetch the wrong instruction when a branch prediction is wrong. In the current implementation, branches are predicted as not taken and the instruction stored immediately after the branch instruction is fetched while the branch instruction is in the ID stage. As the branch instruction propagates down the pipeline, more instructions are fetched until the branch is resolved. To avoid fetching extra instructions whose execution depends on the branch condition, an early branch resolution is implemented. It uses a comparator and resolves the branch condition in the IF stage. By doing so, it saves one clock cycle in the case when the branch prediction is wrong (branch is taken) and does not affect performance when the branch prediction is correct

D. Execution (EX) Stage

After the instruction is decoded, the operands are written to their respective data lines. The arithmetic logic unit (ALU) can now perform computation when needed. The inputs of the ALU depend both on the type of instruction and also when data is forwarded and are selected accordingly. The result of the ALU is passed on to the next stage through the EX/MEM pipeline register.

1) ALU: All computations are performed by the ALU. The ALU takes two 32-bit inputs and returns a 32-bit result. The selection of input is managed by the pipeline controller and the forwarding unit. With the exception of MULT and DIV instructions, the result of the ALU computation is available

after a single clock cycle. MULT and DIV instructions are expected to be followed by MFLO or MFHI instructions.

E. Memory Access (MEM) Stage

Other than fetching an instruction, memory can be access only during a LOAD or STORE instruction. When an instruction arrives at this stage, the control signals will be used to determine if memory access is needed. The data loaded from the memory, is passed on to the next stage through the MEM/WB pipeline register.

1) Main Memory: The main memory module used here was provided by the course TA. Unlike the register file, data cannot be written to and read from the main memory at the same time. This causes the pipeline to stall (disable data fetching) when a LOAD or a STORE instruction in its MEM stage. Due to the slow access time, main memory requires several clock cycles before returning the requested information.

F. Write-Back (WB) Stage

The final stage of the pipeline updates the register file with the computed value or the data loaded from memory. When an instruction arrives at this stage, the control signals will be used to determine if register access is needed. As previously mentioned, data can be written to and read from the register file in the same clock cycle, so no stall will be introduced.

G. Pipeline

In order for the pipeline to function properly, the information of each stage is stored in the pipeline registers. With the intermediate values stored, it becomes possible to implement additional components to reduce the stall time between instructions such as, hazard detection and data forwarding.

1) Pipeline Registers: Registers are essential to the functioning of the pipeline. A pipeline register couples each stage and stores the information of an instruction. For instance, when an instruction moves from the ID stage to the EX stage, its control signals, data and register information are written to a pipeline register. These values kept in the register until the next clock cycle, where the instruction exits the EX stage. This register sits between the ID and the EX stage, therefore is called the ID/EX pipeline register. There are a total of five pipeline registers in this implementation: IF, IF/ID, ID/EX, EX/MEM and MEM/WB pipeline registers and so there are five instructions in the pipeline when the pipeline is fully loaded without stalls. All registers hold their values for one clock cycle and are updated at every rising edge of the clock.

2) Hazard Detection Unit: As mentioned earlier, data cannot be written to or read from memory at the same time. This represents a structural hazard that requires the pipeline to stall until memory access is completed. A structural hazard detection unit is used to verify the control lines in the EX stage. If the instruction in the EX stage need to access memory (either read or write) in the next cycle, the structural hazard detection unit will issue a stall signal to disable instruction fetch at the next clock cycle.

Data dependencies can also affect the performance of the processor in the case of a read-after-write (RAW) dependency. A data hazard detection unit is used to compare the input of one instruction against the output of the previous two instructions. When the instruction in the EX stage requires the ALU output of the instruction one cycle ahead of it, then a EX-to-EX forwarding is required. When the instruction in the EX stage requires the ALU output or the memory output of an instruction two cycles ahead of it, then a MEM-to-EX forwarding is required.

For both structural hazard detection unit and data hazard detection unit, all instructions following the instruction causing the stall are paused and only resume execution once the hazard is resolved.

Branch dependencies are not explicitly handled in the implementation. The early branch resolution unit is the only component that will reduce stall by one clock cycle in the case of a wrong branch prediction as explained earlier.

3) Forwarding Unit: When data hazards are detected, the forwarding unit is signaled to forward data to the input of the ALU in the EX stage. Depending on the type of forwarding, the appropriate input is selected for the ALU. A EX-to-EX forwarding will feed the ALU output directly back to its input. A MEM-to-EX forwarding will feed the output from the memory back to the input of the ALU. The forwarding unit avoids waiting for data to be written back to register before it becomes available to the following instructions. The forwarding unit also forwards to the early branch resolution unit, where the target address can be resolved earlier.

H. Optimization

1) Branch Predictor: A simple branch predictor can be implemented to predict branch not taken. It will reduce the number of wrong predictions in the case of looping. This branch predictor will be added early in the pipeline to update PC to the target address before knowing the outcome of the branch instruction. More complex branch predictors can also be used.

