Assignment #3: 5-Stage Pipelined CPU

CSI3102-02 (Architecture of Computers), Spring 2023

Welcome to the third programming assignment for CSI3102-02 @ Yonsei University for Spring 2022! You are given 11 days to complete this assignment. Please carefully read the instructions below, and make sure you upload your assignment to the LearnUs system (https://ys.learnus.org) until the deadline. The deadline to submit your code is **23:59 KST on June 11th, 2023 (Sun)**.

The Five-Stage Pipelined CPU

In this programming assignment, you will implement a five-stage pipelined CPU capable of executing ten different MIPS instructions in C++. In the previous assignment, you implemented a single-cycle MIPS CPU which can execute the ten MIPS instructions one instruction per cycle. You will use the datapath and control units for the single-cycle CPU to implement the five-stage pipelined CPU in this assignment.

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). The datapath of the single-cycle CPU gets mapped to the five pipeline stages, four latches get placed between the pipeline stages, and the data and control signals get propagated to the later stages through the latches. With pipelining, executing a MIPS instruction now takes five clock cycles on the pipelined CPU, instead of one clock cycle on the single-cycle CPU.

In this assignment, your single-cycle and five-stage pipelined CPUs should support the following nine MIPS instructions (same as Assignment #2 except the excluded `nor` instruction):

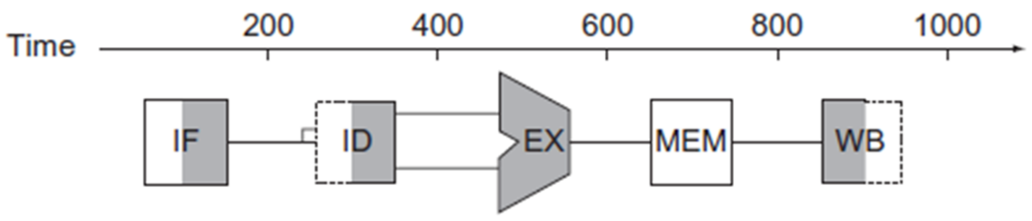
- Integer addition/subtraction (`add`/`sub`/`addi`), logical operations (`and`/`or`), set on less than (`slt`)
- Load/store word (`lw`/`sw`)
- Branch if equal (`beq`)

We already 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.

Although we discussed various hardware-level optimizations to reduce the bubbles (i.e., data forwarding, hazard detection, updating the PC upon a taken branch within the ID stage), you will implement a vanilla five-stage pipelined CPU which does not implement any of the optimizations in this assignment. This implies that your pipelined CPU cannot directly execute some of the assembly code you may have written using the SPIM simulator. The SPIM simulator is essentially a single-cycle CPU and models the impact of the branch delay slots (i.e., the instruction which appears right after a branch instruction always gets executed due to the pipelining). Regarding the hardware components shared between multiple pipeline stages, the vanilla five-stage pipelined CPU avoids the potential interference as follows:

- The PC register gets updated by either the IF stage (for arithmetic/logical instructions and non-taken branches) or the MEM stage (for taken branches) during the first half clock cycle. In the second half clock cycle, the IF stage reads the PC register and accesses the instruction memory.
- The WB stage updates the destination register during the first half clock cycle, and the ID stage reads the source operands/registers in the second half clock cycle.

The following figure demonstrates an example timeline of executing an R-type MIPS instruction on the vanilla five-stage pipelined CPU which implements the above interference-preventing adjustments:



To write a piece of MIPS assembly code which correctly works on the pipelined CPU you implement for this assignment, you should place at least two bubbles (e.g., `add $0, $0, $0`) between two instructions which incur a read-after-write data dependency. By placing at least two bubbles between the two instructions, the prior instruction's WB stage and the later instruction's ID stage get executed within the same clock cycle, allowing the later instruction to correctly read the fresh value which the prior instruction writes to its destination register. Similarly, you should add at least two bubbles after any branch instruction as the PC gets correctly updated during a branch instruction's MEM stage. You need to align the branch instruction's MEM stage and the next instruction's IF stage due to the branch instruction being resolved in the MEM stage, not the ID stage. We provide two test cases which correctly add bubbles between the instructions with respect to the aforementioned limitations. Please have a look at and analyze the test cases to gain better understanding on how to place bubbles between instructions to ensure the functional correctness.

