# 154B Discussion 5

# Goals

- Assignment 3.2
  - Data Hazards.
  - Control Hazards.

# Data Hazards in 5-stage pipeline: RAW

- Occurs when a young instruction reads from a register that is updated by an older instruction.

  E.g.,

  addi t1, t2, 0x400

  xor t4, x5, x6

  mul t3, t1, t2

- We have two options,
  - Option 1: Stalling the pipeline such that the mul instruction won't reach the EX stage until the addi instruction finished updating the register (the cycle after addi WB stage).
  - Option 2: When the mul instruction is at the EX stage, the addi will be in the WB stage, and we can forward the value of t1 from WB stage to EX stage.
- You'll implement option 2 for assignment 3.2.

# Data Hazards in 5-stage pipeline: WAW

- WAW occurs when two instructions update the same register.
- It is not a hazard in a 5-stage pipeline.

  E.g.,

  add t0, t1, t2

  sub t2, t3, t4

  mul t0, t1, t3 // The value of t0 in the add instruction does not affect the correctness of this instruction or any instruction followed the mul instruction

# Data Hazards in 5-stage pipeline: WAR

- WAR occurs when the younger instruction write to a register that is read by an older instruction.
- It is not a hazard in a 5-stage pipeline.

  E.g.,

  add t0, t1, t2

  div t5, t6, t2

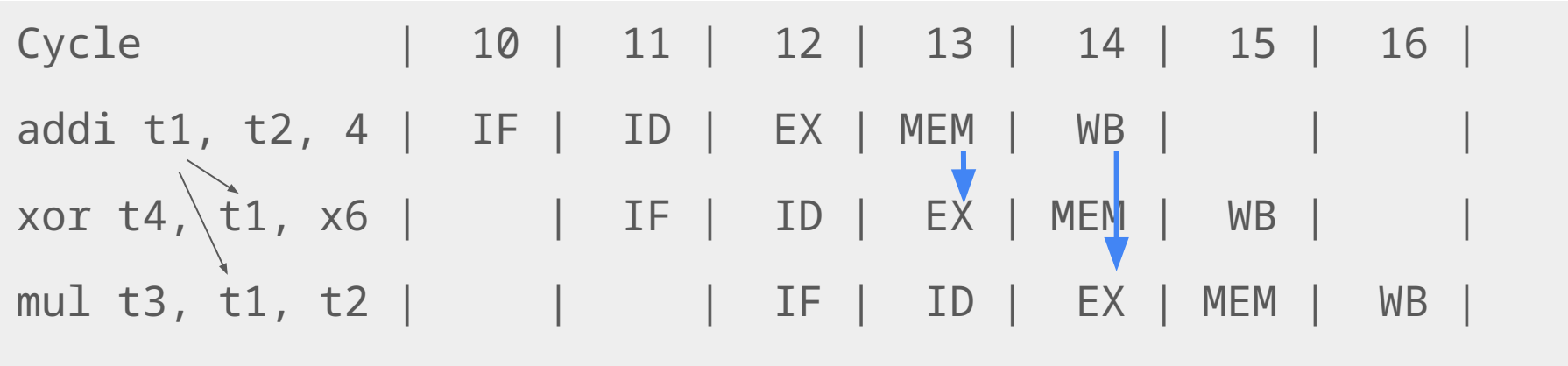
  sub t2, t3, t4 // The value of t2 in the add and the div instructions does not affect the correctness of this instruction or any instruction followed the sub instruction

# Data Hazards in 5-stage pipeline

- WAW and WAR dependencies are not hazards in the 5-stage pipeline because there’s only one instruction in each stage at a time.

# Forwarding

- The updated register value can be forwarded from MEM stage and WB stage to EX stage.
- Forwarding from one stage to another must be done in a cycle! I.e., no arrows span multiple cycles.
- Note: there is no forwarding between the MEM stage and the WB stage.



# Implementing Forwarding

- Update the components/forwarding.scala file.
- The forwardA signal determines which value to use for operand1 of the instruction EX stage.
  - forwardA = 0: operand1 will use value read from the register file, i.e. readdata 1.
  - forwardA = 1: operand1 will use value (either the ALU result or the immediate) from the MEM stage.
  - forwardA = 2: operand1 will use value (either the ALU result or the immediate or the readdata) from the WB stage.
  - forwardB is similar to forwardA.

# Load-to-use Hazard

- In DINO CPU, forwarding works well for almost all combinations of instructions. Almost!
- There is a corner case that simply forwarding values wouldn't work with DINO CPU.
- Load-to-use hazard: occurs when a load instruction writes to a register that the *next* instruction read from.

# Load-to-use Hazard

- In class, we make an assumption that, the data read from data memory won't be complete until the end of the load MEM stage, thus we can only use the readdata in WB stage.
- If we don't make that assumption, it's possible to forward from MEM to EX in this case. But why we don't remove that assumption?
  - E.g., if the time to complete the EX stage is 250ps, and time to complete the MEM stage is 280ps, and the current cycle time is 280ps. If we forward from MEM stage to EX stage, then the EX stage must wait until the MEM stage to complete (280ps) before doing its own operation (250ps). Thus, the new time to complete the EX stage is 280ps + 250ps = 530ps, and the new cycle time would be 530ps. Effectively, we just double the cycle time due to the possibility of forwarding from MEM to EX in this particular case.

