*Computer Organization and Architecture*

Marcel Morin   
Computer Engineering, U3  
260605670  
marcel.morin@mail.mcgill.ca

Uday Sahni  
Computer Engineering, U3  
260614324  
uday.sahni@mail.mcgill.ca

Stone Yun  
Electrical Engineering, U3  
260616314  
stone.yun@mail.mcgill.ca

Richie Piyasirisilp  
Computer Engineering, U3  
260624968  
master.piyasirisilp@mail.mcgill.ca

*Abstract*—This electronic document is a “live” template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. *\*CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract*. (*Abstract*)

Keywords—component, formatting, style, styling, insert (key words)

# Overview

The design problem was to implement a standard five-stage pipelined 32-bit MIPS processor. The proposed processor is able to implement a subset of 27 instructions of the MIPS instruction set architecture. In addition, the processor implements early branching meaning that branch instructions are resolved at the decoding stage rather than the execution stage. This optimization reduces delays imposed by each branch instruction and thereby reduces the performance loss. We will provide a brief overview of each stage in this section.

The instruction fetch stage mainly consists of the program counter, instruction memory, and an ALU. At the beginning of the program, machine code instructions are loaded into the instruction memory module. The ALU and program counter work in tandem to reference the next instruction address. Since each instruction is 32-bits and MIPS uses byte addressing, the ALU adds four to the program counter value to fetch the next instruction. There is also a 2:1 mux that selects between the ALU output and the decoded branch address. This allows our processor to move to another instruction address if our program must branch or jump.

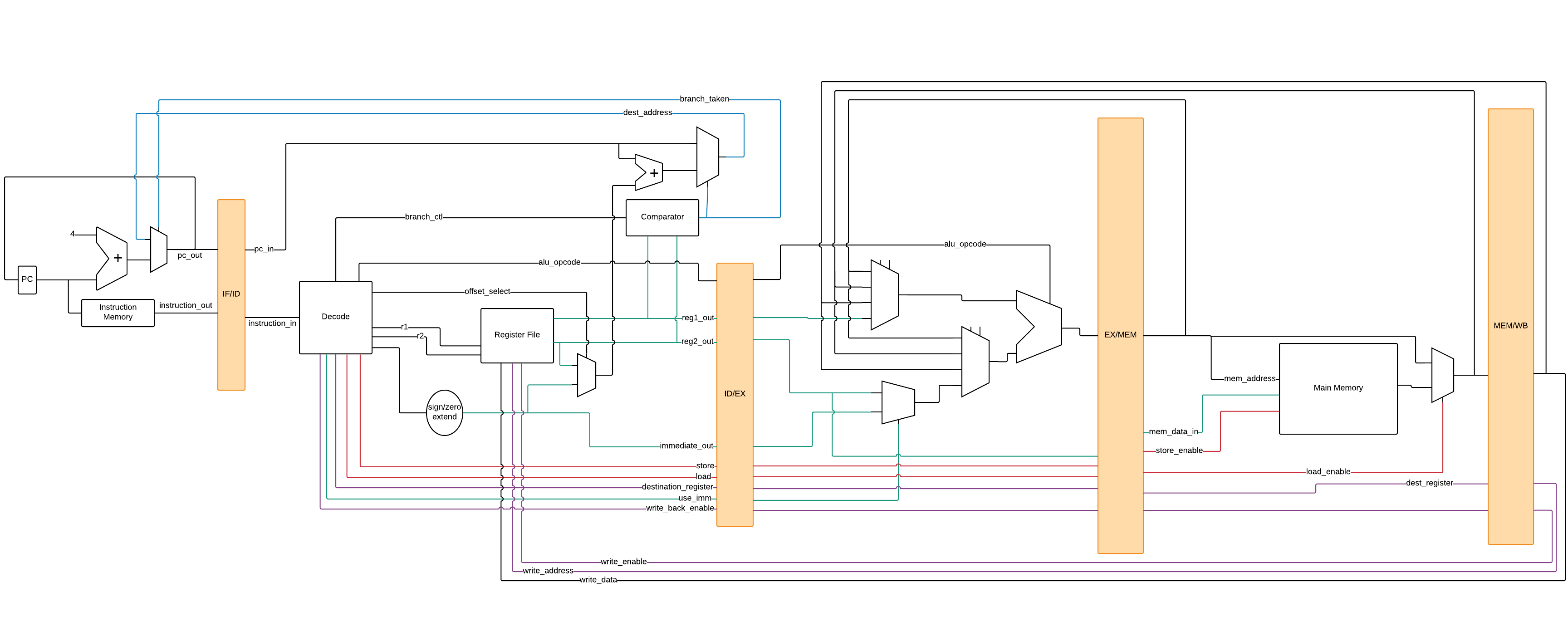
The decode stage mainly consists of our decoder and register file. The decoder interprets the fetched instruction and then outputs the register addresses that the register file will use for referencing and outputting the required data. Since we implemented early branching, branching instructions are resolved in this stage. The decoder outputs the target address of branch instructions and sends it to the comparator, which decides if the program must branch/jump. If the program indeed needs to branch, the target address is added to the current instruction address and sent back to the IF stage for updating the program counter. Finally, there is also a sign/zero extender for sign extending immediate values from 16 bits to 32.

