ECSE 425 Final Project Report - Pipelined Processor

Kua Chen
ECSE Department
McGill University
Montreal, Canada
kua.chen@mail.mcgill.ca

Yuelin Liu
ECSE Department
McGill University
Montreal, Canada
yuelin.liu@mail.mcgill.ca

Xing Shen
ECSE Department
McGill University
Montreal, Canada
xing.shen@mail.mcgill.ca

Ruoli Wang
ECSE Department
McGill University
Montreal, Canada
ruoli.wang@mail.mcgill.ca

Zeyang Xu
ECSE Department
McGill University
Montreal, Canada
zeyang.xu@mail.mcgill.ca

Junlin Zhuo
ECSE Department
McGill University
Montreal, Canada
junlin.zhuo@mail.mcgill.ca

***Abstract*—This is the report for the final project of ECSE 425 Winter 2022. The main task of this project is designing a 5 staged pipelined processor.**

I. Introduction of the Design

CPU pipelining is a technique that breaks instructions into five stages and allows the stages of different instructions to overlap, thus enabling several instructions to be processed in parallel to increase throughput—— the number of instructions completed per unit of time. With pipelining technology, the execution of a single instruction is not accelerated, because the number of stages for each instruction is not reduced. Different stages of multiple instructions are executed simultaneously by the processor, thus speeding up the instruction flow and shortening the program execution time from an overall perspective.

In this project, we are supposed to design a standard five-stage pipelined 32-bit MIPS processor in VHDL. To implement this design, we follow the list of suggested components. A pipelined processor stores control signals and intermediate results associated with each executing instruction in pipeline registers between stages. For full credit, we also need to implement Hazard Detection and Forwarding. After completing all the required functionalities, some bonus tasks are attempted. We use a 2-bit branch predictor to predict branch targets, which can be one clock faster than before.

II. Functions and Implementations

The five stages of pipelined CPU include fetch (IF), decode (ID), execution (EX), memory access (MEM) and writeback (WB).

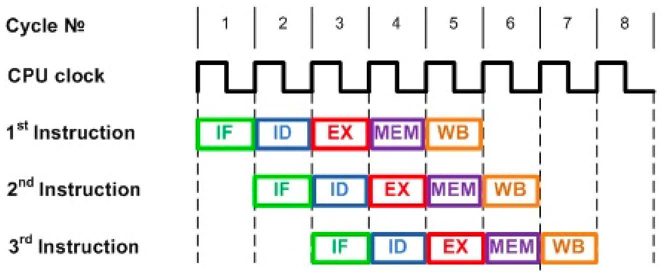


Fig. 1. Pipelined CPU [1]

Instruction Fetch

The main functionality of the IF stage is to fetch the instructions stored in the instruction memory and then pass it one after one to the ID stage in each clock cycle.

In our design, it consists of two parts: the fetch architecture as well as the instruction memory. For the instruction memory, it will load data from program.txt and store inside its memory blocks represented by an array of 32 bit words. The fetch architecture will continuously read instruction from instruction memory by passing the current PC value to the instruction memory. It will also increment PC by 4 for each instruction (instruction 32 bit alignment). Stalls caused by branching and loading from memory are handled by the ID stage as well as the EX stage. When the fetch stage fetches an instruction, it would not know that such instruction will cause stalls or not. The IF stage will receive branch and load flags from ID knowing that a stall is required. It will then insert bubble instructions (ADD \$R0, \$R0, \$R0) to cause a stall. In case of a stall, the PC values will be balanced off to compensate for the increased 4. For branching purposes, it will receive the $branch\_taken$ flag and the branch target from the EX stage. The PC values will be adjusted accordingly by the branch target if the $branch\_taken$ flag is high.

Instruction Decode

The main functionality of the decode stage is interpreting the content in the instruction from the instruction fetch stage, and providing operation code and operands to the execution stage. In addition, the decode stage accesses the register file when retrieving data on registers (operants); and writes data to the register file in the case of handling a write-back event.

