Assignment #4: Improving the Pipelined CPU

CSI3102-02 (Architecture of Computers), Spring 2023

Welcome to the fourth (and final) programming assignment for CSI3102-02 @ Yonsei University for Spring 2023! You are given 2.5 weeks 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 25th, 2023 (Sun)**.

Improving The Five-Stage Pipelined CPU

In this programming assignment, you will implement two hardware-level improvements to the vanilla five-stage pipelined CPU you wrote for Assignment #3. The two improvements are **data forwarding** and **hazard detection**. First, data forwarding allows the data stored in the EX-MEM and MEM-WB latches to be used as an input to the Arithmetic-Logical Unit (ALU) in the EX stage. This allows the CPU to execute two data-dependent instructions back-to-back without the need of placing bubbles between the two instructions. Second, hazard detection further improves the programmability by automatically inserting a bubble cycle upon a load-use data hazard. If the underlying CPU supports hazard detection, programmers and compilers no longer need to insert a mandatory bubble instruction between a prior load-word instruction and a later data-dependent instruction.

Your improved five-stage pipeline CPU should support the following nine MIPS instructions. Note that the vanilla five-stage pipelined CPU you wrote for Assignment #3 also targets the same nine MIPS instructions.

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

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/assn4-0xDEADBEEF.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn4-0xDEADBEEF.tar.gz
csi3102@csi3102:~$ cd ./assn4
csi3102@csi3102:~/assn4$ ls
CPU.hpp    Memory.cpp  PipelinedCPU.cpp  RegisterFile.cpp  testPipelinedCPU.cpp  Makefile  Memory.hpp
PipelinedCPU.hpp  RegisterFile.hpp  tests
```

Among the files, you should modify and submit only the `PipelinedCPU.cpp` for this assignment.

Your 1st Task: Complete the `PipelinedCPU` Class (40%)

Your first task in this assignment is to complete the vanilla five-stage pipelined CPU for Assignment #3 by modifying the five methods in `PipelinedCPU.cpp` (`PipelinedCPU::{InstructionFetch,InstructionDecode,Execute,MemoryAccess,WriteBack}`) which correspond to the five pipeline stages. Refer to the specification for Assignment #3 for details. Even if you only implement the vanilla five-stage pipelined CPU, you will still get some credit for this assignment. 40% of the total score for this assignment will be given for implementing/completing the vanilla five-stage pipelined CPU.