Downloading the Assignment

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

```
csi3102@csi3102:~$ wget https://hpcp.yonsei.ac.kr/files/csi3102/2023sp/assn3-1q2w3e4r0.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn3-1q2w3e4r0.tar.gz
csi3102@csi3102:~$ cd ./assn3
csi3102@csi3102:~/assn3$ ls
CPU.hpp    Makefile    Memory.cpp    Memory.hpp    PipelinedCPU.cpp    PipelinedCPU.hpp    RegisterFile.cpp
RegisterFile.hpp      SingleCycleCPU.cpp      SingleCycleCPU.hpp      testPipelinedCPU.cpp      tests
testSingleCycleCPU.cpp
```

Among the files, you only need to modify `SingleCycleCPU.cpp` and `PipelinedCPU.cpp` for this assignment.

Your 1st Task: Complete the `SingleCycleCPU` Class

Your first task in this assignment is to complete implementing the `SingleCycleCPU` class of Assignment #2. Implementing the five-stage pipelined CPU requires a deep understanding of the single-cycle CPU, so we offer you a second chance to finalize/validate your single-cycle CPU implementation.

For implementing the single-cycle CPU, you should follow the specification for Assignment #2. However, be aware of the following differences between the distributed files for Assignment #2 and this assignment:

- The methods for implementing the datapath of the single-cycle CPU (e.g., the AND gate, adder, multiplexer, ALU) have been moved from the `SingleCycleCPU` class (`SingleCycleCPU.{hpp,cpp}`) to the `CPU` class (`CPU.hpp`). The `CPU` class serves as the base class for both the single-cycle CPU and five-stage pipelined CPU as the CPUs share many hardware components.

- We provide a reference implementation for the datapath-related methods (see `CPU.hpp`). We recommend that you use the provided reference implementation of the methods rather than yours.

- As we provide a reference implementation for the datapath-related methods, your task for implementing the single-cycle CPU is to implement the `SingleCycleCPU::advanceCycle` method. You should now utilize the methods of the `CPU` class, not those of the `SingleCycleCPU` class (e.g., use `CPU::Mux<N>` instead of `SingleCycleCPU::Mux<N>`), for the `SingleCycleCPU::advanceCycle` method.

Your 2nd Task: The Five Stages of the `PipelinedCPU` Class

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. You can also refer to the Assignment #2's specification for more details on how the components of the datapath and control for the five-stage pipelined CPU is implemented in C++ (e.g., which control signals to use for the datapath).