The first task of implementing the decode stage is defining components of the instruction and its opcodes. In this project, our instruction opcodes refer to the MIPS reference card [2] and the MIPS opcodes table [3]. The decoder fetches correct registers (operants) according to the decoded opcode and sends them to the output ports of the stage to serve the execution and the ALU. The decode stage does not output the opcode defined in the reference card, it uses a newly constructed operation code instead, and we call it the ALU opcode. The ALU opcode is a 5 bits vector from 00000 to 11010, which corresponds to instructions from ADD to JAL in the project description. In this way, our ALU performs the arithmetic according to the ALU opcode that is provided by this decode stage.

As indicated in our implementation, instructions MFLO and MFHI are handled in the decode stage to reduce extra clock cycles that are incurred by pushing the writing back to the deeper pipeline. An additional signal named as *ex_flag* is used to differentiate between normal instructions that pass to the execution stage and MFLO, MFHI instructions. The execution stage takes different strategies according to the value of the *ex_flag*, in this case, MFLO and MFHI are omitted when *ex_flag* is false.

In addition, various operational flags are provided by the decode stage to serve for hazard detection. For example, the *wb_flag* is set to true when the instruction at the decode stage needs the data to be written back. The *mem_access_flag* shows the current instruction LW or SW requires reading or writing to the memory. The *br_and_j_flag* indicates whether the current instruction involves branching or jumping, together with the *load_flag* that indicates LW instructions, are sent to the instruction fetch stage to insert necessary stalls.

Further on, our decode stage implements data and register forwarding. It sends RS, RT and the register that requires writing back.

Execution

Execution stage mainly takes inputs from the ID stage. It takes 5 bits opcode, two 32 bits operand data, two 2 bit mux selector and some other data from stages that are not ID. It outputs the ALU computation result, PC after update and some other signals for the DM stage.

First, it uses two mux selectors for the forwarding purposes by selecting the source of two operands. Then an ALU component is connected for the computation purposes, thus it can get the actual outputs given the opcode. The decoded opcode is aligned with the order of the Mnemonic table in the instruction from ADD to JAL. Finally, it will update the PC value if the result is a branch jump signal. Moreover, it uses two 32 bit outputs for the decimal arithmetic.

Data Memory

The data memory stage's responsibilities are to manage memory access used by load and store instructions. The memory structure consists of an array of 8192 32 bit word entries, accounting for a total of 32768 bytes. It will initialize itself automatically. For each clock cycle, it checks whether the memory access is required by checking on the *mem_access_flag* passed by the execution stage, and then use the opcode sent to it to conduct specific instructions (load/ store). It then updates (dumps) the entire memory into the file memory.txt

In addition, as part of the forwarding process, it will forward the *wb_flag*, *mem_access_flag* as well as the register that requires writeback that is received from EX to the WB stage.

Write Back

Writeback is the last stage in the pipeline and refers to the process of writing the result of instruction execution back to the general-purpose register set. This stage also supports the forwarding function.

We set the writeback flag. When it's high, we pick the register to write back. If it is low, then automatically set the write register signal to 00000, indicating that there's nothing to write to the register. Assume the writeback flag is high. If the data memory accessed flag is 1, the write-back data is the data loaded from memory. If the data memory accessed flag is 0, then write back the data of ALU's result. Otherwise, if the writeback flag is low, then just send "00000000000000000000000000000000".

Pipeline

We use a file (pipeline.vhd) to connect all the five stages and allow the program to run successfully. The forwarding process is largely implemented here.

