Assignment #3:
A Five-Stage Pipelined CPU

CAS3102-01 (Architecture of Computers), Spring 2025

Welcome to the third programming assignment for CAS3102-01!

In this programming assignment, you will implement a five-stage pipelined CPU capable of executing the ten MIPS instructions that you already dealt with in Assignment #2. Specifically, you will implement three versions of the pipelined CPU: vanilla, data-forwarding-enabled, and hazard-detection-enabled versions.

Similar to Assignment #2, your five-stage pipelined CPU should support the following ten MIPS instructions:

- 32-bit signed integer addition/subtraction (`add`, `sub`, `addi`)
- Bitwise logical operations (`and`, `or`, `nor`)
- 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.

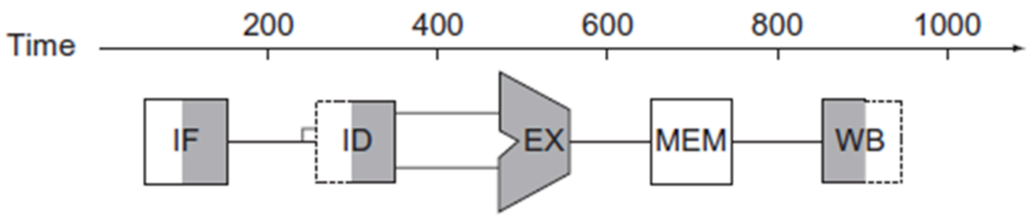
```
csi3102@csi3102:~$ wget https://hpcp.yonsei.ac.kr/cas3102-2025sp/assn3-20250519-91505202.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn3-20250519-91505202.tar.gz
csi3102@csi3102:~$ cd ./assn3
csi3102@csi3102:~/assn3$ ls -p
ALUControl.h  ALU.h  Control.h  DigitalCircuit.h  Makefile  Memory.h  Miscellaneous.h  PipelinedCPU.h
RegisterFile.h  testAssn3.cc  tests/
```

In this assignment, you only need to modify two files (i.e., `Miscellaneous.h` and `PipelinedCPU.h`) if you have fully completed Assignment #2. If you did not fully complete Assignment #2, you must modify more files (e.g., `Control.h`, `RegisterFile.h`) to implement the pipelined CPU in this assignment.

Q1: The Vanilla Five-Stage Pipelined CPU

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 are 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 question, you will implement a “vanilla” five-stage pipelined CPU that does not support any of the hardware-level optimizations for reducing bubble cycles (i.e., the virtual no-op instruction, which adds a sufficient number of delay cycles between the two MIPS instructions, incurring data and control hazards). The following figure illustrates how the vanilla pipelined CPU executes an R-type instruction:



You will notice that the IF, ID, and WB stages do not take a full clock cycle. Theoretically, the IF and the ID stages should occur in the second half of a clock cycle, and the WB stage should happen in the first half of the clock cycle. The remaining two stages, the EX and the MEM stages, occupy an entire clock cycle; however, for the MEM stage of a `beq` instruction, a potential update to the PC register by the `beq` instruction (i.e., when a `beq` instruction gets taken and the PC register should be updated to the target address of the `beq` instruction) should complete within the first half of a clock cycle.

Such an unaligned execution of the IF, ID, and WB stages (also the MEM stage for a `beq` instruction) allows the vanilla pipelined CPU to resolve potential hardware-level conflicts on two hardware-level components:

- The MEM stage of a `beq` instruction (potentially) writes its target address to the PC register in the first half of a clock cycle, allowing the IF stage of a later instruction to read the value from the PC register in the second half of the clock cycle.

- The WB stage of a previous instruction writes a new value to its destination register in the first half of a clock cycle, allowing the ID stage of a later instruction to read the new value from its source register in the second half of the clock cycle.

Despite avoiding the hardware-level conflicts on the PC register and the Register File, your vanilla pipeline will not be able to resolve the data hazard (e.g., read-after-write data dependency) and the control hazard. Therefore, when testing your vanilla five-stage pipelined CPU, you should make sure that a sufficient number of bubble instructions/cycles (e.g., `add $0, $0, $0`) exist between the hazard-incurring MIPS instruction pairs. For example, if a `beq` instruction takes its branch (i.e., `$rs` and `$rt` match), there should be at least two bubble instructions right after the `beq` instruction. As another example, if two back-to-back instructions exhibit a data hazard (e.g., `add $t2, $t0, $t1` followed by `sub $t4, $t2, $t3` exhibit a read-after-write data dependency on `$t2`), there should be at least two bubble instructions placed between the two instructions so that the WB stage of the preceding instruction aligns with the ID stage of the later instruction. We provide two test cases that correctly add bubbles between the instructions with respect to the aforementioned limitations. Please have a look at and analyze the test cases to gain a better understanding of how to place bubbles between instructions to ensure functional correctness.