# Load-to-use Hazard

- So, there shouldn't be forwarding from MEM to EX in this case.
- Instead, in this case, there should be a forwarding from load's WB stage to the next instruction's EX stage.
- This hazard is solved by stalling the decode stage of the next instruction.
- The logic for stalling is implemented in the hazard unit located in components/hazard.scala.

# Load-to-use Hazard

- How the pipeline looks if we stall the decode stage of xor?

```
                |  IF |  ID |  EX | MEM |  WB |
      cycle = 10 |  ld | --- | --- | --- | --- |
      cycle = 11 | xor |  ld | --- | --- | --- |
      cycle = 12 | --- | xor |  ld | --- | --- |
      cycle = 13 | --- | xor | nop |  ld | --- |
      cycle = 14 | --- | --- | xor | nop |  ld |
```

# Control Hazards

- Also recall: when an instruction enters its decode stage, it will reach execute/memory/writeback stages, unless it is flushed at some point.
- When an instruction reaches memory/writeback stages, it might update memory and register file, or forward value back to EX stage, or update the next PC.
- In other words, we don’t want incorrect instruction to ever enter MEM stage and WB stage.

# Control Hazards

- Recall: branch/jump instructions might alternate the next PC of that instruction to a value other than PC+4.
- A jump instruction (jal/jalr) always updates the next PC of that instruction to a value other than PC+4.
- A branch instruction,
  - When it’s resolved to taken, it updates the next PC of that instruction to a value other than PC+4.
  - When it’s not taken, the next PC of that instruction is PC+4.

# Control Hazards

- Recall: branch/jump instructions might alternate the next PC of that instruction to a value other than PC+4.
- Because we don’t know the next PC after a branch/jump instruction, we don’t know where is the next instruction.
- The next PC, as well as the branch result (taken/not-taken), won’t be known until the branch/jump instruction reaches the MEM stage.

# Control Hazards

- For assignment 3, we won't stall the pipeline until the branch/jump is resolved. Instead, we will *speculatively* assume that the next PC of the branch/jump instruction to be PC+4, i.e. always predict that the branch is not taken. Thus, the instructions followed the branch/jump instructions might be incorrect.
- Since we always predict the branch is not taken,
  - We always have a next PC misprediction for every jump instruction.
  - If a branch is taken, we have a misprediction.

# Control Hazards

- Since we know the branch result and the next PC at the branch/jump MEM stage, then when the branch/jump instruction reaches the MEM stage, we must decide whether we flush the pipeline.

# Control Hazards

- Since we know the branch result and the next PC at the branch/jump MEM stage, then when the branch/jump instruction reaches the MEM stage, we must decide whether we flush the pipeline.

```
          |  IF |  ID |  EX | MEM |  WB |
PC = 0x10 | bne | --- | --- | --- | --- |
PC = 0x14 | add | bne | --- | --- | --- |
PC = 0x18 | mul | add | bne | --- | --- |
PC = 0x1c | rem | mul | add | bne | --- | // misprediction
PC = 0x80 | --- | nop | nop | nop | bne |
```

- Assume that bne results in taken.
- By flushing the IF/ID, ID/EX, and EX/MEM stage registers, we don’t allow incorrect instructions (add, mul, rem) to enter the MEM and WB stage.
- At this point, PC is updated so the instruction at the fetch stage is a correct instruction.

# Control Hazards

- Similar for jump instructions, except that a jump instruction always results in a misprediction.

```
          |  IF  |  ID  |  EX  |  MEM |  WB  |
PC = 0x2c | jalr | ---- | ---- | ---- | ---- |
PC = 0x30 |  sll | jalr | ---- | ---- | ---- |
PC = 0x34 |  mul |  sll |  bne | ---- | ---- |
PC = 0x38 |   or |  mul |  sll |  bne | ---- | // misprediction
PC = 0x10 | ---- |  nop |  nop |  nop | jalr |
```

# How to implement stalling?

- `stage_reg.io.valid:`
  - if true.B, the stage_reg will be updated accordingly to the inputs, i.e., signals from stage_reg.io.in.*
  - if false.B, the stage_reg will not take stage_reg.io.in.* as an input.
- `stage_reg.io.flush:`
  - if true.B, the stage_reg will be zeroed out (equivalent to NOP).
  - if false.B, the stage_reg will not be zeroed out.

| | valid === true.B | valid === false.B |
|---|---|---|
| flush === true.B | zeroed out | zeroed out |
| flush === false.B | stage_reg.io.in.* as input | stage_reg stays the same |

# How to implement stalling/flushing?

E.g., by setting the following values, the mul instruction does not advance to the EX stage.

```
IF_ID.io.valid = false.B

IF_ID.io.flush = false.B
```

```
           |  IF |  ID |  EX | MEM |  WB |
cycle = 10 | --- | mul |  ld | --- | --- |
cycle = 11 | --- | mul | ??? |  ld | --- |
```

# How to implement stalling/flushing?

E.g., when the ld instruction advances to the MEM stage, while the mul instruction is stalled at the ID stage, then the EX stage mustn't contain any instruction. Effectively, we want to flush the ID/EX stage register, and replace the instruction at EX stage by a NOP.

```
ID_EX.io.valid = true.B

ID_EX.io.flush = true.B
```

```
           |  IF |  ID |  EX | MEM |  WB |
cycle = 10 | --- | mul |  ld | --- | --- |
cycle = 11 | --- | mul | nop |  ld | --- |
```