**NYU Tandon School of Engineering**

**Fall 2023, ECE 6913**

**Homework Assignment 5**

---

*Instructor: Azeez Bhavnagarwala*, email: ajb20@nyu.edu

*Course Assistant Office Hour Schedule*

On Zoom: 9:30AM – 11AM Monday, Tuesday, Wednesday & Thursday(Focus on Project A)

1. Arushi Arora, aa10350@nyu.edu

2. Prashanth Rebala, pr2359@nyu.edu

3. Moin Khan, mk8793@nyu.edu

4. Raj Ghodasara, g4357@nyu.edu

**Homework Assignment 3** [released Monday November 6th 2022] [due Wednesday November 15th by 11:59PM]

You *are allowed* to discuss HW assignments with anyone. You are *not allowed* to share your solutions with other colleagues in the class. Please feel free to reach out to the Course Assistants or the Instructor during office hours or by appointment if you need any help with the HW.
Please enter your responses in this Word document after you download it from NYU Classes. *Please use the Brightspace portal to upload your completed HW.*

---

**1.** Consider a version of the pipeline from *Section 4.5 in RISC-V text* that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). Suppose that (after optimization) a typical n- instruction program requires an additional **0.**4*n NOP instructions to correctly handle data hazards.

**1.1** Suppose that the cycle time of this pipeline without forwarding is 250 ps. Suppose also that adding forwarding hardware will reduce the number of NOPs from .4*n to .05*n, but increase the cycle time to 300 ps. What is the speedup of this new pipeline compared to the one without forwarding?

T1 = 1.4n × 250ps

T2 = 1.05n × 300 ps

speedup = T1/T2 = [1.4 × 250] / [1.05 × 300] = 1.11

**1.2** Different programs will require different amounts of NOPs. How many NOPs (as a percentage of code instructions) can remain in the typical program before that program runs slower on the pipeline with forwarding?

T0 = 1.4n × 250ps

TF = (1 + NOP) × n × 300ps

$1.4n \times 250ps \geq (1 + NOP) \times n \times 300ps$

$NOP \leq 0.167$

**1.3** Repeat 1.2; however, this time let x represent the number of `NOP` instructions relative to `n`. (In 1.2, x was equal to 0.4) Your answer will be with respect to x.

$(1+x)\ n \times 250ps \geq (1 + NOP) \times n \times 300ps$

$NOP < (250x - 50)/300$

**1.4** Can a program with only `.075*n NOPs` possibly run faster on the pipeline with forwarding? Explain why or why not.

No. In the optimal scenario, where forwarding incurs no NOPs, the execution time will be 300n, exceeding 250 times 1.075n. Utilizing forwarding is more prone to enhance the performance of programs characterized by minimal data hazards, especially when the associated cycle time overheads with forwarding hardware are kept to a minimum.

**1.5** At minimum, how many NOPs (as a percentage of code instructions) must a program have before it can possibly run faster on the pipeline with forwarding?

The greater the count of NOPs per instruction, denoted as x in the absence of forwarding, the more favorable it becomes for forwarding to operate and reduce the number of NOPs per instruction. As the count of NOPs per instruction without forwarding (x) decreases, the advantages of forwarding are confined to offsetting the cycle time overheads associated with the use of forwarding hardware. When the cycle time overheads balance out the benefits of forwarding (NOP = 0), no further enhancements in performance are achievable. Consequently, a smaller x, corresponding to the case of NOP=0 in the equation NOP < (250x – 50)/300 mentioned earlier, does not facilitate performance improvements beyond a certain point.

**2.** Consider the fragment of RISC-V assembly below:

```
sd x29, 12(x16)
ld x29, 8(x16)
sub x17, x15, x14
beqz x17, label
add x15, x11, x14
sub x15, x30, x14
```

Suppose we modify the pipeline so that it has only one memory (*that handles both instructions and data*). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.

**2.1** Draw a pipeline diagram to show were the code above will stall.

IF ID EX ME WB

IF ID EX ME WB

IF ID EX ME WB

** ** IF ID EX ME WB

IF ID EX ME WB

IF ID EX ME WB

**2.2** In general, is it possible to reduce the number of stalls/NOPs resulting from this structural hazard by reordering code?