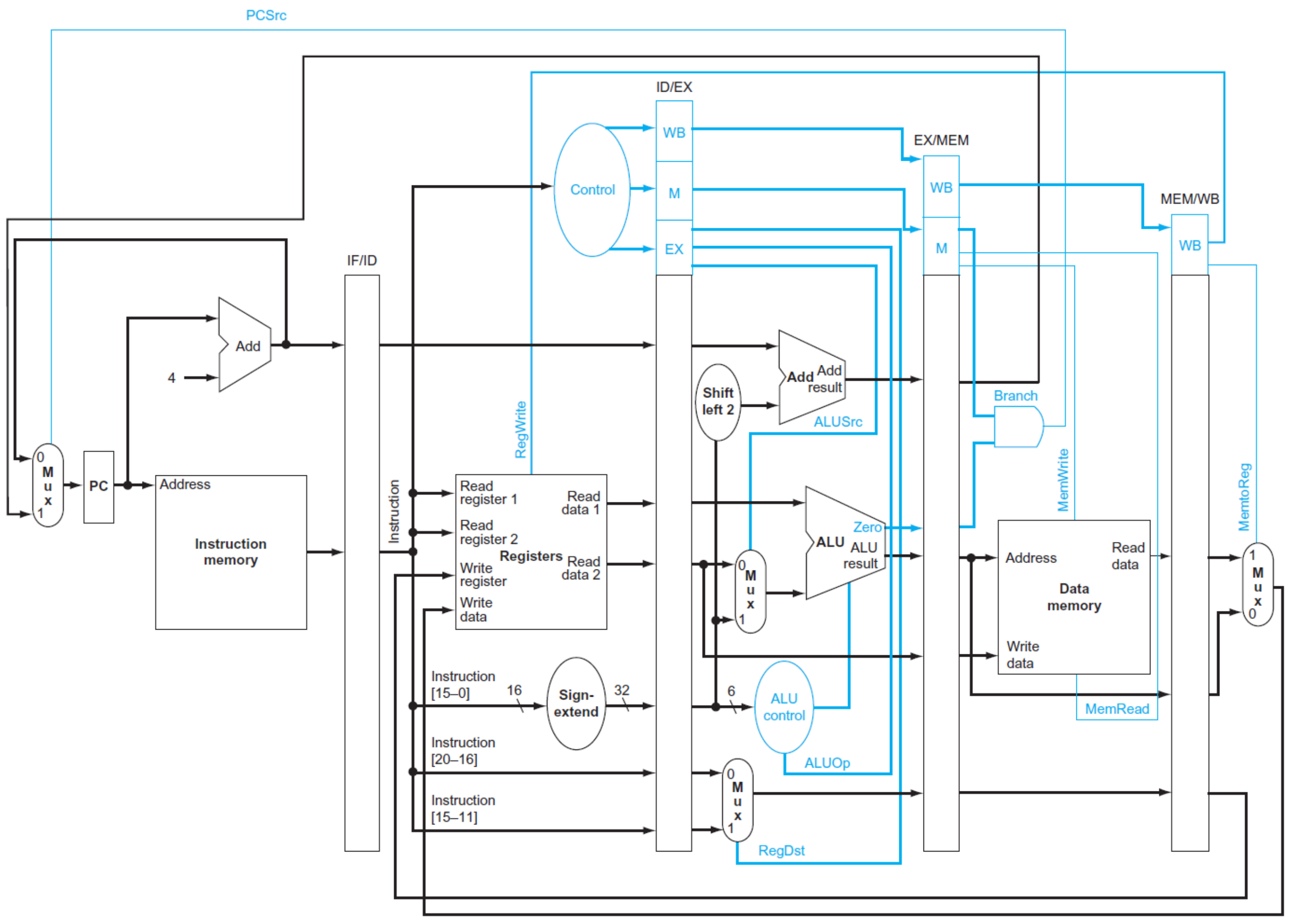


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each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

Your primary task of this assignment is to simulate the per-cycle activities of the five-stage pipelined CPU. For doing so, you should accurately implement each of the five pipeline stages. One key difference between the single-cycle CPU and pipelined CPU implementations is that you are required to implement the five stages of the pipelined CPU. That is, instead of implementing `PipelinedCPU::advanceCycle` method (which is already provided in `PipelinedCPU.hpp`), you should implement the following methods of the PipelinedCPU class which correspond to the five stages of the five-stage pipelined CPU: `PipelinedCPU::InstructionFetch`, `PipelinedCPU::InstructionDecode`, `PipelinedCPU::Execute`, `PipelinedCPU::MemoryAccess`, and `PipelinedCPU::WriteBack`. The five methods should be implemented within the provided `PipelinedCPU.cpp`. In each of these methods, you should utilize appropriate hardware components of the datapath available through the methods of the `CPU` class.

You will notice a few things when analyzing the `PipelinedCPU.hpp`:

- The `PipelinedCPU::advanceCycle` method invokes the five pipeline stages in the reverse order. That is, the WB stage is invoked first and the IF stage is invoked last. Although such an invocation of the pipeline stages may seem weird, invoking the pipeline stages in the reverse order can accurately model the behavior of the actual five-stage pipelined CPU. Think of this in this way: for each of the four latches, the data should be consumed by the next pipeline stage before the previous pipeline stage writes new data to the latch.

- The latches which get placed between the consecutive pipeline stages are defined as `struct`s. The four latches of interest are: `PipelinedCPU::m_latch_{IF_ID,ID_EX,EX_MEM,MEM_WB}`. As the four latches of the five-stage pipelined CPU consist of different pieces of data, the four latches utilize different `struct` definitions. For example, the IF-ID latch (i.e., `PipelinedCPU::m_latch_IF_ID`) is defined to store 1) the 32-bit (PC+4) value, and 2) the 32-bit instruction loaded from the instruction memory. The detailed description of the latches and their contents are provided in `PipelinedCPU.hpp`. Given the latches, your implementation of the five pipeline stages should consume the input data from the source latch, perform necessary operations with respect to the five pipeline stages, and store new pieces of information to the destination latch.

