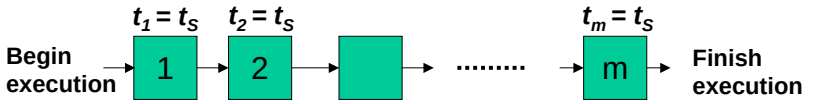
L12: Pipelining Hazards

Adapted from:

CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

What is "Pipelining"?

- An implementation technique in which multiple instructions (tasks) are overlapped in execution.
- A pipeline consists of several sections, called stages



- Multiple tasks operating simultaneously using different resources (stages)
- Tasks flow from each stage to the next at the same pace. The rate of progress is limited by the slowest pipeline stage – stage time, t_S

Features of Pipelining

- Pipelining doesn't help latency of single task, it helps throughput of entire workload
  - Latency (time to complete executing one instruction)

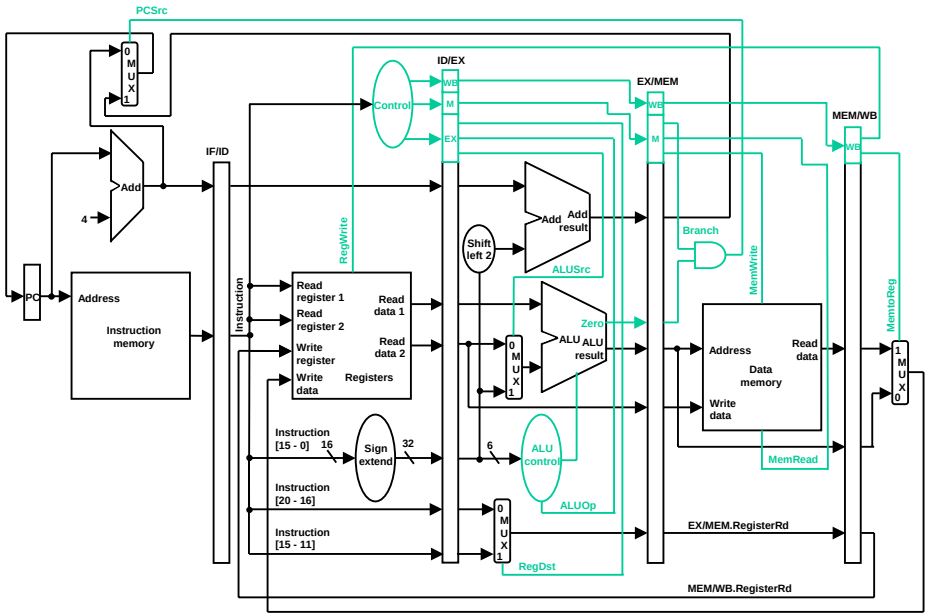
$$t_l = m \times t_S$$

  - Throughput (inversely proportional to time to process entire job [time for first instruction plus time to finish the remaining instructions])

$$\propto 1/(t_l + (n-1) \cdot t_S)$$

- Potential speedup = Number pipe stages (m)
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Time spent stalled waiting for dependences also reduces speedup

Pipelined datapath and control (so far)



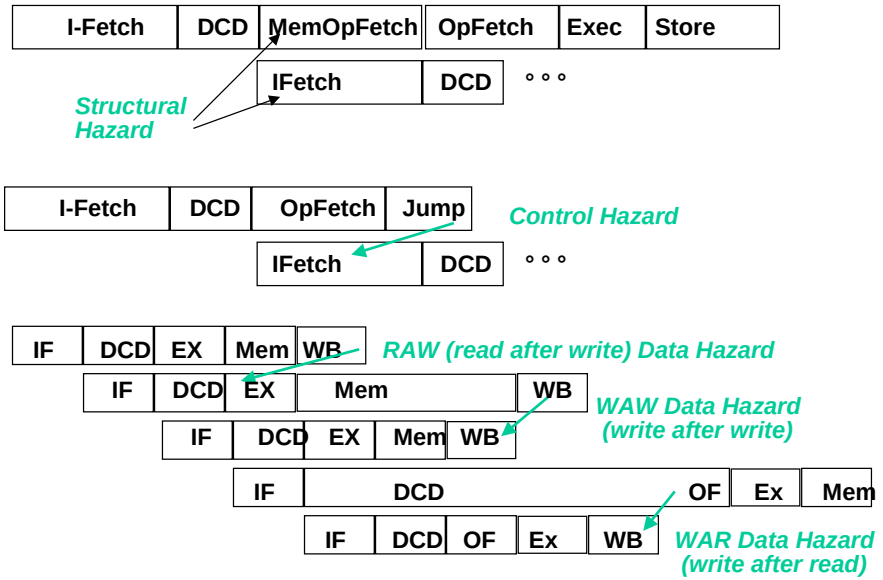
Exercise

[P&H 6.11] Consider executing the following code on the pipelined datapath of Figure 6.46 on page 492 (P&H's textbook):

```
add   $1,   $2,   $3
add   $4,   $5,   $6
add   $7,   $8,   $9
add   $10,  $11,  $12
add   $13,  $14,  $15
```