The contention for memory access persists whenever a load or store instruction is in play, creating a conflict between the initial stage of the RISC pipeline, which occurs three cycles earlier, and the fourth stage of the pipeline for the load or store instruction. Reducing NOPs resulting from structural hazards caused by load or store instructions cannot be achieved by

rearranging the code three cycles later than the load or store instruction. This limitation arises because every such instruction, occurring three cycles after a load or store instruction, necessitates memory access during the instruction fetch stage.

**2.3** Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding `NOPs` to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.
Certainly, the removal of this structural hazard is possible by inserting a NOP three instructions downstream for each load/store instruction. However, this approach comes with a significant performance penalty, as load/store instructions usually constitute 30%-40% of the instructions in a program. Adding a NOP for each load/store instruction can result in a substantial increase in execution time, ranging from 30%-40%. An alternative and more efficient solution to address this structural hazard in hardware, without incurring the substantial overhead of introducing NOPs, is to incorporate separate memory arrays for data and instructions.

**2.4** Approximately how many stalls would you expect this structural hazard to generate in a typical program? (*Use the instruction mix shown below*)

| R-type/I-type (non-ld) | ld | sd | beq |
|---|---|---|---|
| 52% | 25% | 11% | 12% |

In the program depicted in the table above, all of the load/store instructions contribute to 36% of the instructions experiencing stalls.

**3.** If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the `ALU`. (See Problem 4 in HW 4) As a result, the `MEM` and `EX` stages can be overlapped and the pipeline has only four stages.

**3.1** How will the reduction in pipeline depth affect the cycle time?
The cycle time is determined by the latency of the slowest among the five RISC pipeline stages, and it is not influenced by the quantity of stages. Reducing the number of pipeline stages or altering the pipeline depth of the processor will not impact the cycle time.

**3.2** How might this change improve the performance of the pipeline?
Overlapping the MEM with the EX pipeline stages, facilitated by a reduction in the number of pipeline stages, results in a one-cycle improvement in the overall latency of an instruction. However, the presence of load-use data hazards, which typically requires the insertion of a NOP cycle between a load instruction and an instruction utilizing the data read from memory, can potentially eliminate the necessity for this NOP cycle. This is because the MEM stage executes in Clock Cycle 3 of the load instruction, enabling pipeline registers to forward data read from the MEM stage to the ALU input multiplexer. Consequently, there is a potential reduction in the Execution Time penalty in NOPs per instruction for load-use data hazards.

**3.3** How might this change degrade the performance of the pipeline?
Eliminating the offset from load/store instructions may necessitate the use or pairing of "addi" instructions with load/store instructions, thereby leading to an increase in the total number of instructions in the program.

**4.** Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

*Choice 1:*

```
ld x11, 0(x12): IF ID EX ME WB
add x13, x11, x14: IF ID EX..ME WB
or x15, x16, x17: IF ID..EX ME WB
```

*Choice 2:*

```
ld x11, 0(x12): IF ID EX ME WB
add x13, x11, x14: IF ID..EX ME WB
or x15, x16, x17: IF..ID EX ME WB
```

In the initial sequence, the data in register x11 becomes accessible only at the conclusion of CC4 but is employed at the beginning of CC4 in the execute stage of the add instruction, resulting in a 'load-use-data-hazard.' Inserting a stall in CC5 is untimely. In the subsequent sequence, the content of register x11, available at the end of CC4, is utilized at the commencement of CC5 (as opposed to CC4 as described earlier) due to the delay in the EX stage caused by the insertion of a stall in CC4.

