Assignment #3: Pipelined CPU

CSI3102-02 (Architecture of Computers), Spring 2022

Welcome to the third programming assignment for CSI3102-02 @ Yonsei University for Spring 2022! You are given 10 days to complete this assignment. Please carefully read the instructions below, and make sure you upload your assignment to the LearnUs system (https://learnus.yonsei.ac.kr) until the deadline. The deadline to submit your code is **23:59 KST on June 1st, 2022 (Wed)**.

The Five-Stage Pipelined CPU

In this programming assignment, you will implement a five-stage pipelined CPU capable of executing nine different MIPS instructions in C++. In the previous assignment, you implemented a simplified version of SPIM which is essentially equivalent to a single-cycle CPU. The single-cycle CPU can be modeled by interpreting `MiniMIPS::execInstr` as `SingleCycleCPU::advanceCycle` which simulates one clock cycle of the single-cycle CPU. The single-cycle CPU executes one MIPS instruction per clock cycle, and this is exactly what `MiniMIPS::execInstr` does.

Now, you will exploit your experience implementing the single-cycle CPU for implementing the five-stage pipelined CPU. As discussed during the last few lectures, the pipelined CPU breaks down the execution of a MIPS instruction into five stages: instruction fetch (IF), instruction decode (ID), execute (EX), data memory access (MEM), and writeback (WB). Then, the datapath of the single-cycle CPU gets mapped to the five pipeline stages, four latches get placed between the pipeline stages, the data and control signals get propagated from the ID stage to the later stages through the latches. Accordingly, a MIPS instruction now takes five clock cycles on the pipelined CPU, instead of one clock cycle on the single-cycle CPU.

Your five-stage pipeline CPU should support the following nine MIPS instructions:

- R-type arithmetic & logical instructions: `add`, `sub`, `and`, `or`, `slt`
- I-type arithmetic instruction: `addi`
- I-type data transfer instructions: `lw`, `sw`
- I-type branch instruction: `beq`

You should strictly follow the typing rules of the instructions in this assignment. That is, executing an `add` instruction should treat the two source operands as 32-bit signed integers, not unsigned ones, and should store the result to the destination as a 32-bit signed integer using two's complement format. For logical instructions such as `and`, on the other hand, should treat the input and output register operands as 32-bit bitstring, not as signed/unsigned integers. Note that you are not required to implement the jump (`j`) instruction in this assignment.

During the lectures, we discussed that the five-stage pipeline CPU can suffer from three types of stalls: structural hazard, data hazard, and control hazard. We also discussed that, to ensure the functional correctness despite the existence of the stalls, we can add bubbles between the instructions. The bubbles are essentially no-op instructions which have no impact on the Programmer Visible State (PVS) of the CPU. There can be many implementations for the no-op instructions; however, in this assignment, we will use `add $0, $0, $0` instruction as bubbles. The instruction adds $0 (i.e., 0 + 0 = 0) and stores the result to $0 (i.e., the `$zero` register). Since MIPS ISA discards any writes to $0, executing `add $0, $0, $0` has no effect on the PVS, making it suitable to use as a bubble instruction.

To summarize, the following figure shows the five-stage pipelined CPU you should implement in this assignment. You can refer to the textbook's Figure 4.51 and relevant contents for more information.