In summary, the following figure illustrates the vanilla five-stage pipelined CPU, which supports neither data forwarding nor hazard detection. The figure can also be found in your textbook in Figure 4.51.

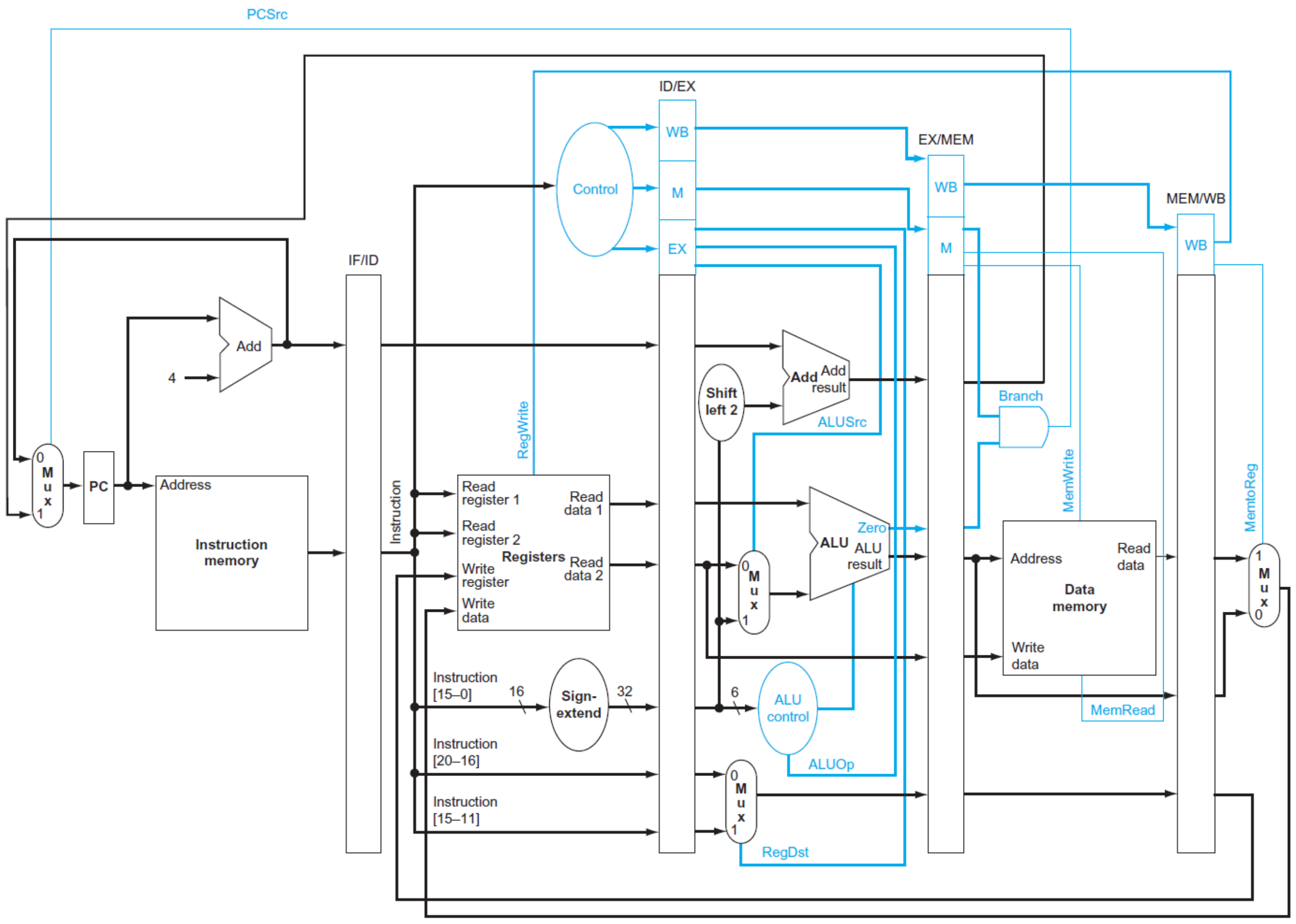


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each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

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 that 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 the `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` does not affect the PVS, making it suitable to use as a bubble instruction.