**5.** Consider the following loop.

```
LOOP: ld   x10, 0(x13)
      ld   x11, 8(x13)
      add x12, x10, x11
      subi x13, x13, 16
      bnez x12, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the `EX` (as opposed to the `ID`) stage.

**5.1** Show a pipeline execution diagram for the first two iterations of this loop.

`Please see below in response to 5.2`

**5.2** Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? (Begin with the cycle during which the `subi` is in the `IF` stage. End with the cycle during which the `bnez` is in the `IF` stage.)
[1] Due to the utilization of flawless branch prediction, there are no cycle losses attributable to branch hazards; in other words, the branch is consistently predicted and taken accurately.
[2] We presume complete forwarding support, mitigating the impact of data hazards that can be resolved with forwarding and preventing pipeline stalls.
[3] Load-use hazards are highlighted in red boxes below and are resolved by introducing stalls in the pipeline.
[4a] Stages within the pipeline that remain unused by any instruction are marked in blue.
[4b] Any clock cycle lacking the utilization of all pipeline stages is denoted with 'N.'

**6.** This exercise is intended to help you understand the cost/complexity/performance trade-offs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from *Figure 4.53 in RISC-V text (reproduced below)*. These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions has a particular type of RAW data dependence.

| 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. |

The type of `RAW` data dependence is identified by the stage that produces the result (`EX` or `MEM`) and the next instruction that consumes the result (1st instruction that follows the one that produces the result, 2nd instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so "`EX` to 3rd" and "`MEM` to 3rd" dependences are not counted because they cannot result in data hazards. We also assume that branches are resolved in the `EX` stage (as opposed to the `ID` stage), and that the CPI of the processor is 1 if there are no data hazards.

| EX to 1st Only | MEM to 1st Only | EX to 2nd Only | MEM to 2nd Only | EX to 1st and EX to 2nd |
|---|---|---|---|---|
| 5% | 20% | 5% | 10% | 10% |

Assume the following latencies for individual pipeline stages. For the `EX` stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

| IF | ID | EX (no FW) | EX (full FW) | EX (FW from EX/MEM only) | EX (FW from MEM/WB only) | MEM | WB |
|---|---|---|---|---|---|---|---|
| 120 ps | 100 ps | 110 ps | 130 ps | 120 ps | 120 ps | 120 ps | 100 ps |

**6.1** For each RAW dependency listed above, give a sequence of at least three assembly statements that exhibits that dependency.

1) EX to 1st only:

   sub x2, x7, x8

   add x10, x2, x9

   add x27, x28, x29

2) MEM to 1st only:

   ld x2, 0(x7)

add x8, x2, x9

sub x15, x22, x23

3) EX to 2nd only:

add x2, x7, x8

sub x15, x22, x23

add x22, x2, x23

4) MEM to 2nd only:

ld x2, 0(x7)

add x5, x6, x7

add x8, x2, x9

5) EX to 1st and EX to 2nd :

sub x2, x7, x8

add x10, x2, x9

add x27, x2, x29

**6.2** For each RAW dependency above, how many NOPs would need to be inserted to allow your code from ***6.1*** to run correctly on a pipeline with no forwarding or hazard detection? Show where the NOPs could be inserted.

1) EX to 1st only:

sub x2, x7, x8

NOP

NOP

add x10, x2, x9

add x27, x28, x29

2) MEM to 1st only:

ld x2, 0(x7)

NOP

NOP

add x8, x2, x9

sub x15, x22, x23

3) EX to 2nd only:

add x2, x7, x8

sub x15, x22, x23

NOP

add x22, x2, x23

4) MEM to 2nd only:

ld x2, 0(x7)

add x5, x6, x7

NOP

add x8, x2, x9

5) EX to 1st and EX to 2nd :

sub x2, x7, x8

NOP

NOP

add x10, x2, x9

add x27, x2, x29

**6.3** Analyzing each instruction independently will over-count the number of NOPs needed to run a program on a pipeline with no forwarding or hazard detection. Write a sequence of three assembly instructions so that, when you consider each instruction in the sequence independently, the sum of the stalls is larger than the number of stalls the sequence actually needs to avoid data hazards.

**6.4** Assuming no other hazards, what is the CPI for the program described by the table above when run on a pipeline with no forwarding? What percent of cycles are stalls? (For simplicity, assume that all necessary cases are listed above and can be treated independently.)

| EX to 1st Only | MEM to 1st Only | EX to 2nd Only | MEM to 2nd Only | EX to 1st and EX to 2nd |
|---|---|---|---|---|
| 5% | 20% | 5% | 10% | 10% |

0.05*2 + 0.2*2 + 0.05*1 + 0.1*1 + 0.1*2 = 0.85 NOPs per instruction

CPI = 1.85

0.85/1.85 cycles = 46%, are NOPs

**6.5** What is the CPI if we use full forwarding (forward all results that can be forwarded)? What percent of cycles are stalls?

The sole RAW dependency that forwarding cannot address is load-use-data hazards. In this scenario, 20% of instructions, transitioning from the MEM stage to the subsequent instruction, will introduce a single NOP, causing an increase in CPI from 1 to 1.2. Therefore, 0.2 out of the 1.2 cycles account for NOPs, equating to 17% of the total cycles.

**6.6** Let us assume that we cannot afford to have three-input multiplexors that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). What is the CPI for each option?

With MEM/WB register unavailable, Average NOPs of 0.05*0 + 0.2*2 + 0.05*1 + 0.10*1 + 0.10*1 = 0.65 stalls/instruction So, CPI = 1.65

With EX/MEM register unavailable, Average NOPs of 0.05*1 + 0.2*1 + 0.1*1 = 0.35 stalls/instruction So, CPI = 1.35

**6.7** For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by each type of forwarding (EX/MEM, MEM/WB, for full) as compared to a pipeline that has no forwarding?

| | No Forwarding | EX/MEM | MEM/WB | Full Forwarding |
|---|---|---|---|---|
| CPI | 1.85 | 1.65 | 1.35 | 1.2 |
| Period | 120ps | 120ps | 120ps | 130ps |
| Time | 222ps | 198ps | 162ps | 156ps |

| | | | | |
|---|---|---|---|---|
| Speedup | ref | 1.12 | 1.37 | 1.42 |

**6.8** What would be the additional speedup (relative to the fastest processor from 6.7) be if we added "timetravel" forwarding that eliminates all data hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100 ps to the latency of the full-forwarding EX stage.

CPI with full forwarding is 1.2.

CPI for "time travel" forwarding is 1.0 [Max with 0 Hazards]

Clock period for full forwarding is 130.

Clock period for zero hazards forwarding is 230 Speedup = Told / Tnew = (1.2*130)/ (1*230) = 0.68

**7.** Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