In the pipeline file, we connect the relevant ports of different stage components to ensure the "processor" runs successfully. The logic behind it is very simple: each clock cycle, IF will send fetched instruction and PC to ID; ID will send PC, $ALUop$, $sign\_extend$, rs_data, rt_data, writeback register, RS, RT, memory and writeback flags to EX; EX will send writeback and memory flags, memory opcode, data to be written, result from alu, writeback register to DM stage; DM sends memory and writeback flags, readdata, ALU result that is from EX, and writeback register to WB. Such a process inside one clock cycle ensures appropriate signals are passed correctly to each stage to complete the functionality of the processor. Besides the signals that are passed based on the clock cycle, some signals are “hard-mapped” meaning that they are reflected immediately after a value change, to avoid synchronization problems. These signals include: branch flag and load flag from ID to IF, branch_taken flag and target PC from EX to IF, writedata and writeback register from WB to ID, HI and LO registers from EX to ID, and forwarded results of writeback data as well as memory access data send to EX from WB and DM respectively.

The forwarding process is relatively simple. It is achieved by two MUX switches. One controls which value to take for RS in the EX and the other one takes control for RT. The switches check the situations and update their code (00, 10 or 01) based on different situations. EX stage then checks this code to determine for (RS or RT), to use the data from ID (when code = 00) , or data forwarded from DM (when code = 10) or data forwarded from WB (when code = 01). Pipeline.vhd checks the WB flag first to see if a writeback is required, then, in scenarios where writeback is required, it checks whether the RS/RT register sent by ID to EX is the same as the writeback register in DM stage. If so, it sends 10 so that EX will use data forwarded from the DM stage as an updated value of RS/RT, compared to the value from the ID stage. If this is not the case, the pipeline.vhd checks whether the RS/RT register is the same as the writeback register in WB stage. If so, it sends 01 so that EX stage will use the register data forwarded from WB stage. Otherwise, the MUX will be set to 00 to tell EX stage to stick with the RS/RT data sent from the ID stage.

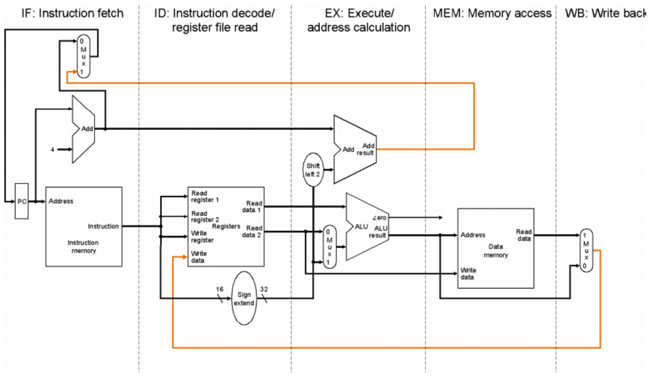


Fig. 2. Pipelined Processor Datapath illustration [4].

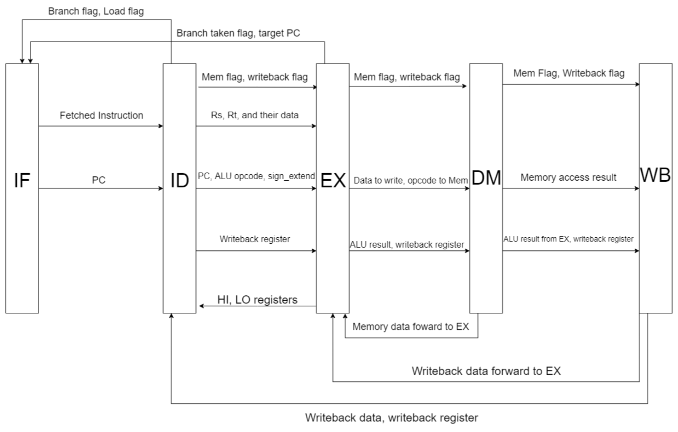


Fig. 3. Signal Passing Scheme and Forwarding.