When implementing the vanilla five-stage pipelined CPU, you should utilize the digital circuits you wrote for Assignment #2 (e.g., the Control unit, the ALU Control unit). In addition, you should utilize the four digital circuits defined in the `Miscellaneous.h` file. The four digital circuits are: a signed N-bit adder, the Sign-extend unit, the two-to-one multiplexer, and the three-to-one multiplexer. The signed N-bit adder and the Sign-extend units are for calculating PC+4 and a `beq` instruction's target address. The two-to-one multiplexer is the same as the typical MUX you used until now, and the three-to-one multiplexer is for implementing the data forwarding mechanism, which is not required for this problem but for the next one.

The primary file you should work on is the `PipelinedCPU.h` file. It defines the `PipelinedCPU` class whose `advanceCycle` method simulates the per-cycle behavior of the vanilla five-stage pipelined CPU.

Compared to the `SingleCycleCPU` class you used in Assignment #2, you will notice that the `PipelinedCPU` class defines four latches using `struct`s. The four latches of interest are: `PipelinedCPU::_latch{IFID, IDEX, EXMEM, MEMWB}`. 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::_latchIFID`) is defined to store the 32-bit (PC+4) value and the 32-bit instruction loaded from the instruction memory. The detailed description of the latches and their contents is provided in `PipelinedCPU.h`. Given the latches, your implementation of the vanilla pipelined CPU should emulate the five pipeline stages by consuming the input data from the source latch, performing necessary operations with respect to the pipeline stages, and storing new pieces of information in the destination latch.