- You will notice that the constructor for the `PipelinedCPU` class (i.e., `PipelinedCPU::PipelinedCPU`) takes as input the following parameters: `enable{DataForwarding,HazardDetection,IDBranch}`. You can safely ignore the parameters for this assignment and just implement the vanilla five-stage pipelined CPU. Extending the vanilla pipelined CPU to enable such hardware-level optimizations will be your next assignment.

Examples

We provide two examples in the `tests` directory. Similar to the two tests of the previous assignment, the first example (i.e., `ex1_{regFile,instMemFile,dataMemFile}`) consists of sequential arithmetic and logical instructions, whereas the second example (i.e., `ex2_{regFile,instMemFile,dataMemFile}`) includes both taken and not-taken branch instructions.

A key difference between the examples of the prior and current assignments is the addition of bubble instructions (i.e., `add $0, $0, $0`). For example, in the first example of this assignment (i.e., `ex1_instMemFile`), two bubble instructions are inserted between the second and third instructions (i.e., two `add $0, $0, $0` between `lw $t2, 4($t0)` and `add $t3, $t1, $t2`). As mentioned earlier, the pipelined CPU you implement in this assignment implements neither data forwarding nor hazard detection.

Therefore, the two bubble instructions are needed to align the ID stage of the third instruction with the WB stage of the second instruction. By doing so, the read-after-write dependency between the second and third instructions gets resolved; the second instruction updates its destination register in the first half clock cycle, and the third instruction reads its source registers in the second half clock cycle. Similarly, the second example adds two bubble instructions after a branch instruction as it gets resolved (i.e., whether the branch gets taken gets identified) in the first half clock cycle of the MEM stage. By placing two bubble instructions after a branch instruction, the next valid instruction will either get executed (in case of a not-taken branch) or the CPU will fetch the instruction located at the branch target address (in case of a taken branch) in the second half clock cycle of the IF stage of the correct next instruction.

When you analyze `testPipelinedCPU.cpp`, you will notice that the provided PC value gets subtracted by four before instantiating a `PipelinedCPU` object. Since a correct implementation of the pipelined CPU always updates its PC value in the first half of any clock cycle (e.g., update PC to PC+4), subtracting the provided PC by four before instantiating the object is necessary.

After you complete implementing the single-cycle and five-stage pipelined CPUs, you can test your implementation with the two examples as follows:

```
## To test your single-cycle CPU implementation:

csi3102@csi3102:~/assn3$ make clean && make testSingleCycleCPU
csi3102@csi3102:~/assn3$ ./testSingleCycleCPU ./tests/ex1_regFile \
                                 ./tests/ex1_instMemFile ./tests/ex1_dataMemFile 16
...
...  <-- Compare these against tests/ex1_SingleCycleCPU.out!
...
csi3102@csi3102:~/assn3$ ./testSingleCycleCPU ./tests/ex2_regFile \
                                 ./tests/ex2_instMemFile ./tests/ex2_dataMemFile 20
...
...  <-- Compare these against tests/ex2_SingleCycleCPU.out!
...

### To test your pipelined CPU implementation:

csi3102@csi3102:~/assn3$ make clean && make testPipelinedCPU
csi3102@csi3102:~/assn3$ ./testPipelinedCPU 0 ./tests/ex1_regFile \
                                 ./tests/ex1_instMemFile ./tests/ex1_dataMemFile 16
...
...  <-- Compare these against tests/ex1_PipelinedCPU.out!
...
csi3102@csi3102:~/assn3$ ./testPipelinedCPU 0 ./tests/ex2_regFile \
                                 ./tests/ex2_instMemFile ./tests/ex2_dataMemFile 20
...
...  <-- Compare these against tests/ex2_PipelinedCPU.out!
...
```

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

You should compress **only your `SingleCycleCPU.cpp` and `PipelinedCPU.cpp`**, and upload the compressed file to the LearnUs system. The deadline to submit your code is **11:59pm KST on June 11th, 2023 (Sun)**. **No late submissions will be accepted**.

One way to create a compressed file is:

```
csi3102@csi3102:~/assn3$ tar -czvf assn3-submit.tar.gz SingleCycleCPU.cpp PipelinedCPU.cpp
```

After creating the compressed file (i.e., `assn3-submit.tar.gz`), upload it to the LearnUs system.