```
add x15, x12, x11
ld  x13,  4(x15)
ld x12, 0(x2)
or x13, x15, x13
sd x13, 0(x15)
```

---

**7.1** If there is no forwarding or hazard detection, insert `NOPs` to ensure correct execution.

add x15, x12, x11
nop
nop
ld x13, 4(x15)
ld x12, 0(x2)
nop
or x13, x15, x13
nop
nop
sd x13, 0(x15)

**7.2** Now, change and/or rearrange the code to minimize the number of `NOPs` needed. You can assume register `x17` can be used to hold temporary values in your modified code.\

Impossible to reduce the number of NOPs.

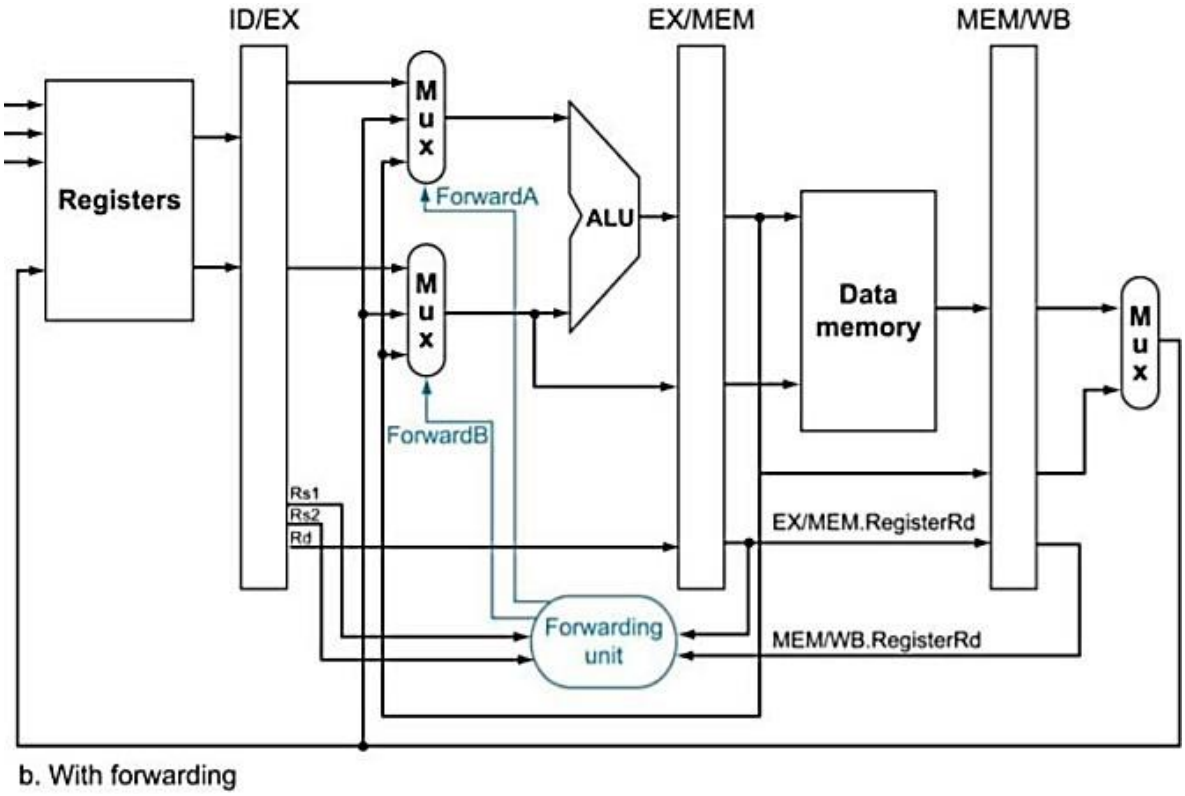
**7.3** If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

The code executes correctly.

**7.4** If there is forwarding, for the first seven cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.53 of the RISC V text (reproduced below).

| 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. |

| Clock Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **add** | IF | ID | EX | MEM | WB | | | | |
| **ld** | | IF | ID | EX | MEM | WB | | | |
| **ld** | | | IF | ID | EX | MEM | WB | | |
| **or** | | | | IF | ID | EX | MEM | WB | |
| **sd** | | | | | IF | ID | EX | MEM | WB |




(1) ForwardA = X; ForwardB = X

(2) ForwardA = X; ForwardB = X

(3) ForwardA = 0; ForwardB = 0

(4) ForwardA = 2; ForwardB = 0

(5) ForwardA = 0; ForwardB = 0

(6) ForwardA = 0; ForwardB = 1

(7) ForwardA = 0; ForwardB = 2

**7.5** If there is no forwarding, what new input and output signals do we need for the hazard detection unit in the Figure above? Using this instruction sequence as an example, explain why each signal is needed.

The hazard detection unit requires the values of rd obtained from the MEM/WB register. If the instruction presently in the ID stage relies on a value produced by (or forwarded from) either the instruction in the EX stage or the instruction in the MEM stage, it needs to be stalled. Therefore, it is necessary to examine the destination register of these two instructions. The Hazard unit already possesses the value of rd from the EX/MEM register as inputs, so incorporating the value from the MEM/WB register is the only addition required. No extra outputs are necessary. The pipeline can be stalled using the three existing output signals. The value of rd from EX/MEM is crucial for detecting the data hazard between the add and the subsequent ld. Simultaneously, the value of rd from MEM/WB is essential for identifying the data hazard between the initial ld instruction and the subsequent or instruction.

**7.6** For the new hazard detection unit from Problem 6.5 of this HW assignment, specify which

output signals it asserts in each of the first five cycles during the execution of this code.

| Clock Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| add | IF | ID | EX | MEM | WB | | | | |
| ld | | IF | ID | - | - | EX | MEM | WB | |
| ld | | | IF | - | - | ID | EX | MEM | WB |

(1) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(2) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(3) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(4) PCWrite = 0; IF/IDWrite = 0; control mux = 1

(5) PCWrite = 0; IF/IDWrite = 0; control mux = 1