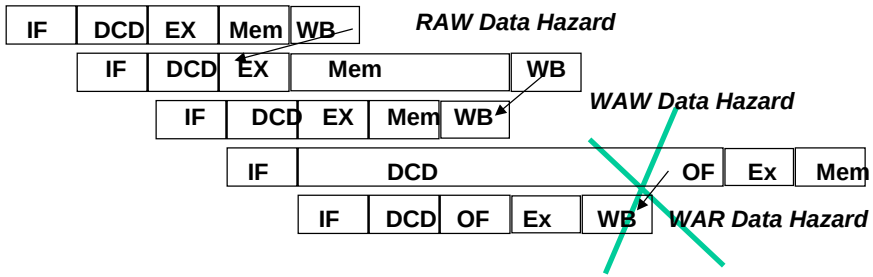
At the end of the fifth cycle of execution, which registers are being read and which register will be written?

Potential Pipeline Hazards



Data Hazards

- **Avoid some "by design"**
  - **eliminate WAR by always fetching operands early (DCD) in pipe**
  - **eleminate WAW by doing all WBs in order (last stage, static)**
- **Detect and resolve remaining ones**
  - **stall or forward (if possible)**



Exercise

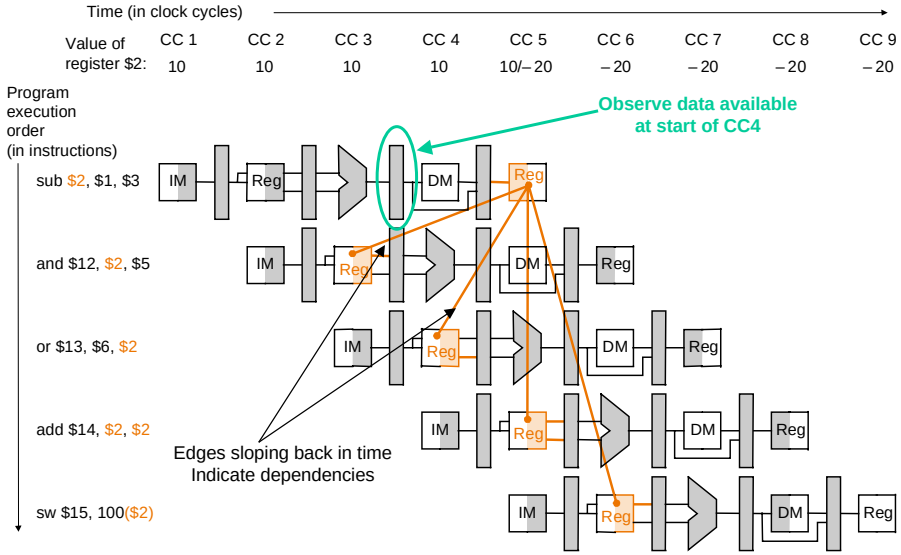
[P&H 6.4] Identify all of the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding?

```
add   $2,   $5,   $4
add   $4,   $2,   $5
sw    $5,   100($2)
add   $3,   $2,   $4
```

Equivalence of graphical representations

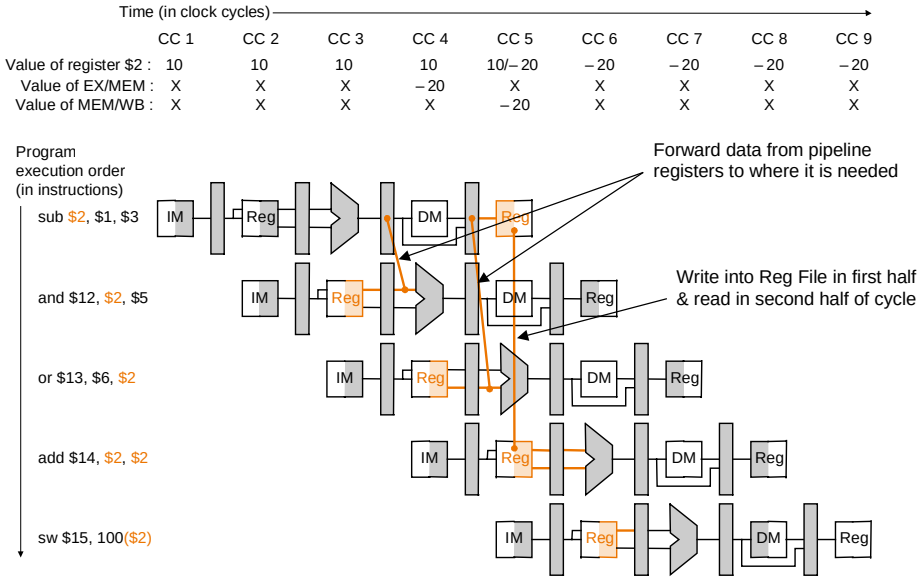
Data hazards

- **Happen when there are data dependencies between instructions executing in the pipeline**



