Title - Implementation of a Pipeline Processor using VHDL

Daniel D - ddobos@cse.unl.edu
Guilherme Krzisch - guilhermekrz@gmail.com

Date of completion - 12/10/2012

Abstract

Our group implemented a MIPS processor with four pipeline stages that can run an LCS algorithm. We initially had problems determining what instructions to add to the ISA, this was solved once we finished translating our LCS algorithm from C into MIPS assembly. The main result of our project is a MIPS implementation that includes all but the advanced features described in the project's manifest.

1. Introduction

MIPS type processors follow the RISC(Reduced Instruction Set Computing) philosophy of design, this means that they have few instructions, but those instructions cover all of the common-cases, such as: reading from main memory; writing to main memory; performing arithmetic and logical operations on values stored in registers; and following branch instructions. The simplicity brought by favoring the common case allows us to more readily add advanced features, aids in debugging, and increases the performance of the processor.

To gain a more intimate understanding of processors that use the RISC design principle, we took a basic single-cycle processor and added more advanced features. In the course of our project we learned that it can be a constant struggle to balance the complexity of each component, whether that component is a piece of hardware, or a piece of code.

2. Design

We used an existing single-cycle MIPS processor as the basis for our 4-stage pipelined processor, which implements a subset of the MIPS-I ISA.
Based on the 5-stage pipeline MIPS processor (Instruction Fetch Stage [IF], Instruction Decode Stage [ID], Execute Stage [EX], Control Unit [CON], Memory Stage [DMEM] and Write Back Stage [WB]), we initially decided to implement a 3-stage pipeline, divided in the following way: IF, ID-EX-CON, DMEM-WB. We managed to resolve almost all hazards. The only one that we could not solve was the RAW hazard, where the first instruction is a load from memory instruction. That is because the load instruction takes one more cycle to read the data from memory, so the value will not be available until the beginning of the next cycle.
So we split the last pipeline stage, and the result is shown in the image below:

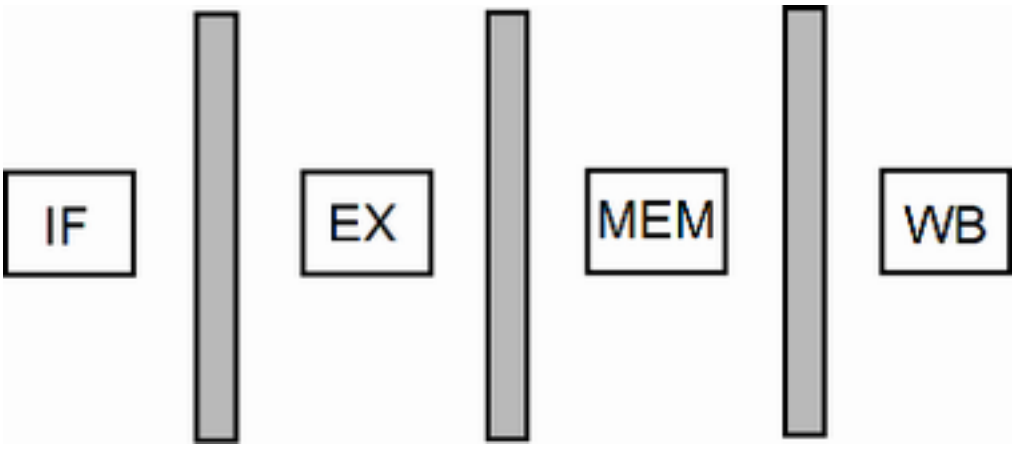

Figure 2: 4-stage pipeline.

In the Instruction Fetch stage, IF, we fetch the instruction of the current program counter. At each cycle, the program counter is incremented by one to reach the next instruction. The IF stage receives the information of the previous instruction saying if it was a taken branch. If yes, the instruction to be fetched is the target branch. If not, the IF stage simple continue fetching the next sequential instruction.