2) Split Level-1 Cache: A main cause for stalling is the long memory access time. To reduce waiting time, caches can be implemented. A split cache would be best because it would allow instruction fetching at the same time of a LOAD or STORE instruction accessing memory.

3) Instruction Rescheduling: Optimization could also be done by the assembler. A better assembler could detect independent instructions and insert them between dependent instructions that require extra clock cycles between them. The assembler could also detect loops and perform loop unrolling to allow more rescheduling possibilities.

4) Branch Delay Slot: nstead of using branch prediction, the branch delay slot can be populated with an instruction which will not affect the branch instruction. Since in the current implementation the instruction following the branch instruction is fetched regardless of the branch resolution, it could be replaced by an instruction that was originally before the branch instruction. This rescheduling of instructions must be done by the assembler. The assembler must make sure

that the instruction in the branch delay slot will not affect the branch instruction outcome.

III. System Evaluation

The pipelined system was successfully implemented for our project. Here we show relevant example outputs that show specific processor functions.

In Figure 3 we can see that the program counter successfully increments by 4 on the next rising edge of clock signal which displays that it is functioning properly.

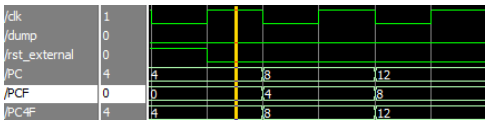

Fig. 3. Incrementing the program counter.

Figure 4 shows that the inputs to the ALU are returning the correct results at its output. Here we show three instructions being executed, the first instruction is an addi, followed by an add and then another addi. Our processor correctly chooses the right signal using the decoded instruction.

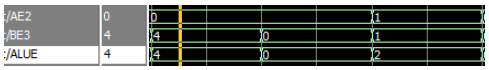

Fig. 4. Testing the ALU with an add operation.

Figure 5 shows that when a store instruction is executed, our processor stalls for one clock cycle and so when a stall occurs the PC and INSTR are present for a total of two clock cycles. We can see this on the rising edge of the clock when the program counter is at 56. The representation of the Instruction is in decimals in this case to save space.

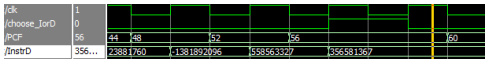

Fig. 5. Demonstrating stalling on load and store operations.

In Figure 6, we can see the propagation of the RegWrite control signal through the pipeline. The last letter in the signal name specifies in which section of the pipeline the signal is in. Therefore, we can see the same signal propagating from the decode stage all the way to the write back stage. We can also see that the signal is delayed by one clock cycle per stage which is what is expected in a pipelined processor.

Fig. 6. Demonstrating propagation of control signals through the pipeline

In Figure 7 we show one of our larger tests to determine correct instruction decode and execution. We used the instruction: addi \$10, \$0, 4 where the value in \$0 is 0. The instruction is correctly decoded from our assembler and in our processor the corresponding ALU instruction is fed into the ALU; which is an add instruction. Rs and Rt are also selected correctly with Rt as the destination register and Rs as one of the sources. We can also see that the Immediate value is 4, which is what was required. Following the decode stage we can see that the values being fed into the ALU are again correct and happen one cycle later. We can also verify that the output of the ALU is correct as 0 + 4 = 4 as desired.

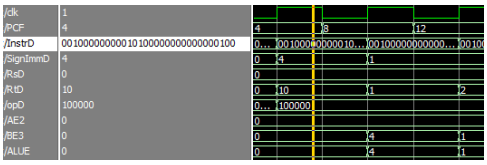

Fig. 7. Demonstrating instruction decode and execution.

IV. Recommendation

As we look back on how the project was approached, the main challenge was to get the first implementation to work. If we had the opportunity to start the project all over again, we would like to perform functional tests on each individual components before assembling the entire processor. We have under-estimated the time needed to both assemble and test the processor. After combining all components in a single VHDL file, we had many errors preventing the program from simulating. It was hard to find out what might have caused each issue due to the complexity of the connections and the multiple number of components. For the second deliverable it was particularly hard to insert new components. Instead of modifying the first deliverable, we decided to restructure the entire processor. To avoid having to debug or modifying a large file, we think it would be preferable to have multiple small deliverables. We propose to breakdown the project into pipeline stages instead of changing a functional implementation into a pipeline. For instance, the first deliverable could consist of fetching and decoding the instruction. This deliverable will focus on on accessing the memory, interpreting the instruction code, and generating control signals. The second deliverable could be the execution which would aim at creating an execution stage that takes the right input and performs the correct operation depending on the control signals. The third deliverable could be the memory access and hazard detection. During this deliverable, we will detect hazards and stall the pipeline when needed. Finally, the last deliverable will consists of the write-back and forwarding. We think by building the project stage by stage, it would be easier to debug knowing that the previous stage is working properly.

References

[1] D. Patterson and J. Hennessy, *Computer Organization and Design*, 4th ed. Waltham, MA: Morgan Kaufmann, 2012.