Solution: Forwarding

- **A very efficient approach in avoiding hazard**

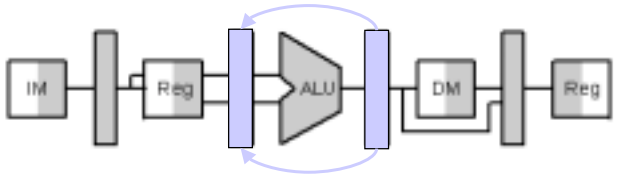


Implementing forwarding

- **Detect data hazards**
  - **Check for data dependencies between each instruction and the preceding instructions in the pipeline**
- **Forward the proper value**

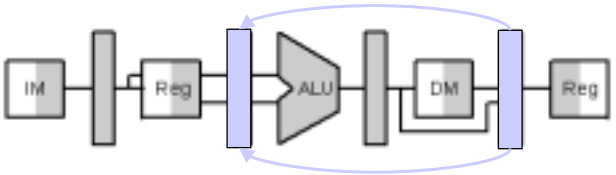
Detecting a data hazard (1)

- Based on pipeline register field contents
- Dependent on the immediately preceding instruction
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
- Dependent on the instruction before that
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt



Detecting a data hazard (2)

- Based on pipeline register field contents
- Dependent on the immediately preceding instruction
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
- Dependent on the instruction before that
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt



Detecting a data hazard – example

```
Sub    $2, $1, $3     IF  ID  EX  ME  WB
And    $12, $2, $5        IF  ID  EX  ME  WB
Or     $13, $6, $2            IF  ID  EX  ME  WB
Add    $14, $2, $2                IF  ID  EX  ME  WB
Sw     $15, 100($2)               IF  ID  EX  ME  WB
```


Detecting a data hazard – example

```
Sub    $2, $1, $3     IF  ID  EX | ME  WB
And    $12, $2, $5        IF  ID | EX  ME  WB
Or     $13, $6, $2            IF  ID  EX  ME  WB
Add    $14, $2, $2                IF  ID  EX  ME  WB
Sw     $15, 100($2)               IF  ID  EX  ME  WB
```

Dependent instructions: sub-and

Data hazard type: 1a

Detecting a data hazard – example

```
Sub   $2, $1, $3      IF  ID  EX  ME  WB
And   $12, $2, $5         IF  ID  EX  ME  WB
Or    $13, $6, $2             IF  ID  EX  ME  WB
Add   $14, $2, $2                 IF  ID  EX  ME  WB
Sw    $15, 100($2)                    IF  ID  EX  ME  WB
```

Dependent instructions: sub-and, sub-or

Data hazard type: 1a, 2b

Detecting a data hazard – more considerations

- **Previous instructions must**
  1. **Write back a result**
  2. **If they write to $0, the result is never stored, and hence need not be forwarded**
     - **Recall: $0 is a special register in MIPS that always has value 0**

Revised Detect Condition – EX hazard

- **Data forwarded from EX stage:**

Detection:

```
IF    EX/MEM.RegWrite
AND   EX/MEM.RegisterRd != 0
AND   EX/MEM.RegisterRd = ID/EX.RegisterRs
      forward ALU result to first ALU op

IF    EX/MEM.RegWrite
AND   EX/MEM.RegisterRd != 0
AND   EX/MEM.RegisterRd = ID/EX.RegisterRt
      forward ALU result to second ALU op
```


Revised Detect Condition – MEM hazard

- **Data forwarded from MEM stage:**

Detection:

```
IF    MEM/WB.RegWrite
AND   MEM/WB.RegisterRd != 0
AND   MEM/WB.RegisterRd = ID/EX.RegisterRs
      forward ALU result or MEM data to first ALU op

IF    MEM/WB.RegWrite
AND   MEM/WB.RegisterRd != 0
AND   MEM/WB.RegisterRd = ID/EX.RegisterRt
      forward ALU result or MEM data to second ALU op
```

