Pipelined processor in VHDL with hazard detection and forwarding

Stefan Barbu
ECSE Department, McGill University
Montreal, QC, Canada
260806224

Michael Frajman
ECSE Department, McGill University
Montreal, QC, Canada
260863814

Shi Tong Li
ECSE Department, McGill University
Montreal, QC, Canada
260857759

***Abstract*—The goal of this project will be to create a five-stage pipelined processor using VHDL. The processor should be able to read from a set of integer MIPS assembly instructions. It should also be able to stall when hazards are detected as well as forward data. The processor should be robust enough to handle large amounts of instructions during various tests with outputted memory and register test files being used to evaluate correctness.**

Keywords—pipelined processor, VHDL, MIPS, hazard detection, data forwarding

I. Introduction

The goal of the project is to create a five-stage pipelined processor capable of hazard detection and issuing stall instructions as well as data forwarding. The five-stages more specifically are: 1) instruction fetch (IF) which retrieves instruction from main memory, 2) instruction decode (ID) which manages the register file and serialized the retrieved instruction word to facilitate further operations, 3) execution (EX) which performs any potential arithmetic operations, 4) memory access (MEM) which writes to memory on stores, and 5) writeback (WB) which sends data back to the ID stage to be stored back to the registers. The design is created in VHDL and uses a reduced set of 27 MIPS integer instructions. The word length is 32 bit and there are 32 integer registers.

Our VHDL files are structured around the five aforementioned stages. Each stage has its own vhd file containing its core logic. The pipelined processor vhd file is the central file of the project and connects all five stages using port maps. We were able to get most instructions to work for small test programs we created. Hazard detection and stalling were implemented however forwarding was not implemented. Text files are used to inputting and outputting memory and register data and tcl is used for automating testing.

II. Methodology

A. Overall plan

Our team decided to generally follow the recommended file structure as outlined in the assignment document. We had a pipelined processor vhd file as well as a separate vhd file for each of the five stages, IF, ID, EX, MEM, and WB. The stages were treated as components with the processor file. We also had an instruction memory vhd file modelled after the cache project's files for reading in text inputs. The instruction memory file was a component of IF. A tcl file was used to automate testing. We went with the recommended file structure as the breakdown made natural sense given the complexity of some of the processor stages and it saved time in not having to reinvent the wheel.

In working on the project, we tried to complete and partially test each stage individually from ID to WB one after the other. This was done to ensure that signals were propagating through the stage and to test the clocks timing. We were able to catch major clock problems where data was only propagated every alternate clock cycle early on by doing this. We also outlined the entities' inputs and outputs from the start before writing their architecture in order to ensure consistency and better visualize the requirements.

B. Instruction Fetch (IF)

In the instruction fetch stage, we use an instruction memory (IM) component. The IM component reads from the "program.txt" file to initialize the byte addressable memory. The 32-bit instructions from the

file are divided in four byte-sized vectors to be stored in the memory. The instruction is stored big-endian and the memory block itself has decreasing indexes. Since an instruction is 4 bytes long, it is 4-aligned and a multiple of four address indexes the instruction. When reading, the IM component concatenated the block at the address provided and the next three indexes to return a 32-bit long instruction.

The IF stage simply reads the instruction stored at the program counter in the IM component each cycle and propagates it to the ID stage with no memory delay. At each cycle, there are three behaviors for the PC: 1) it may increment by four, which is the default behavior, when there are no stalls, or the jump target bit is not set, 2) it may keep PC to the same value when the stall bit is set, or 3) it may set PC to the jump target provided by the ID stage when the valid jump target bit is set.

At each cycle, IF also propagates PC+4 to ID, so ID can resolve jump targets for control flow instructions.

C. Instruction Decode (ID)

The ID stage contains the register block that stores register values. It also decodes the raw instruction data that will feed the EX stage inputs. It is also ID that decodes the jump targets and resolves branch conditions.

It is important to store the register block as a process variable rather than a component signal so we can write to them and read back the update value within the same clock cycle.

To decode the raw instructions, ID differentiates between function/register instructions that have "000000" as their opcode and all other kinds of instructions. For other instructions, the opcode is propagated in the pipeline, so each stage knows what to do, whereas for function instructions, the function, i.e. the last 6 bits of the instruction, is propagated. These fields never collide so there is no ambiguity. There would have been collisions between jump and shift-right-logical, and jump-and-link and shift-right-arithmetic, where the opcode is the same as the function, however control instructions are resolved in ID, so stalls get propagated instead. It is important to note, we noticed the provided compiler uses a different funct field for XOR than the MIPS Green Sheet, so we followed the compiler's code "101000" rather than MIPS' "100110".

Once the instruction has been identified, we follow the MIPS Green Sheet to 1) set the register reference to the register to update with the result, 2) set the data lines for the EX stage with the appropriate register values for register type instructions, and 3) set the data lines for the EX stage with the sign-extended immediate values from the raw instruction for immediate type instructions.