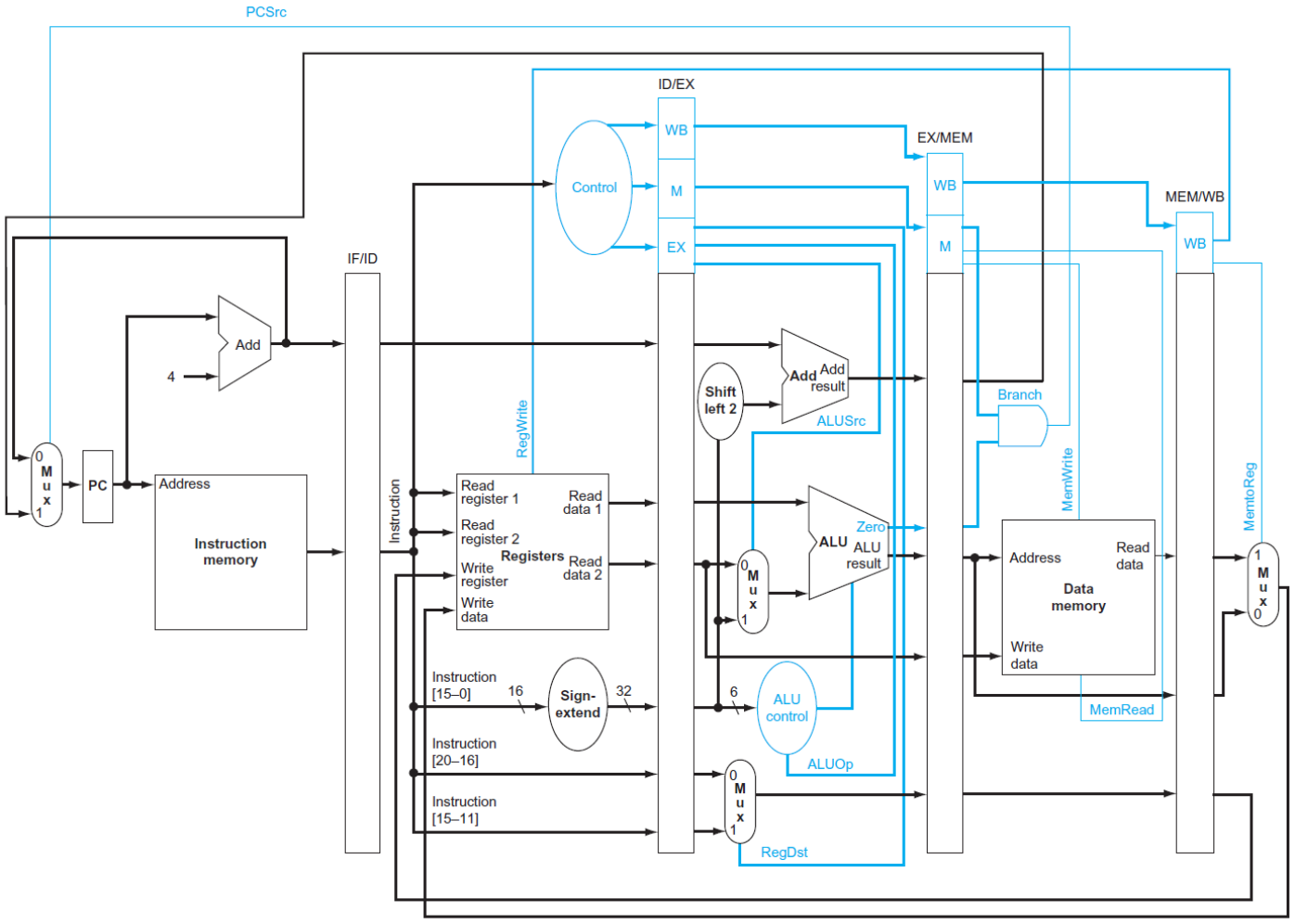

FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Downloading the Assignment

Type the following commands in the terminal to download the assignment.

```
csi3102@csi3102:~$ wget https://hpcp.yonsei.ac.kr/files/csi3102/assn3.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn3.tar.gz
csi3102@csi3102:~$ cd ./assn3
csi3102@csi3102:~/assn3$ ls
data.0.addTwoSums    data.0.addTwoSums.out    data.1.fibonacci    data.1.fibonacci.out
Makefile  MIPS.cc  MIPS.h  testMIPS.cc
```

Among the files, you only need to modify `MIPS.cc` for completing this assignment.