Multiple data dependencies

- **What if a data is dependent on both EX and MEM results?**

  Add $1, $1, $2
  Add $1, $1, S3
  Add $1, $1, $4

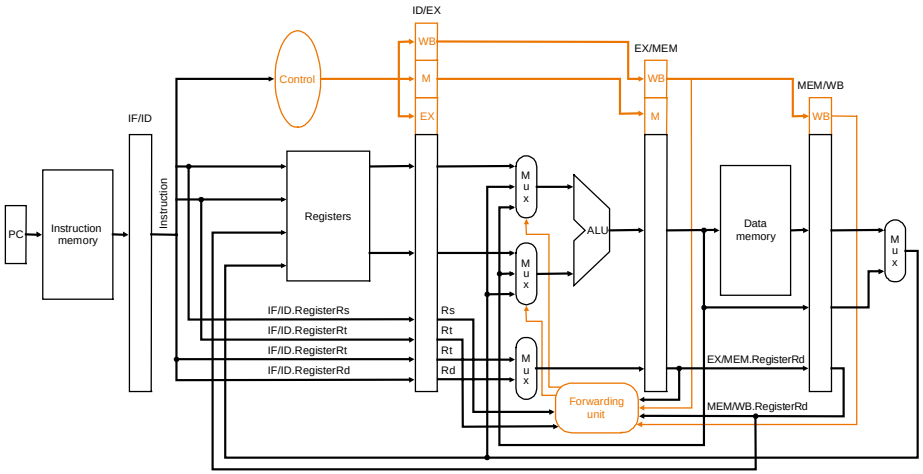
- **Forward the more recent results – forward the EX result**

Detection:

IF **MEM/WB.RegWrite**
AND **MEM/WB.RegisterRd != 0**
AND **EX/MEM.RegisterRd != ID/EX.RegisterRs**
AND **MEM/WB.RegisterRd = ID/EX.RegisterRs**
forward ALU result or MEM data to first ALU op

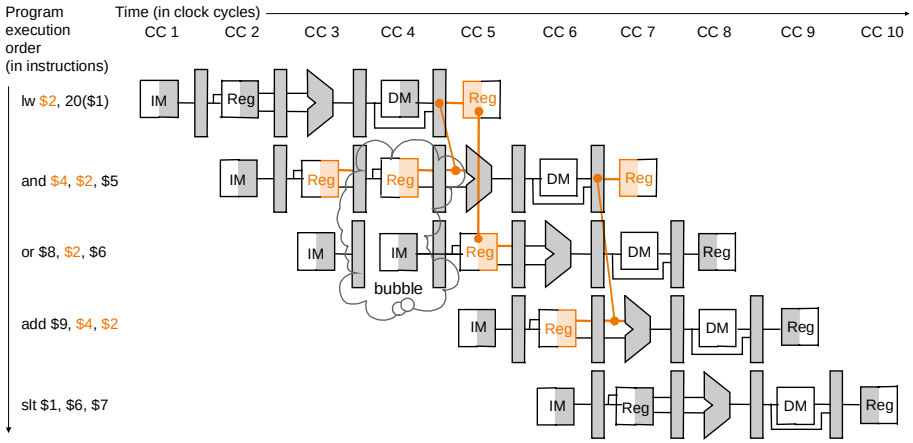
IF **MEM/WB.RegWrite**
AND **MEM/WB.RegisterRd != 0**
AND **EX/MEM.RegisterRd != ID/EX.RegisterRt**
AND **MEM/WB.RegisterRd = ID/EX.RegisterRt**
forward ALU result or MEM data to second ALU op

Datapath with modified forwarding unit



When forwarding does not work ⇒ STALL

- **Detect hazard**
  - **Stall at ID stage**



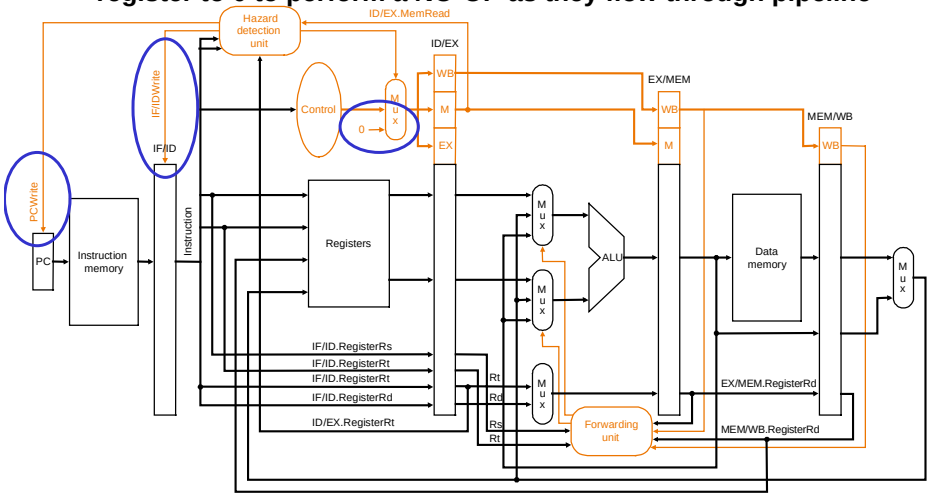
Detecting hazards

- **When a load instruction is followed by an instruction dependent on the memory data (load-use hazard)**
- **We add a hazard detection unit that checks for the following condition:**