You may reuse your code for Assignment #3 to complete the first task. Despite the additions in `PipelinedCPU.hpp` to facilitate data forwarding and hazard detection, your code for Assignment #3 should seamlessly work for the files distributed for this assignment.

Your 2nd Task: Implement Data Forwarding (30%)

Your next task is to implement data forwarding on top of your vanilla five-stage pipelined CPU. To implement data forwarding, you should not only extend your implementation for the five pipeline stages, but also implement the data forwarding unit.

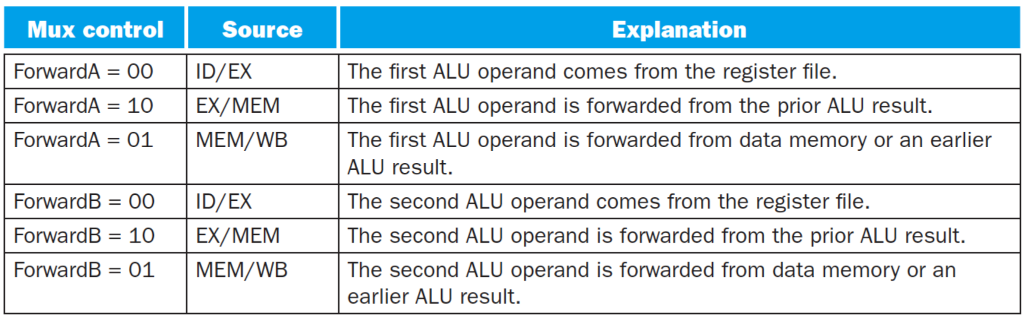
The data forwarding unit decides the two inputs for the ALU in the EX stage by placing two 3-to-1 multiplexers in front of the ALU inputs. In this assignment, the data forwarding unit is defined as a method of the `PipelinedCPU` class, namely `PipelinedCPU::ForwardingUnit`. The method's inputs are as follows:

- `ID_EX_rs` and `ID_EX_rt`: The values of the `rs` and `rt` registers currently stored in the ID-EX latch. These two values are the original inputs to the ALU on the vanilla five-stage pipelined CPU. When no data forwarding occurs, the two values should be used as the inputs to the ALU.

- `EX_MEM_regWrite` and `EX_MEM_rd`: The `RegWrite` control signal and the index of the destination register (`rd`) for the instruction whose metadata is currently stored in the EX-MEM latch. If the `RegWrite` control signal is set (i.e., the signal's value is 1), the instruction which performs its MEM stage in the current clock cycle writes some value to its destination register. In such a case, the index of the instruction's destination register is passed through the `EX_MEM_rd` parameter.

- `MEM_WB_regWrite` and `MEM_WB_rd`: The RegWrite control signal and the index of the destination register (`rd`) for the instruction whose metadata is currently stored in the MEM-WB latch. The meanings of the parameters are the same as EX_MEM_regWrite and `EX_MEM_rd` except that the two values are from the MEM-WB latch instead of the EX-MEM latch.

Using the inputs, the data forwarding unit should produce two control signals: `forwardA` and `forwardB`. The correct values for the two control signals are shown in the following table:



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 4.55 The control values for the forwarding multiplexors in Figure 4.54. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

With the data forwarding unit, you should extend your implementation of the vanilla five-stage pipelined CPU to 1) identify whether data forwarding should occur using the `PipelinedCPU::ForwardingUnit` method, and 2) feed correct input data to the ALU depending on the occurrence of data forwarding. The figure below shows how the two control signals produced by the data forwarding unit gets utilized for data forwarding:

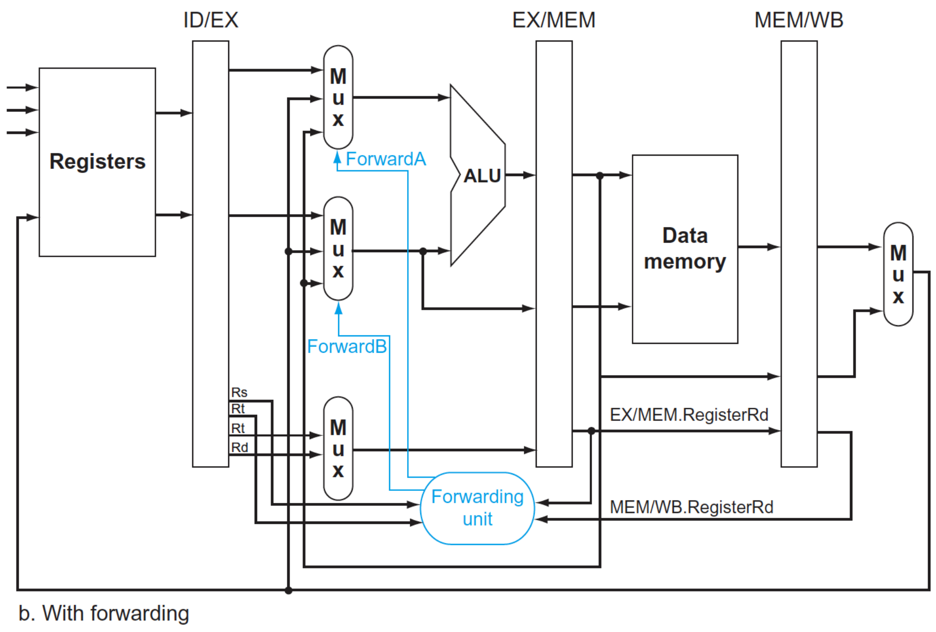


FIGURE 4.54 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware. Note that the ID/EX.RegisterRt field is shown twice, once to connect to the Mux and once to the forwarding unit, but it is a single signal. As in the earlier discussion, this ignores forwarding of a store value to a store instruction. Also note that this mechanism works for `slt` instructions as well.

Note that you may use the `PipelinedCPU::Mux<N>` method provided in the `PipelinedCPU.hpp` as the three-to-one multiplexers placed in front of the ALU's inputs. Please refer to the method's code for identifying how it emulates a three-to-one multiplexer. A close-up of the ALU's inputs in the presence of the data forwarding support is shown below:

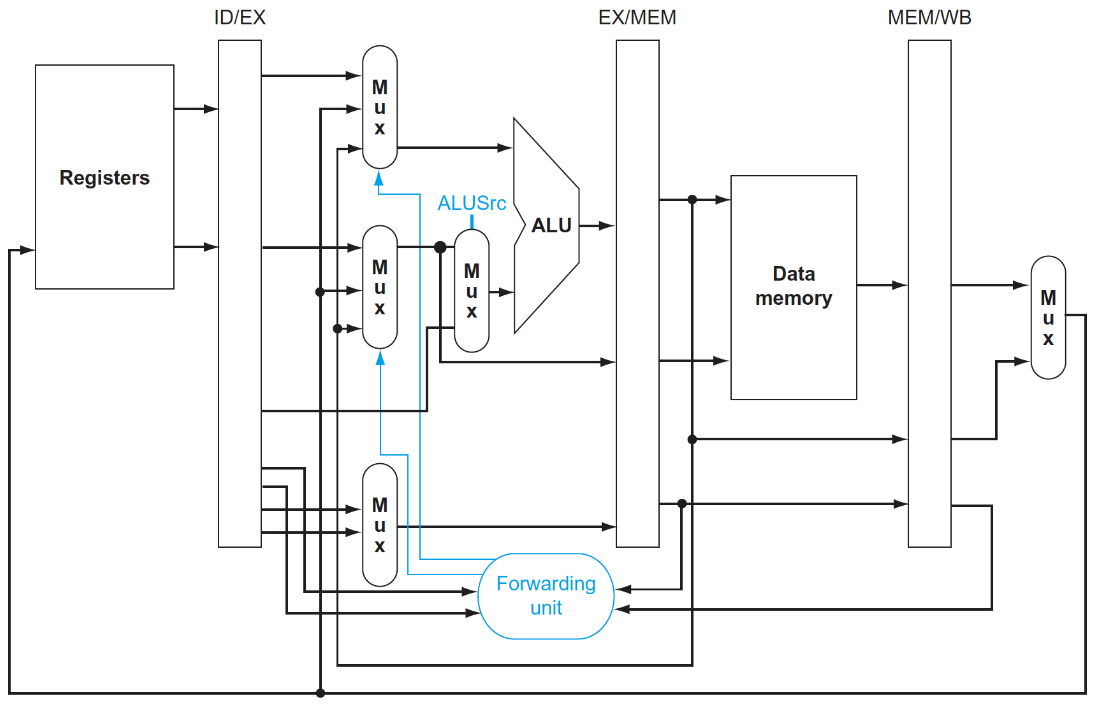


FIGURE 4.57 A close-up of the datapath in Figure 4.54 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

To utilize the data forwarding unit, you will need to transfer some data from the MEM and WB stages to the data forwarding unit and the three-to-one multiplexers placed in front of the ALU's inputs. For transferring the data required for the data forwarding unit, you can utilize the following attributes of the PipelinedCPU class: `m_MEM_to_FwdUnit_{regWrite,rd,rdValue}` for passing the `RegWrite` signal, the index of the destination register, and the value to be written to the destination register for the instruction stored in the EX-MEM latch, and `m_WB_to_FwdUnit_{regWrite,rd,rdValue}` which store the same data for the instruction stored in the MEM-WB latch.

Implementing the data forwarding support will grant you 30% of the total score of this assignment.

Your 3rd Task: Implement Hazard Detection (30%)

Your final task in this assignment is to implement hazard detection. Hazard detection allows the CPU to automatically detect a load-use data dependency and insert a bubble between a load-word instruction and the subsequent instruction which depends on the load-word instruction. Note that implementing data forwarding and implementing hazard detection are orthogonal; the extensions to the vanilla five-stage pipelined CPU for data forwarding and hazard detection do not necessarily depend on each other, so you can implement hazard detection without data forwarding.

Similar to data forwarding, you will need to extend not only the five pipeline stage implementations, but also implement the hazard detection unit. The hazard detection unit is defined within the `PipelinedCPU` class as the `PipelinedCPU::HazardDetectionUnit` method, and its inputs are as follows:

- `IF_ID_rs` and `IF_ID_rt`: The (speculative) indices of the `rs` and `rt` registers of the 32-bit instruction stored in the IF-ID latch. The indices are speculative since we do not know whether the instruction is an R-type instruction yet; if the instruction is an I-type or a J-type instruction, the `rt` field will contain the index of the instruction's destination register instead. In this assignment, however, we will conservatively assume that both the `rs` and `rt` fields contain the source register indices of the instruction by speculatively assuming that the instruction stored in the IF-ID latch is R-type.

- `ID_EX_memRead` and `ID_EX_rt`: The `MemRead` control signal and the destination register index (`rt`) of the instruction currently stored in the ID-EX latch. If the `MemRead` control signal is set (i.e., its value is set to 1), it means that the instruction stored in the ID-EX latch is a load-word instruction. In such a case, the `rt` will be the destination register of the load-word instruction.

Using the inputs, the `PipelinedCPU::HazardDetectionUnit` method should produce the following three control signals:

- `PCWrite` indicates whether to allow the PC register to be updated in this clock cycle. If `PCWrite` is set to 1, the CPU will allow the PC register to be updated to a new value (e.g., PC+4). Otherwise, the CPU will discard any updates to the PC register and keep the PC register's value as-is.

- `IFIDWrite` indicates whether the CPU will allow the contents of the IF-ID latch to be updated in this clock cycle. Similar to PCWrite, setting `IFIDWrite` to 1 indicates that the IF stage is allowed to update the IF-ID latch in this clock cycle. Otherwise, no updates to the IF-ID latch should be allowed.

- `ctrlSelect` serves as the selector for the two-to-one multiplexer placed in front of the ID-EX latch's control signal fields. When a load-use data dependency is detected by the CPU, the CPU should insert a bubble between the prior load-word instruction and the subsequent data-dependent instruction. Inserting the bubble is achieved by setting all the control signals of the ID-EX latch as 0.

After implementing the `PipelinedCPU::HazardDetectionUnit` method, you should extend your five pipeline stage implementations to accurately support hazard detection. The figure below shows how the internal structure of the CPU changes according to the hazard detection and data forwarding support:

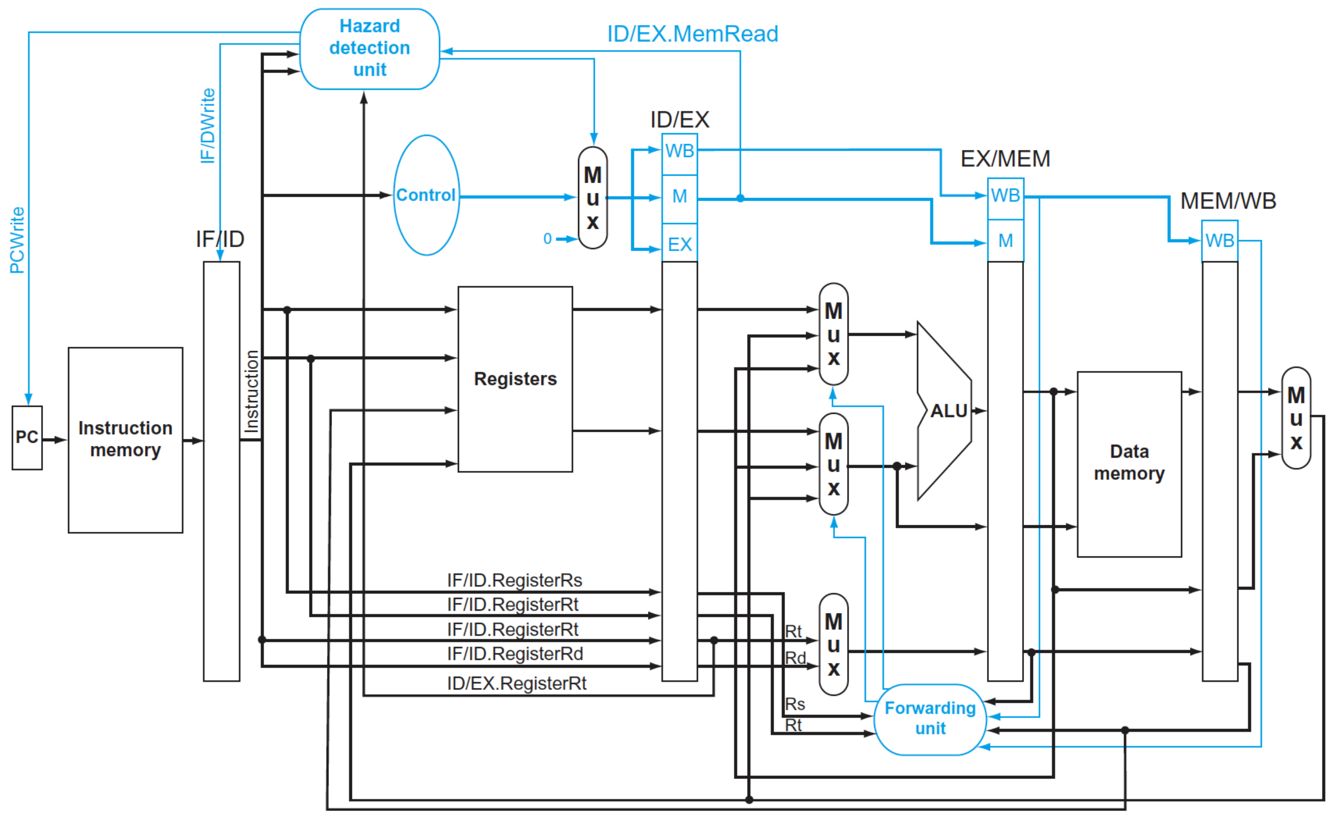