The PipelinedCPU Class

Similar to the MiniMIPS class of the previous assignment, you will utilize the PipelinedCPU class and its associated classes in this assignment.

Open and inspect MIPS.h. The file defines the top-level class named DigitalCircuit which the various other classes for implementing the pipelined CPU inherit. The DigitalCircuit class is an abstract class and defines a single pure virtual method called DigitalCircuit::advanceCycle. As the name implies, the method simulates one clock cycle of the corresponding digital circuit.

Below, you will see many classes implementing different parts of the pipelined CPU including Memory, RegisterFile, and ALU. The Memory class can be used for simulating the instruction memory and the data memory. The RegisterFile class models the 32 general-purpose registers used in the ID stage. The ALU should be used by the EX stage to perform necessary arithmetic/logical operations. To facilitate better understanding on the use of DigitalCircuit::advanceCycle method, we provide complete implementations of Memory::advanceCycle and RegisterFile::advanceCycle methods. The two methods simulate all the operations performed by the instruction memory, the data memory, and the register files within a single clock cycle. We recommend you to analyze and understand how the provided advanceCycle methods are implemented. Note that an implementation of the ALU::advanceCycle method is not provided as it is part of the assignment.

MIPS.h also defines the PipelinedCPU class which serves as the basic building block for your five-stage pipelined CPU implementation. When instantiated, a PipelinedCPU instance spawns two Memory instances, one RegisterFile instance, and an ALU instance as its members. After that, the PipelinedCPU instance will connect the inputs and outputs of the spawned instances to latches and the variables declared to act as wires.

Pay special attention to how the spawned instances' inputs and outputs are connected to the latches and variables of the PipelinedCPU instance. Some inputs and outputs are directly connected to the latches, whereas the others are connected to the wires instead. Such a difference in the input and output connectivities are due to the fact that C/C++ is not very suitable for expressing hardware. In real hardware, different hardware modules concurrently operate; however, the C++ programming model executes all instructions in a sequential manner. This is in fact why we have low-level hardware description languages such as Verilog and VHDL.

Also, pay special attention to how the inputs and outputs of the spawned instances are utilized within the instances' advanceCycle methods. In particular, it is highly recommended to make yourself familiar with the hardware-like C++ programming skills demonstrated in the provided advanceCycle methods.

You can also see that an implementation of the PipelinedCPU::advanceCycle method is already provided. Interestingly, the implementation invokes the five methods of the PipelinedCPU class which correspond to the five pipeline stages being invoked in the reverse order. This is highly related to the aforementioned discrepancies between the C++ programming model and real hardware, and the hardware-like C++ programming skills. Note that you shall not modify the implementation of the provided PipelinedCPU::advanceCycle method. You can fully implement the five-stage pipelined CPU without modifying the method :)

Your Task: The ALU and Five Pipeline Stages

Now, open and inspect `MIPS.cc`. You will see six methods: one method for the `ALU` class and five for the `PipelinedCPU` class. Your task is to implement the six methods with respect to the classes and their methods defined in `MIPS.h`.

First, implement the ALU::advanceCycle method which, upon invocation, (1) reads the values of the `_iInput1`, `_iInput2`, and `_ctrlInput` members, and (2) updates the values of the `_oZero` and `_oResult` members with respect to the provided input values. `_iInput1` and `_iInput2` are the first (i.e., the value of the `rs` register) and second (i.e., either the value of the `rt` register or the sign-extended immediate) inputs to the ALU, respectively, and the _ctrlInput corresponds to the four-bit control signal for the ALU. When implementing the `ALU::advanceCycle` method, refer to the table shown below for performing the operation demanded by the EX stage in each clock cycle. The 'ALU control lines' column of the table corresponds to the `_ctrlInput` member of the `ALU` class.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Then, implement the five methods of the `PipelinedCPU` class which correspond to the five pipeline stages. The names of the members of the `PipelinedCPU` class should be easy to interpret as we tried to follow the names of the corresponding wires and signals of the textbook as closely as possible.