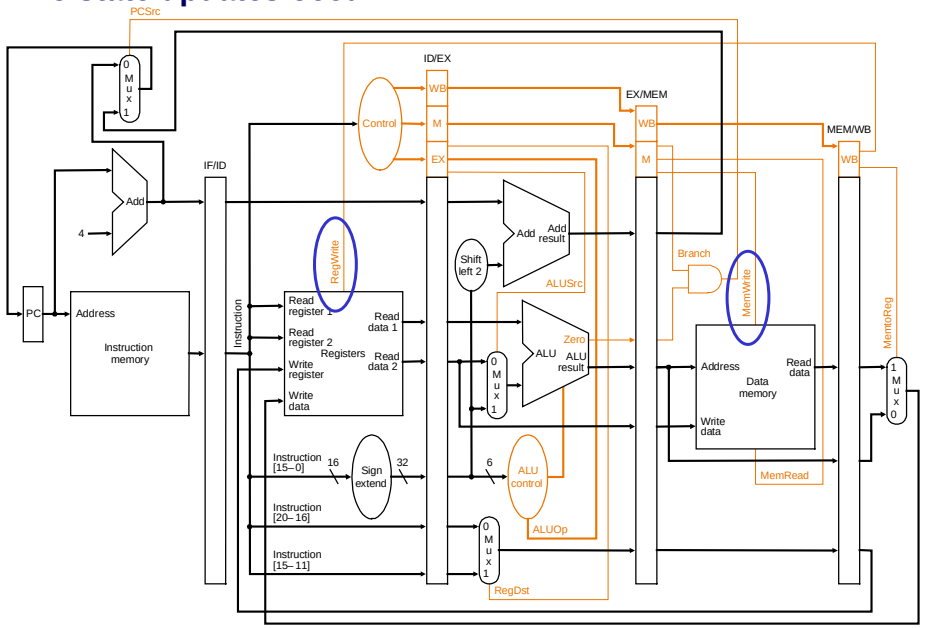
IF **ID/EX.MEMRead**
AND **(ID/EX.RegisterRt = IF/ID.RegisterRs**
OR **ID/EX.RegisterRt = IF/ID.RegisterRt)**
stall the pipeline

How do we stall the pipeline?

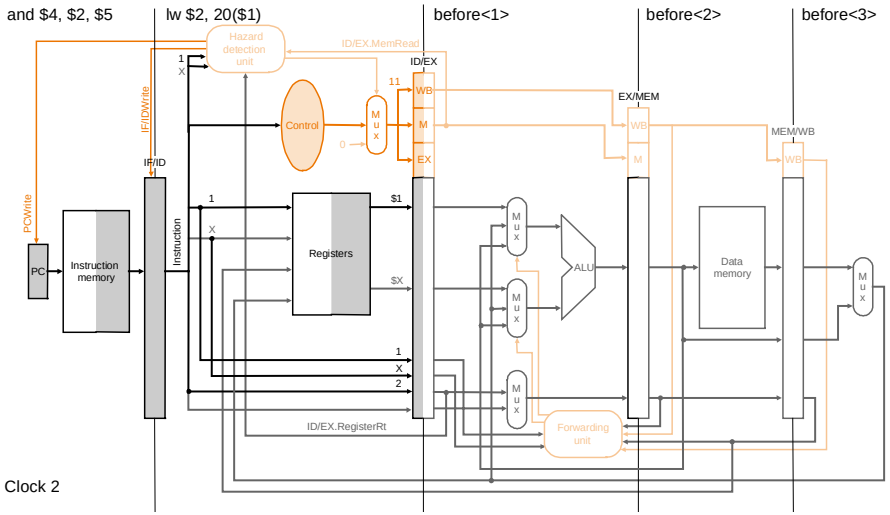
- Prevent the PC and IF/ID pipeline register from changing by not loading new values
- Set the EX, MEM and WB control fields of the ID/EX pipeline register to 0 to perform a NO-OP as they flow through pipeline