III. Results and Discussions

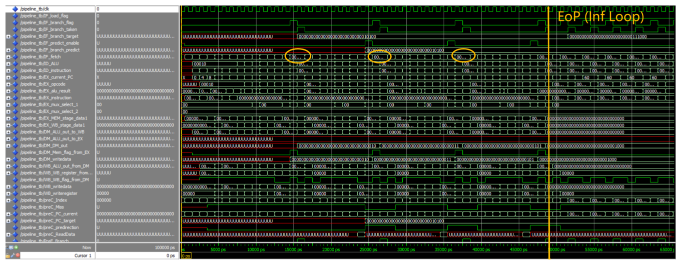
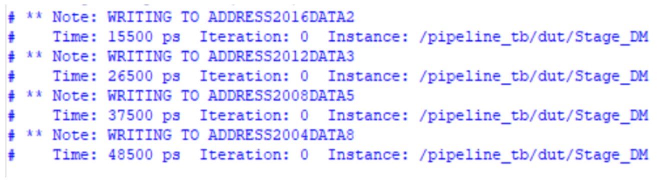


Fig. 4. Simulation Result



```
# ** Note: WRITING TO ADDRESS2016DATA2
#    Time: 15500 ps  Iteration: 0  Instance: /pipeline_tb/dut/Stage_DM
# ** Note: WRITING TO ADDRESS2012DATA3
#    Time: 26500 ps  Iteration: 0  Instance: /pipeline_tb/dut/Stage_DM
# ** Note: WRITING TO ADDRESS2008DATA5
#    Time: 37500 ps  Iteration: 0  Instance: /pipeline_tb/dut/Stage_DM
# ** Note: WRITING TO ADDRESS2004DATA8
#    Time: 48500 ps  Iteration: 0  Instance: /pipeline_tb/dut/Stage_DM
```

Fig. 5. Transcript showing memory operations.

Our simulation result shows that the loop in test 2 has been executed four times. When our program enters an infinite loop at the fourth BNE instruction, the program ends. The yellow circles in Figure 4 are the two bubble instructions loaded during BNE instructions. Note that there is no stall between the instructions except for the BNE instruction.

Since the next instruction uses the register modified by the previous instruction, the program without forwarding needs to wait until the register is written back before calling it. Otherwise an incorrect register value would be read, which would result in runtime errors. However, the simulation result matches the expected results of running the provided bench-mark files, without adding the stall among normal instructions. This explicitly proves our forwarding mechanism is up and working.

IV. Optimization Results

We implemented a 2-bit predictor for the branch prediction. As shown in Figure 7, we designed a branch table like a cache, each row contains a 1bit valid flag, 2-bit state machine, the PC tag, and target PC. The BranchTable.vhd will continuously check every PC value in the IF stage and return the read data to the prediction controller if possible. When the next PC hits the branch table(must be a branch instruction), the controller will set a $Pred\_Enable$ signal to IF and send the target PC, thus IF can use the predicted PC instead of doing stalls. The actual PC in the EX stage will be compared with the target PC after the EX stage. If it is a wrong prediction, we will use a reset signal to reset IF, ID and EX, and update the table. 2 state bits will be updated according to the state diagram in Figure 6.

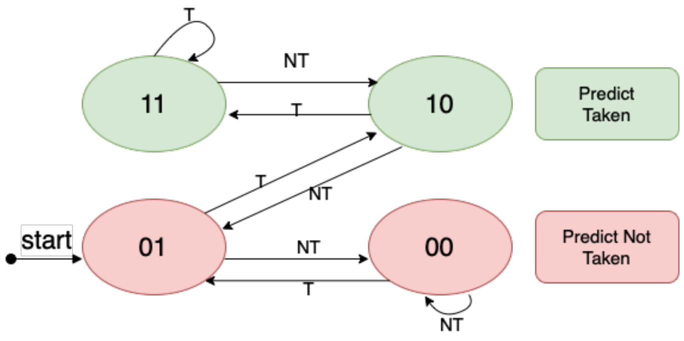


Fig. 6. State Diagram of the 2 bit Prediction.