Examples for Q1

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 (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 vanilla pipelined CPU 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. 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 `testAssn3.cc`, 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 vanilla five-stage pipelined CPU, you can test your implementation with the two examples as follows:

```
### To test your vanilla pipelined CPU implementation:

csi3102@csi3102:~/assn3$ make clean && make testAssn3V1
csi3102@csi3102:~/assn3$ ./testAssn3V1 0 ./tests/ex1_regFile ./tests/ex1_instMemFile \
                         ./tests/ex1_dataMemFile 16
...
...  <-- Compare these against tests/ex1_Assn3V1.out!
...
csi3102@csi3102:~/assn3$ ./testAssn3V1 0 ./tests/ex2_regFile ./tests/ex2_instMemFile
                         ./tests/ex2_dataMemFile 20
...
...  <-- Compare these against tests/ex2_Assn3V1.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.

Q2: Data Forwarding

Your next task is to implement data forwarding on top of your vanilla five-stage pipelined CPU. You will notice that the `PipelinedCPU.h` file has some code wrapped with the `ENABLE_DATA_FORWARDING` macro. Only when this macro is defined through the command-line argument of the `g++` compiler, the code wrapped with the macro take effect. For example, you will see the `-DENABLE_DATA_FORWARDING` as a command-line argument for producing the `testAssn3V2` binary file in the `Makefile` file. The macro will only be enabled when evaluating your implementation of data forwarding on top of the vanilla pipelined CPU.

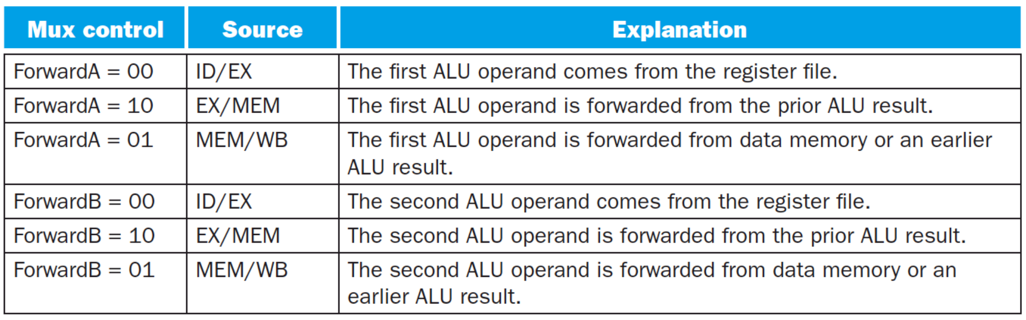
To implement data forwarding, you should not only extend the `advanceCycle` method of your vanilla five-stage pipelined CPU implementation, but also implement the Forwarding unit defined as the `ForwardingUnit` class in the `PipelinedCPU.h` file. The forwarding unit decides the two inputs for the ALU in the EX stage by placing two 3-to-1 MUXes in front of the ALU inputs. The data forwarding unit's inputs are:

- `IDEXrs` and `IDEXrt`: 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.

- `EXMEMRegWrite` and `EXMEMRegDstIdx`: 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 that 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 `EXMEMRegDstIdx` parameter.

- `MEMWBRegWrite` and `MEMWBRegDstIdx`: 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 `EXMEMRegWrite` and `EXMEMRegDstIdx`, 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.

Using the Forwarding unit, you should extend your implementation of the vanilla five-stage pipelined CPU to 1) identify whether data forwarding should occur using the Forwarding unit, and 2) feed the 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 forwarding unit are utilized for data forwarding:

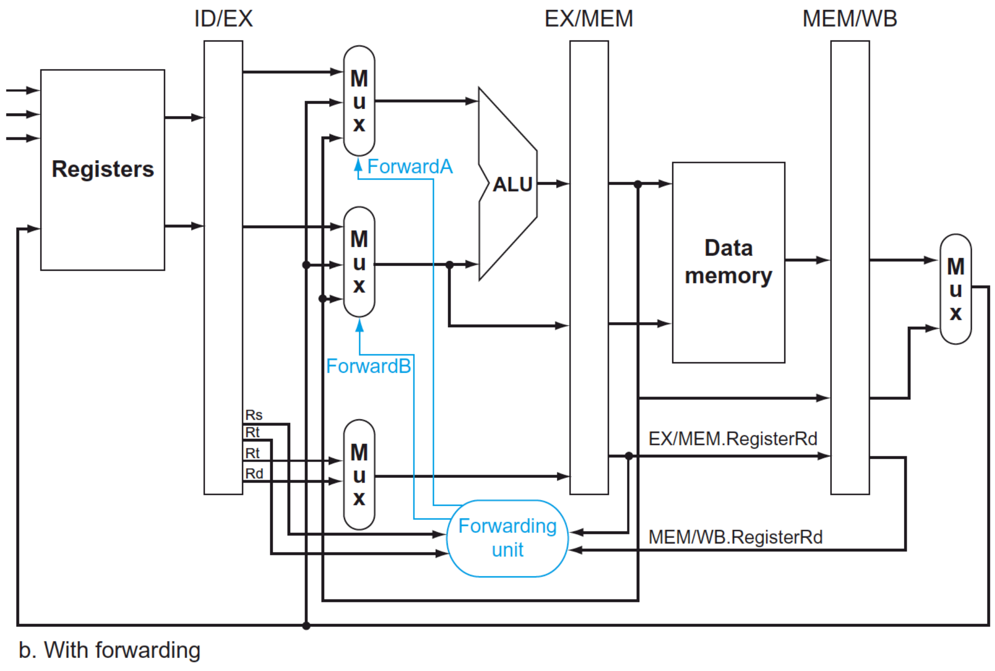


FIGURE 4.54 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from 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.

A close-up of the ALU's inputs in the presence of the data forwarding support is shown below:

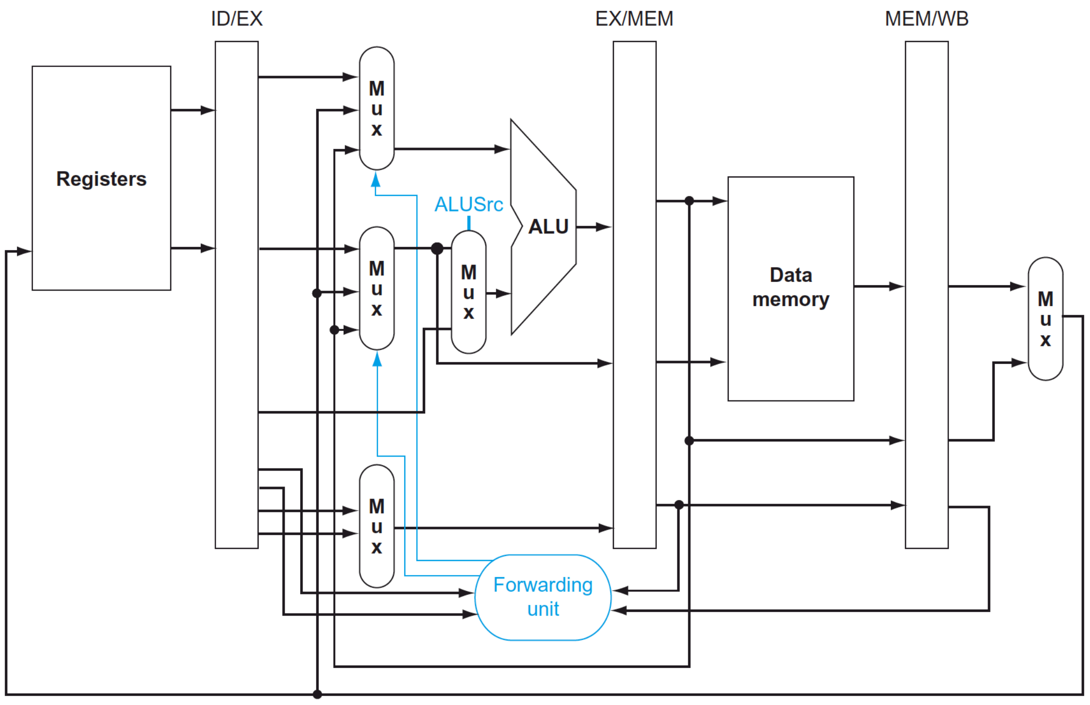


FIGURE 4.57 A close-up of the datapath in Figure 4.54 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

To utilize the Forwarding unit, you will need to transfer some data from the MEM and WB stages to the forwarding unit and the three-to-one multiplexers placed in front of the ALU's inputs. All the necessary inputs except one (i.e., the index of the `rs` register from the ID-EX latch) are already available from the vanilla pipelined CPU implementation, so the `PipelinedCPU` class only extends its ID/EX latch to include the index of the rs register for enabling data forwarding (i.e., `PipelinedCPU::_latchIDEX::rs`).

Example for Q2

After you complete implementing data forwarding on top of the vanilla five-stage pipelined CPU, you can test your implementation with one example as follows:

```
### To test your data-forwarding-enabled pipelined CPU implementation:

csi3102@csi3102:~/assn3$ make clean && make testAssn3V2
csi3102@csi3102:~/assn3$ ./testAssn3V2 0 ./tests/ex3_regFile ./tests/ex3_instMemFile \
                         ./tests/ex3_dataMemFile 15
...
...  <-- Compare these against tests/ex3_Assn3V2.out!
...
```

Q3: Hazard Detection

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 that depends on the load-word instruction. In this assignment, we assume that hazard detection can be enabled only on top of data forwarding; if data forwarding is disabled, so is hazard detection.

Similar to data forwarding, you will need to implement one extra digital circuit, namely the hazard detection unit. The hazard detection unit is defined as the `HazardDetectionUnit` class within the `PipelinedCPU.h` file, and its inputs are as follows:

- `IFIDRs` and `IFIDRt`: 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.

- `IDEXMemRead` and `IDEXRt`: 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 hazard detection unit 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 are allowed.

- `IDEXCtrlWrite` 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 hazard detection unit, 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:

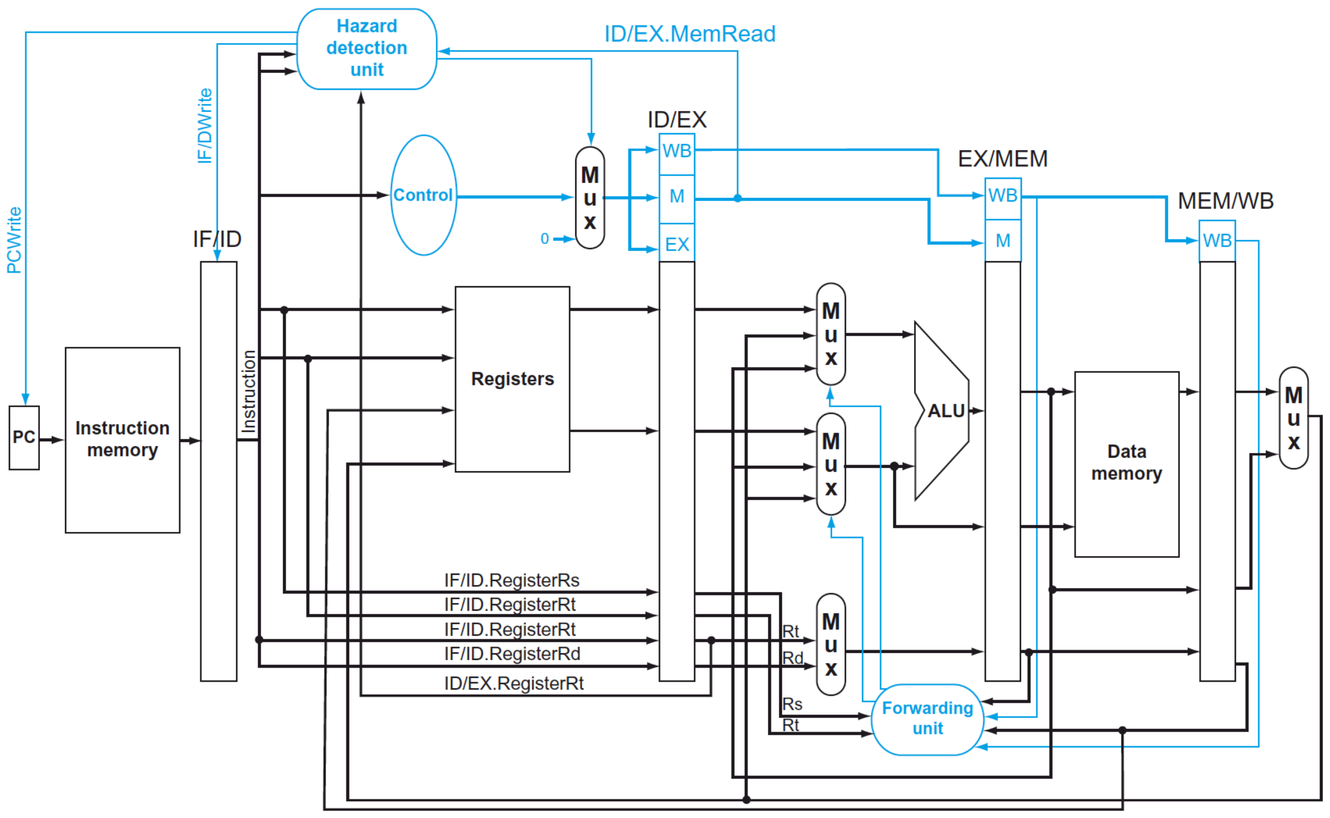


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware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

Similar to the data forwarding support, implementing hazard detection requires passing some data from the ID-EX latch to the hazard detection unit. This can be achieved by leveraging additional wires from the IF-ID and ID-EX latches to the hazard detection unit. The updates to the PC register and the IF-ID latch should be properly controlled by the pipelined CPU's `advanceCycle` method according to the hazard detection unit's outputs. The same applies to the control signal part of the ID-EX latch; the control signal part should be set to all 0s if the `IDEXCtrlWrite` is set to 0, not 1.

Example for Q3

After you complete implementing hazard detection on top of the data-forwarding-enabled five-stage pipelined CPU, you can test your implementation with one example as follows:

```
### To test your hazard-detection-enabled pipelined CPU implementation:

csi3102@csi3102:~/assn3$ make clean && make testAssn3V3
csi3102@csi3102:~/assn3$ ./testAssn3V3 4096 ./tests/ex4_regFile ./tests/ex4_instMemFile \
                          ./tests/ex4_dataMemFile 12
...
...  <-- Compare these against tests/ex4_Assn3V3.out!
...
```

Submitting Your Code

You should upload **only the six `.h` files you modified** to LearnUs (i.e., `ALU.h`, `ALUControl.h`, `Control.h`, `Memory.h`, `PipelinedCPU.h`, and `RegisterFile.h`). If you solved only a part of the assignment, then upload only the `.h` files you modified to LearnUs.

The deadline for uploading the .h files to the LearnUs system is **11:59 pm on June 1, 2025 (Sun)**. If you need more time, you may upload the .h files to the LearnUs system by **11:59 pm on June 3, 2025 (Tue)**; however, uploading the file after June 1 will result in **a 50% reduction in your score**. For example, if your file gets 80 points, your score will be (1) 80 points if you uploaded your file by June 1, or (2) 40 points if you uploaded your file by June 3.

Please be careful that if your file results in a compilation error, you will receive no score (i.e., 0 points). Therefore, please double-check that your file successfully gets compiled before submitting it.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< END OF ASSIGNMENT #3 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>