When updating the latches, follow the values specified in the textbook and discussed during the last few lectures. The table below shows all the information you need for properly updating the latches.

Input or output	Signal name	R-format	`lw`	`sw`	`beq`
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE 4.22 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings. (Remember that Op [5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{Op5} \cdot \overline{Op2}$, since this is sufficient to distinguish the R-format instructions from `lw`, `sw`, and `beq`. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

When implementing the `addi` instruction, we need to instruct the ALU to perform an integer addition. For this purpose, use the `ALUOp` signal value of `00` used by the `lw` and `sw` instructions.

We will assume that programmers insert a sufficient number of bubble instructions (i.e., `add $0, $0, $0`) between two instructions which may break the functional correctness of your five-stage pipelined CPU. For this assignment, assume that (1) an update to a destination register of the WB stage is visible to the ID stage within the same clock cycle, and (2) an update to the PC register due to a branch instruction is visible to the IF stage within the same clock cycle. That is, whether a branch instruction is taken gets determined in the first half of a clock cycle of the MEM stage, and the IF stage reads the value of the PC register and fetches the next instruction from the instruction memory in the second half of the same clock cycle.

Examples

We provide two examples, namely `data.0.addTwoSums` and `data.1.fibonacci`. The two examples demonstrate how a correct five-stage pipelined implementation will operate. The examples get executed through `testMIPS.cc` which is used to generate a binary file named `testMIPS`. The `testMIPS` binary takes as input three command-line arguments: the path to a data file, the initial value of the PC register, and the number of clock cycles to simulate. After parsing and initializing the instruction memory, the data memory, and the registers, the `testMIPS` binary will create a `PipelinedCPU` instance and invoke `PipelinedCPU::advanceCycle` method for the specified number of clock cycles. The binary will print the per-cycle status of your pipelined CPU implementation including the value of the PC register, the non-zero values of the 32 registers, instruction memory, and data memory, followed by the values of the four latches located between two adjacent pipeline stages.

As the output for the two examples are too long to show in this document, the reference outputs are included in the assignment: `data.0.addTwoSums.out` and `data.1.fibonacci.out`. You can execute the following commands in the terminal and validate your implementation's outputs:

```
csi3102@csi3102:~/assn3$ make clean && make && ./testMIPS ./data.0.addTwoSums 996 12
...
...  <-- compare these against data.0.addTwoSums.out!
...

csi3102@csi3102:~/assn3$ make clean && make && ./testMIPS ./data.1.fibonacci 1020 20
...
...  <-- compare these against data.1.fibonacci.out!
...
```

Note that the initialPC values for the two examples are provided as 996 and 1020, respectively. When you analyze the examples, the initialPC values seem to be 1000 and 1024; however, given that the IF stage of the five-stage pipelined CPU first updates the PC register and then reads the instruction memory, 996 and 1020 are the correct initialPC values to be used for the examples. Also note that multiple bubble instructions (i.e., `add $0, $0, $0`) are inserted between the meaningful instructions in the example data files to ensure functional correctness. We will discuss how we can reduce the bubbles through additional hardware support; however, for this assignment, we will assume that programmers provide enough bubbles in their code to simplify the assignment.

When validating your implementation against the reference outputs, you should inspect whether your implementation correctly updates not only the PVS, but also the values of the latches in each clock cycle. You can safely assume that your implementation will not correctly update the PVS unless the implementation assigns correct values to the latches in each clock cycle.

Submitting Your Code

After completing your assignment, you should submit your **`MIPS.cc`** to the LearnUs system. The deadline to submit your code is **23:59 KST on June 1st, 2022 (Wed)**. **No late submissions will be accepted**.

In addition, the TAs will not confirm whether you have successfully submitted your code to the LearnUs system. It is your responsibility to check and validate whether your submission is successfully submitted.