Setting the control signals to 0 does nothing (nop) since no state updates occur

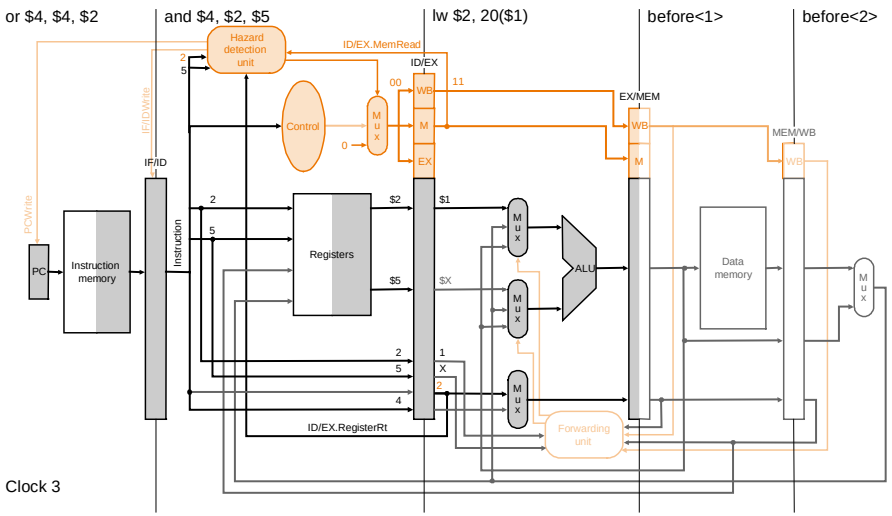


Load-use hazard (LUH) example



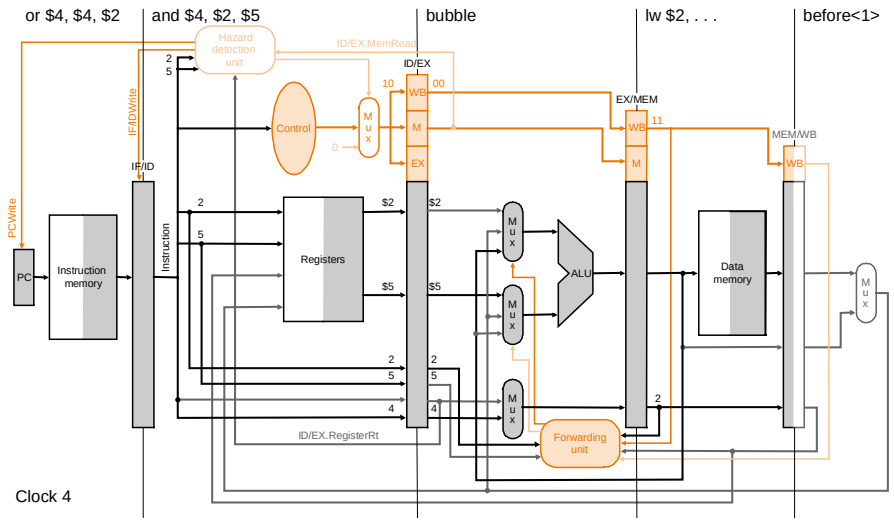
Clock 2

LUH example (2)



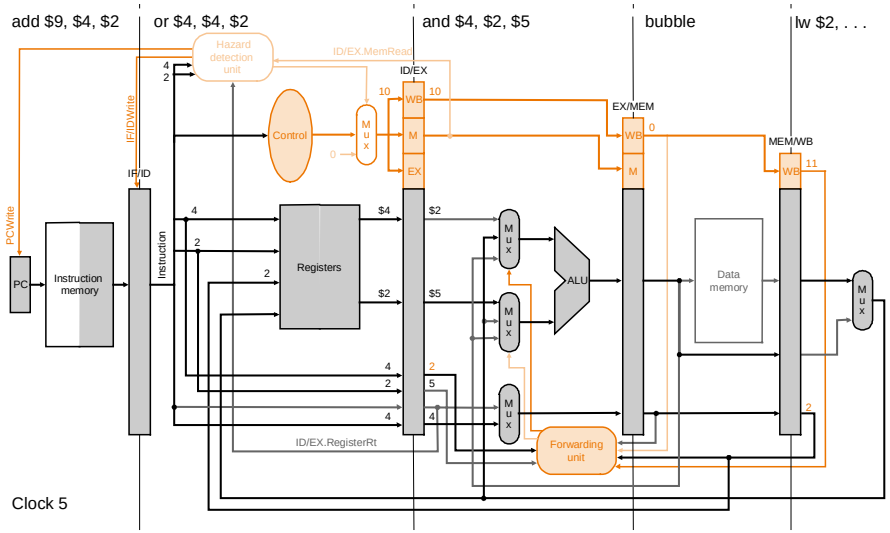
Clock 3

LUH example (3)