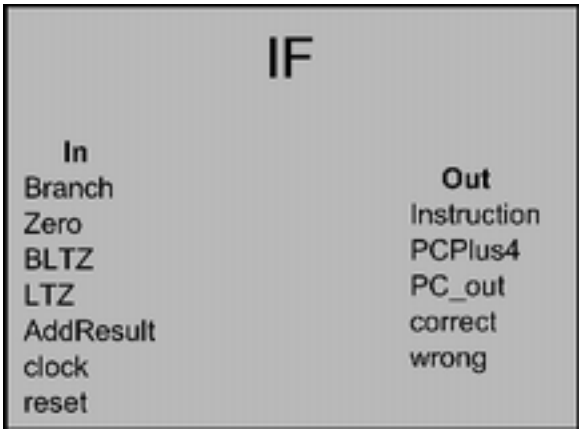

Figure 3: Instruction fetch stage.

The next stage is a combination of two original stages of the 5-stage MIPS processor, along with the control unit (we will call this the ID-EX stage). We did not split this stages because of the large number of connections between them. In the instruction decode unit, we decode the instruction received from the IF stage. Accordingly to the instruction specified operands, we fetch the content of the corresponding registers from the register file. After, this values are sent to the EX unit, where it will perform the defined operation on the ALU (arithmetic-logic unit). Also, the CONTROL unit is setting the flags to send back to the IF stage or forward to the DMEM stage

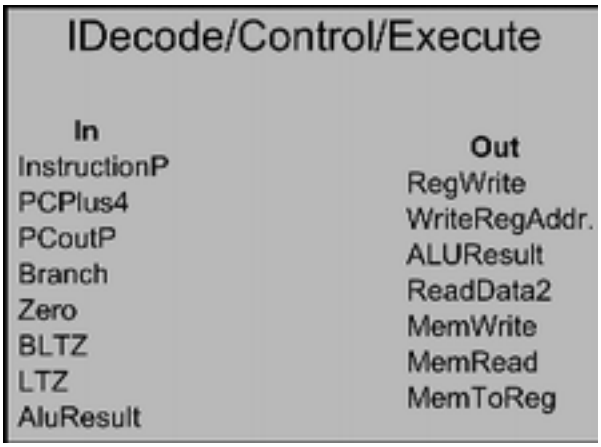

Figure 4: ID/EX stage.

The memory stage is where operations with the data memory occurs. If it is a store instruction, the memory will be written with the value coming out of the previous stage. If it is a load instruction, we will pass the read value to the write back stage. All the others instructions do not perform any operations on this stage, only carrying the required values to the next stage.

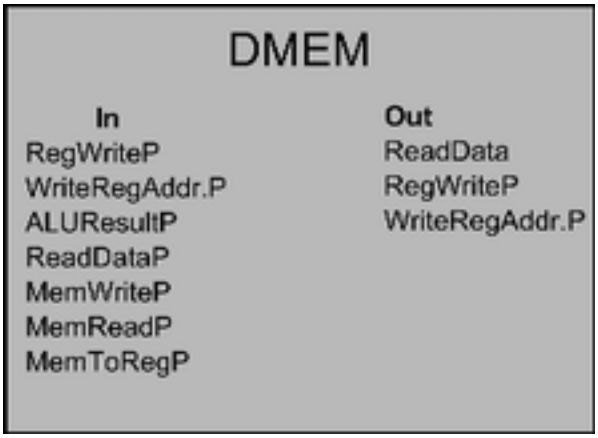

Figure 5: DMEM stage.

Finally, the write back stage is when we write, if any, the value in the corresponding register file.