The execute stage mainly consists of a comparator, 4:1 mux’s, a 2:1 mux, and the ALU. The ALU is the most important component of this stage since it performs the arithmetic necessary to execute the instruction. If we are performing a load or store, the ALU adds the offset (forwarded from ID stage) with the register data to create the necessary memory address. If we are performing an add or subtract operation, for example the ALU performs the necessary arithmetic on the data that has been presented to it by the ID stage. The 2:1 mux is used to toggle between feeding the ALU register contents, or an immediate value based on the decoded instruction. It is controlled by the decoder, which outputs a signal to indicate whether the mux should select the register data or the immediate value. Finally, the comparator and 4:1 mux’s implement our data forwarding mechanism. The 4:1 mux’s are connected to the EX, MEM, and WB stage. The comparator checks the register addresses to see if any of the following instruction make references to the same register address that is in the EX, MEM, or WB stages and then sends a control signal to the 4:1 mux’s so that they can select the appropriate register data.

The MEM stage houses our main memory block and either retrieves data on a load instruction or re-writes the referenced memory address on store instructions (in which case, a write enable request has been submitted by the decoder and sent to the memory block). Furthermore, there is a 2:1 mux at the output controlled by the load signal that controls the output of the MEM stage. If we are executing a load instruction, the data from the desired memory location is selected and sent to the register file for write-back. Otherwise, the output of the ALU is selected for writing back to the register file since that data will be the result of executing the instruction.

At the very end of our processor is the WB stage which forwards the resulting data of an instruction to the required location such as the register file and the EX stage (for data forwarding). *Figure 1* [1] shows the high-level organization of our processor. It details the connections and signals within and between each stage.

1. Block diagram of the pipelined processor. Components between the latches (in orange) were organized into module blocks. The top level module manages signals and data that are used by multiple stages.



# Procedure

## Design Approach

As this is a large project with five stages and several components per stage, we took a multi-step approach to design. First, we needed a complete block diagram detailing all the components needed for each stage, as seen in *Figure 1* [1]. During the creation of this block diagram, course material and online resources were reviewed. This was to get an idea of which components would be needed and the type of control logic required for implementing features such as data forwarding, branching, and memory addressing (for load and store). After verifying with the course material and online sources that we had all the necessary components and control signals, we wrote the descriptions of these components stage by stage.