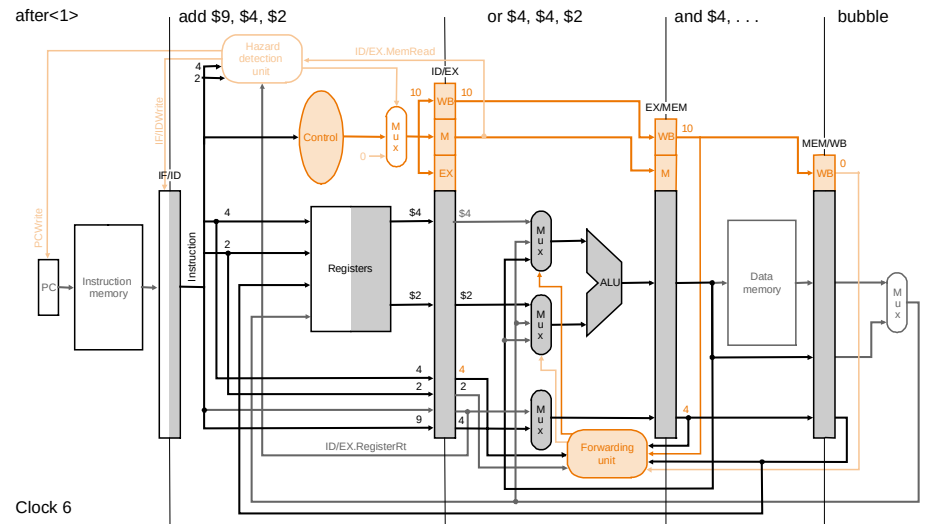
Clock 4

LUH example (4)



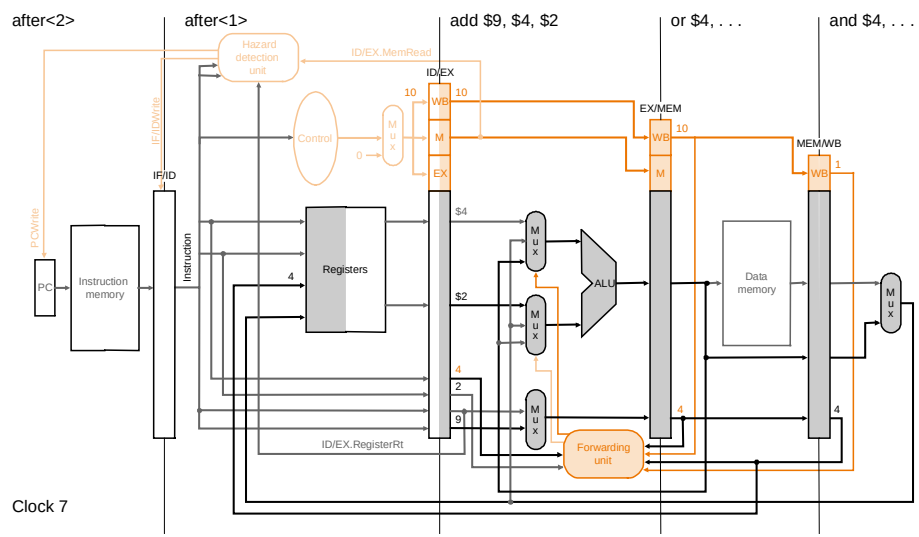
Clock 5

LUH example (5)



Clock 6

LUH example (6)