Control instructions implement a different behavior than explained previously. Since they are resolved in ID, a stall rather than the decoded instruction is propagated to the next stages. The jump target is calculated from the PC value from the IF stage. Instructions J, JAL, BEQ and BNE compute the jump target from the four most significant bits of PC concatenated with the address from the jump-type instruction. The address corresponds to the 26 lower bits of the instruction logically left-shifted by two. Instruction JR sets the jump target to the register value.

For unconditional jumps, J, the valid jump target bit is always set. Instruction JAL also save the value of PC+8 in the 32rd register. For conditional branches, it is only when the register values fulfill the condition that the valid jump target bit is set.

Whenever setting the jump target and indicating it is valid, we also set a variable local to ID to indicate that the next instruction should be ignored. This is important because the IF stage will have been fetching the next instruction while ID was resolving the jump target, so it does not correspond to the instruction at the jump target and should be ignored. It is the instruction in the delayed slot, as if using a static predict not taken branch predictor.

D. Execution (EX)

The Execution stage's logic is condensed into the EX_stage.vhd file. The current instruction is passed to the MEM stage, along with the register reference, the output of the ALU, and another line to pass the contents of a register if store (SW) needed to be done. However, there is no separate ALU component in our code. Instead, its core logic is injected in our rising edge clock process inside EX_stage.vhd and depending on the current instruction, dictated by the MIPS Opcode Reference, does operations on incoming data. For example, if the current instruction is "100000", which are the funct bits of an Add operation, we will add the incoming two data lines together and output it to the next stage. Local signals are also declared to store the result of a multiplication and division into "hi" and "lo", two signals of std_logic_vector of 32 bits that represent the hi and lo registers. Finally, our execution stage does not handle branches (BEQ, BNE) and jumps (J, JR, JAL). The EX stage never expects to see control instructions because they are resolved in the ID stage and replaced with stalls.

E. Memory Access (MEM)

For the memory stage, we have created a ram block that is 8192 lines long (8192*4 = 32768 bytes) using vector arrays similar to the code given in the cache

project assignment. There is no memory delay, so the value is either read or written to the memory block instantaneously. The clock process will check the current instruction data passed by the previous stage, and if it is a load (LW = 10011) or a store (SW = 101011), it will perform the appropriate actions on our vector array ram block. If it is not a memory operation, it would mean that the current received immediate data is meant for the writeback stage and will be passed onto the next stage without any modifications.

F. Writeback (WB)

The logic of the writeback stage is quite simple compared to other stages. Its goal is to set the bit that controls writing to registers inside the ID stage to high when the current operation is a register or immediate type that needs register writeback (for example: ADD, ADDI, SUB, etc.). If there is no writeback to register needed, the control bit is set to 0.

G. Hazard Detection and Stalling

For hazard detection, we have implemented its logic inside the pipelined_processor.vhd file. It essentially checks the incoming instruction out of the instruction fetch stage and compares the instruction's registers (rs, rt and rd if it is a register type or rs and rt if it is an immediate type) to the register reference in the ID, EX, MEM, and WB stages. If one of the registers listed inside the incoming instruction is also referenced in the stages mentioned previously, a hazard is then detected. The bit that controls stalling is set to high and will initiate stalling in the IF and ID stages. Stalling for the instruction fetch stage is done by passing the current value of the program counter onto the next clock cycle instead of incrementing the program counter by 4, As for the ID stage, stalling works by feeding the equivalent of the operation ADD $0 $0 $0 in binary to the EX stage, and the EX stage will propagate the stall to the next stage and so on.

III. RESULTS

A. Benchmark 1

Benchmark 1 was able to be successfully run using our processor. Our outputted register and memory files matched those that were provided. This shows that the ADDI and BEQ instructions were able to be successfully executed in scenarios with no hazards.

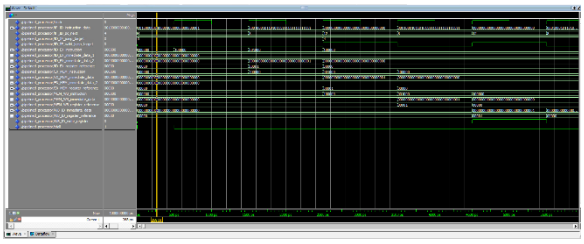

Fig. 1. Wave output from ModelSim for Benchmark 1, all signals are properly initialized.

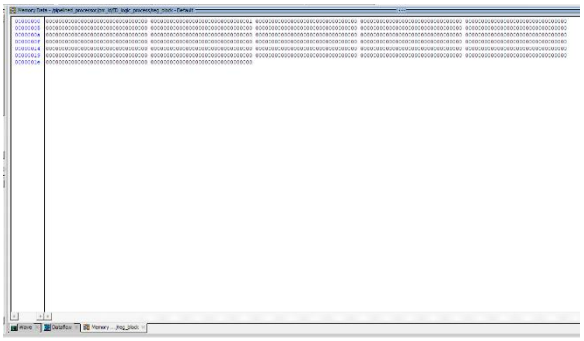

Fig. 2. Register viewer output from ModelSim for Benchmark 1, register 1 contains a value of 1.