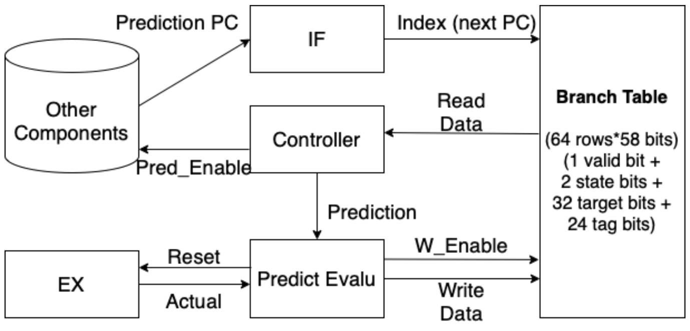


Fig. 7. Prediction Illustration.

Figure 8 shows our result. The blue circle is the normal branch jump signal without branch prediction. Red circles indicate our prediction enable signal along with the predicted PC in yellow signal. After the first blue circle, the predictor updated the model. So in the next same branch instruction, it can provide the predicted PC. It is 2 clock faster than before. In this example(Example 2 as provided by professor), the predicted PC is ..010100 where the entry of the loop exists. It can be much faster from the second loop until the second last loop. The 2-bit predictor has a good performance in parallel loops other than inner loops. If it makes a wrong prediction, it will set the reset signal to disable the previous IF, ID and EX stage. However, we did not implement the reset signal in our project due to the time. We found a feasible solution that uses a reset signal for all stages.[5]

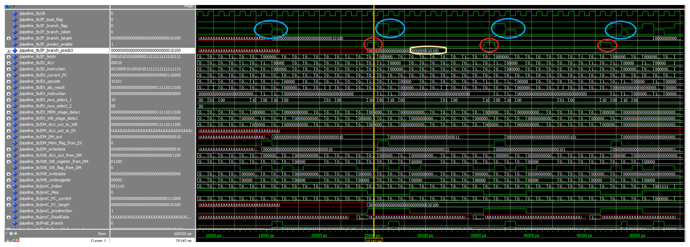


Fig. 8. Prediction Results.

V. Conclusion

Pipelining technology improves the efficiency of the CPU by first standardizing the instruction format. Then, it divides the instructions into five stages with multiple instructions processed in parallel, resulting in a pipeline pattern. During the design, we want to ensure that the execution time of each step is approximately the same as others in the pipeline. In addition, we want to keep the pipeline as full as possible to maximize utilization and throughput while minimizing setup time. Pipelined CPUs usually have problems related to the efficient execution of instructions, which are called hazards. There are mainly three types: structural hazards, data hazards, and control hazards. We solve this problem by adding some stalls in the Decode stage. And, we further optimize the Decode stage by using forwarding techniques to reduce the performance defects resulting from stalls. Finally, we achieve the expected result and successfully pass the provided test.

It is worth mentioning that we try to optimize the existing pipeline processor with a branch target buffer and neural networks for branch prediction in the fetch stage. Given the complexity of neural networks, we finally choose to implement the former approach and achieve a significant speed increase.

VI. References

[1] Open4Tech. Microprocessor Instruction Pipelining. [Online]. Available:
https://open4tech.com/microprocessor-instruction-pipelining
(Accessed 15/04/2022).

[2] EECS Berkeley Edu. MIPS Reference Card. [Online]. Available:
https://inst.eecs.berkeley.edu/ cs61c/resources/MIPS_Green_Sheet.pdf
(Accessed 15/04/2022).

[3] OpenCores. Plasma - most MIPS I(TM) opcodes. [Online]. Available:
https://opencores.org/projects/plasma/opcodes
(Accessed 15/04/2022).

[4] Jeff Brown, UCSD Edu. Designing a Pipelined CPU. [Online]. Available:
https://cseweb.ucsd.edu/classes/su06/cse141/slides/s09-pipeline-1up.pdf
(Accessed 15/04/2022).

[5] ChunXian Wang. Pipelined MIPS Processor with Branch Prediction. [Online]. Available:
https://github.com/chunxianwang/Pipelined-MIPS-Processor-with-Branch-Prediction
(Accessed 15/04/2022).