The descriptions of all components seen in the block diagram were written from scratch and implemented using VHDL. As we were a group of four, we could do the component design in parallel. Each component is its own module. Upon completion of the components of a stage, they were integrated together as a single block in a higher-level module. This module contains all the input and output signals of its respective stage and connects the ports of each component to the necessary signals. We should note that the latches dividing each of the five stages were written as modules of their own. They acted sort of like “gates” that acted as data control between stages, storing data values from one stage and presenting them to the next stage on the following clock cycle. These latches are essential for the implementation of a pipeline. Storing the results of each stage in a latch frees up that stage to perform subsequent instructions.

Finally, after successfully integrating the components within each stage, a top-level module was designed for connecting the entire pipeline together. In this top-level module, inter-stage signals such as forwarding signals, branching signals, write enabling, and write back data and signals were implemented and connected to their required ports.

## Testing and Evaluation

The pipelined processor was tested in several stages in ModelSim. The first testing stage was to evaluate each individual component separately to ensure that they behaved as expected. For example, the adder component was tested by forcing its two inputs and its enable signal and verifying that the correct value was outputted. It was important to test each component individually before testing them together because problems could be easily isolated and fixed.

The next testing stage was to evaluate each of the five higher-level pipeline stage modules with a test bench. This was done to verify that given the correct inputs and signals, each pipeline stage would be able to produce the correct outputs. For example, the test bench written for the instruction fetch stage set the branch\_taken variable to force which instruction would be performed next. The program counter and destination address inputs were tracked to check that they only changed when they were expected to, depending on the test case. The pc\_out and instruction\_out outputs were also tracked to ensure that the fetching stage produced the appropriate program count and instruction to be fed into the decoding stage.

Lastly, all five stages were tested together to evaluate the pipelined processor as a whole. Similarly to the individual pipeline stages, the final product was tested using a test bench. A set of MIPs instructions composed of arithmetic, logical, transfer, shift, memory, and control-flow operations was inputted and processed in the pipeline processor. Variable delay cycles were also added between the instructions to provide a more detailed test bench. The results were then written to a .txt file that was used to verify that all the operations and procedures performed as expected.

Even though the entire pipelined processor was tested from each individual component to full stages, there were still multiple problems when testing all five stages together. The two main issues that persisted were in the port mapping and latency between stages. To fix the issues in the port mapping, every associated component was checked and remapped so that all the wiring connected correctly. As for the latency problems, the components in question were reviewed and were found to have design problems that lead to timing errors.

# Optimizations

There are several ways to optimize a pipelined processor including caching, branch prediction, and early branch detection. For our project, we chose to implement early branch detection. This means that branching instructions are detected and resolved at the ID stage rather than at the EX stage of the pipeline. This is possible because as soon as the decoder has finished interpreting a branch instruction, it has extracted the target address from the machine code. Therefore, instead of waiting until the EX stage to update the program counter, we can do this immediately following instruction decode. The implementation of our early branch prediction is outlined in the Overview section. Early branch detection speeds up branching instructions by two clock cycles (as learned from the course material). According to the SPEC92 benchmark mentioned in class, branching instruction constitute 20% of all instructions [2]. Assuming the standard of five clock cycles for a regular instruction to be executed, early branch detection speeds up branching by a factor of 2.5. Using Amdhal’s Law (see Equation 1), this leads to an overall speed up of 1.136. To test this, we wrote a testbench for our pipeline without early branch detection and simulated a test program. After that, we ran the same test program but with our early branch detection implemented. We then observed and compared the delay of our program counter updates. After extensive debugging of our branching mechanism we verified that early branch detection speeds up branch predictions by two clock cycles.

*Speedupoverall =* 1/[(1-F)+(F/S)] (1)

Where F is fraction of instructions and S is factor of speedup.

# Conclusions

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper; use the scroll down window on the left of the MS Word Formatting toolbar.

##### References

[1] D. Lavoie-Boutin, W. Chang, M. Lashari and S. Sheriff, "MIPS 5 Stage Pipeline", GitHub, 2016. [Online]. Available: https://github.com/dlavoieb/ecse-425. [Accessed: 15- Apr- 2018].