B. Benchmark 2

Benchmark 2 did not run successfully with incorrect values being placed in both memory and the registers and not all in the correct locations. We believe this to have stemmed from an error in delays and stalling between the addi $10, $10, -1 and bne $10, $0, loop instructions. This may be in part due to the data dependency of $10 and there being not enough delay for the value to be calculated. This is made worse by our processor not implementing forwarding. Overall, our processor exhibited issues with timing and stalling in regard to branch instructions, particularly for bne. Figures 3 and 4 depict our memory and register outputs for benchmark 2 using our processor.

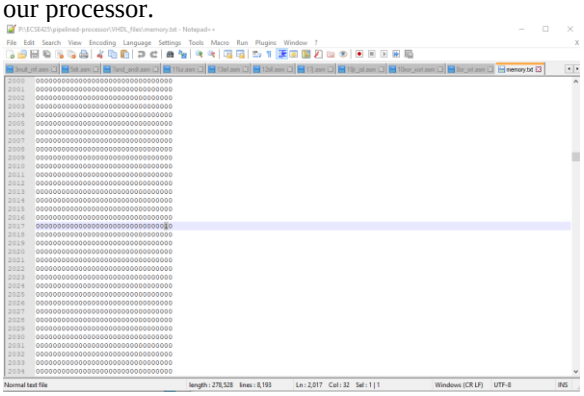

Fig. 3. Memory viewer output from ModelSim for Benchmark 2, a vlaue was written to a single memory location however it was not the expected output from the provided test files.

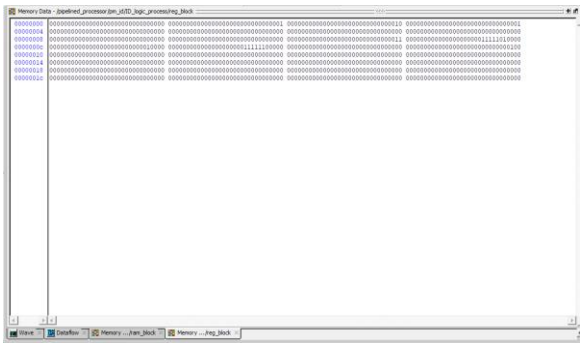

Fig. 4. Register viewer output from ModelSim for Benchmark 2.

C. *Other tests*

Our team created 18 tests to evaluate the basic functionalities of the each of the instructions with some instructions like MULT, MFHI and MFLO, for example, being tested together. An example of one of these tests is shown in the figure below. We included our test files along with our submission.

```
1   addi $1, $1, 1
2   addi $2, $2, 2
3   mult $1, $2
4   mflo $12
5   mult $1, $2
6   mfhi $13
7   EoP: beq $0, $0, EoP
```

Fig. 5. Example of one of our basic testing program, this one for MULT, MFLO and MFHI.

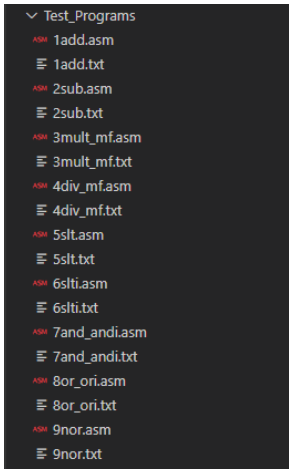

Fig. 6. Part of the directory showing our test files, their names indicate which instructions were having their basic functionality under test.

These tests followed the same basic structure used in benchmark 1 using ADDI to load values into memory and an infinite loop using BEQ to terminate the program. Otherwise only the basic functionalities of the instructions were under test and not necessarily their ability to forward or withstand hazards. These test programs demonstrated that the basic functionalities of most instructions worked with exception to SRA which did not always sign extend properly, and BEQ and BNE which had small problems with delay and data dependencies. The problems with branching were noticeable in non-trivial programs like benchmark 2 where calculations were not fast enough for the branch to have the correct data to compare.

IV. Additional Optimization

No additional optimizations were made. Our processor only attempts to implement the basic functionality requirements.

V. Limitations

Our team was unable to implement forwarding. This in part due to time constraints as debugging stalling and several of the instructions took longer as expected. Additionally, our team was unable to settle on a design for the forwarding mechanism which could fit in with the rest of our processor design. Our ideas for forwarding registers to save and compare intermediate stages of data tried to follow the ideas of pipelining as seen on the course lectures. There would be something along the lines of a “forwarding.vhd” component which would contain arrays to hold the register names of the instructions in the ID, EX, MEM and WB stage. This component would be able to tell which stage the instruction was in and would be able to send data to the stages that needed it as it became available. However, we were never able to smooth out the complexities of this design and we could not find a way to get the timing of something like this to work with the rest of the processor. Thus, we shifted our focus back to stalling and instruction execution to at least improve the functionality of those features.

VI. Conclusion

While we are fairly satisfied with what we were able to accomplish in our processor design, namely getting almost all instructions to function on their own and mostly functioning hazard detection while being able to successfully run smaller programs, we also acknowledge we were unable to fully fulfill all design requirements. Our inability to meet all requirements largely came down to time constraints. While we discussed strategies for forwarding together, we decided that debugging and improving errors in the existing code would be more beneficial. Ultimately,

we felt this time was well spent as we were able to make most instructions achieve basic functionality.