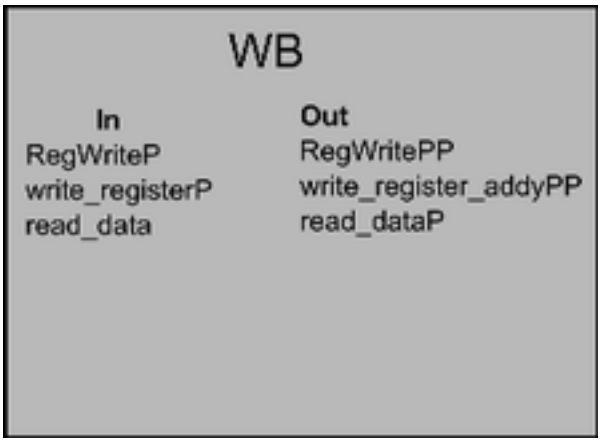

Figure 6: Write back stage.

After implement the 4 stage pipeline, we had to deal with hazards. There are three types of them that we need to consider: structural, data and control hazards.
We don't need to take care of the first one - structural hazards. That is true in part because our processor implements a Harvard architecture (one memory for instructions and one for data), so there is no memory conflicts. Also, two distincts pipe stages don't share resources, for example the ALU unit is only used by the execution stage.
Nevertheless, we have data hazards (RAW - read after write). To solve this, we forward the data from the DMEM or from the WB stage to the ID-EX stage when the destination registers of one the previous two instruction is equal to one or both register sources of the current instruction.

- An example of RAW hazard:
  add **$t0**,$t1,$t2
  sub **$t1**, $t3, $t4
  add $t2, **$t0**, **$t1**

A special case occurs when the previous instruction is a load from memory instruction, so the value will only be available in the beginning of the WB stage, and if the instruction right after the load instruction uses as a source operand the destination of the LW instruction, it needs to stall one cycle.

- An example of RAW hazards, with the first instruction being a load from memory:
  lw **$t0**, 0($t1)
  add $t2, $t0, **$t0**

Finally, there are control hazards. Our processor has a static branch predictor, always not-taken. We choose it because it is simple and efficient to implement (if the branch is not taken, we don't waste any clock cycle, and if the branch is taken, we waste only one clock cycle, flushing the instruction after the branch and fetching the target address of the taken branch).

3. Evaluation of Pipeline

We can empirically test the pipeline's implementation by making two test cases: the first where we make the assumption that there are no hazards, and the second where we assume that there are hazards. If a predictable "hazard" condition manifests under the first assumption and is then relieved under the second assumption--by inserting nops, then we may reason that the pipeline functions.

We can then test the fully hazard-corrected pipeline by removing the nops from our initial test cases.

To show the pipeline fully working, we program the LCS (longest common algorithm) in assembly (based on [1]). Then the assembler generates the program MIF file and the memory MIF file. We input this to our processor, and show the results on the 7-segment displays.
Accordingly to the user input on the board switches, the desired value is shown on the displays.

Switch input	Output on the 7-segment displays
00 0000 0000	Initial Index | Final Index
10 0000 0000	Initial Index
10 0000 0001	Final Index
10 0000 0010	First character
10 0000 0011	Second character
10 0000 0100	Third character
10 0000 0101	Fourth character
10 0000 0110	Fifth character
10 0000 0111	Sixth character
10 0000 1000	Seventh character
10 0000 1001	Eight character
10 0101 0000	Program counter
10 0100 000	ALU result
10 0010 0000	Performance counters - number of correctly predicted branches
10 0011 0000	Performance counters - number of wrongly predicted branches

The displayed characters are from the resulting string of the LCS algorithm, in ASCII. If the string is already over, the 7-segment display will show the null character (0x00 in hexadecimal).

Conclusion

We have constructed a 4-stage pipelined MIPS processor that implements a subset of the MIPS-I ISA, as well as an assembler that produces machine code usable by the processor. We have done research on into which stage each functional unit should go. Our future work is the addition of instruction and memory caches and the improvement of the branch predictor to our current design.

References

[1] - Algorithm Implementation/Strings/Longest common substring. (2012, December 4). *Wikibooks, The Free Textbook Project*. Retrieved 01:26, December 14, 2012 from http://en.wikibooks.org/w/index.php?title=Algorithm_Implementation/Strings/Longest_common_substring&oldid=2452905.