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Similar to the data forwarding support, implementing hazard detection requires passing some data from the ID-EX latch to the hazard detection unit. In addition, you will need to pass some of the outputs of the hazard detection unit to the IF stage. For these purposes, the `PipelinedCPU` class provides the following additional attributes you can utilize for passing the data: `m_EX_to_HazDetUnit_{memRead,rt}` for passing the data from the ID-EX latch to the hazard detection unit, `m_HazDetUnit_to_IF_{PCWrite,IFIDWrite}` for passing the `PCWrite` and `IFIDWrite` control signals to the IF stage, and `m_MEM_to_IF_{PCSrc,branchTarget}` for the IF stage to correctly update the PC register with respect to the `PCWrite` control signal.

Implementing the hazard detection support will account for 30% of the total score of this assignment.

Examples

We provide four examples in the `tests` directory. 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. You can refer to the specification for Assignment #3 on how to examine your vanilla five-stage pipelined CPU implementation with the two examples.

The third example can be used to test whether your data forwarding support operates correctly, and the fourth example illustrates how the CPU operates when both the data forwarding and hazard detection supports are properly implemented. You can test your data forwarding and hazard detection implementations by invoking the commands shown below:

```
## To test your implementation for data forwarding:
csi3102@csi3102:~/assn4$ make clean && make testPipelinedCPU
csi3102@csi3102:~/assn4$ ./testPipelinedCPU 0 ./tests/ex3_regFile \
                         ./tests/ex3_instMemFile ./tests/ex3_dataMemFile 15 1 0
...
...  <-- compare these against tests/ex3_PipelinedCPU_DataForwarding.out
...

## To test your implementation for data forwarding & hazard detection:
csi3102@csi3102:~/assn4$ ./testPipelinedCPU 4096 ./tests/ex4_regFile \
                         ./tests/ex4_instMemFile ./tests/ex4_dataMemFile 12 1 1
...
...  <-- compare these against tests/ex4_PipelinedCPU_DataForwarding_HazDet.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 submit **only your `PipelinedCPU.cpp`** to the LearnUs system. The deadline to submit your code is **11:59pm KST on June 25th, 2023 (Sun)**. **No late submissions will be accepted**.

Since you only need to submit only a single file for this assignment, we recommend you to not compress your **`PipelinedCPU.cpp`** when submitting it to the LearnUs system. Simply uploading your `PipelinedCPU.cpp` to the LearnUs system as-is is sufficient.