Clock 7

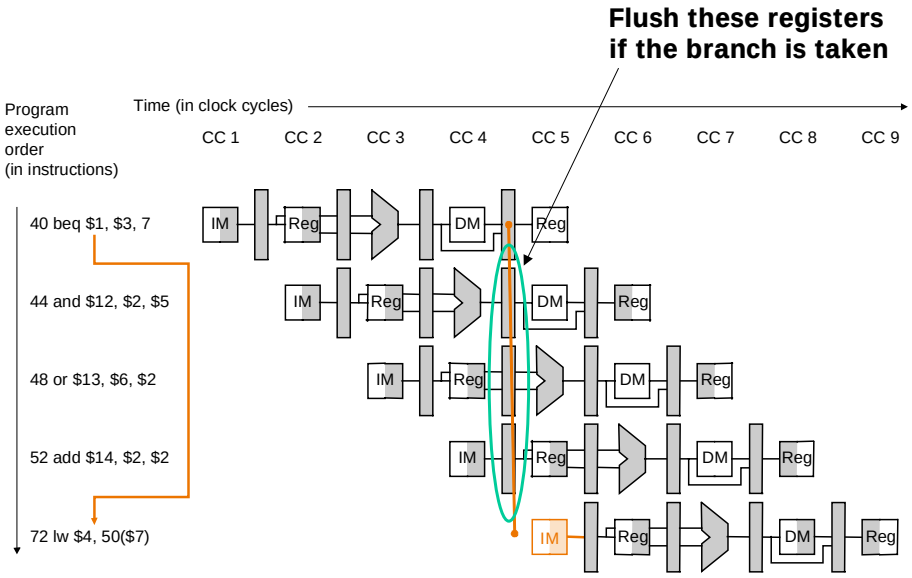
Control hazards

- Delay in determining the proper instruction to fetch is called a control hazard or branch hazard
- Solutions to control hazards
  - Stall
    - Wait until the branch decision is clear
      - Up to three cycles will be wasted for each branch depending upon the design of the datapath (availability of resources)
  - Branch prediction
    - static
    - dynamic
  - Branch delay

Static prediction

- Predict the branch always displays the same behavior
  - Never taken
  - Always taken
- When the prediction is wrong, the instructions being fetched, decoded, and executed after the branch instruction must be discarded (flushed) – here we assume the result of the branch is computed during the EX stage and the PC is updated during the MEM stage
- When the prediction is correct, the pipeline continues processing
- Better than stall
  - If the accuracy of the prediction is x, then there is 3/(3-2x) improvement in branch performance

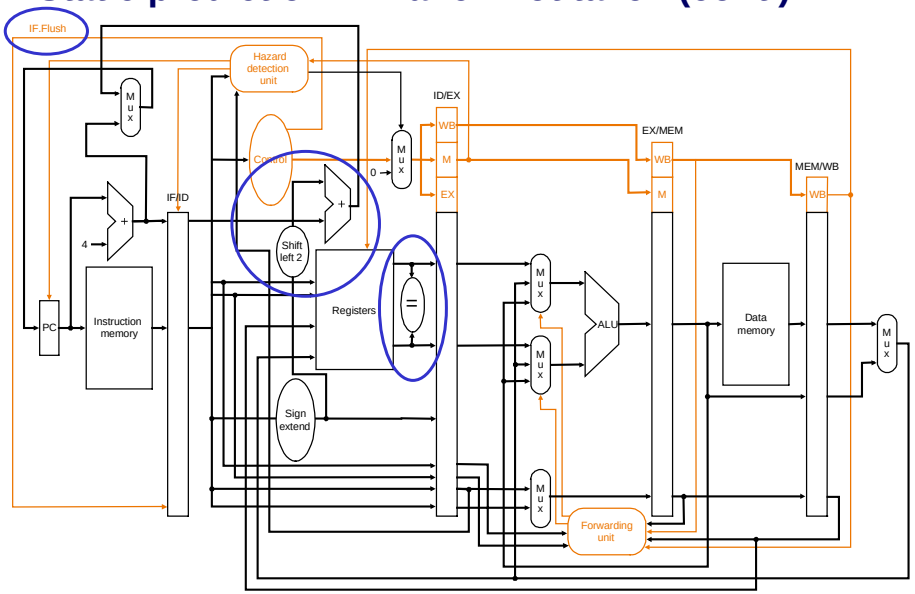
Static prediction – Branch not taken



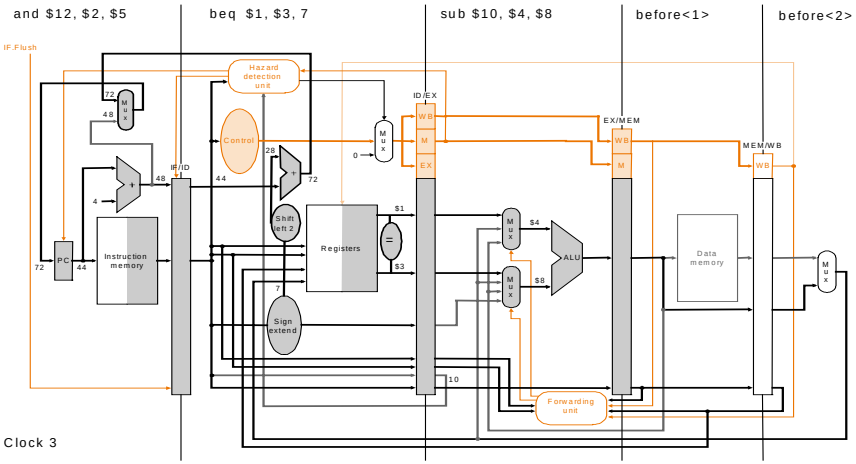
Static prediction – Branch not taken (cont.)

- Reducing the delay of branches – flushing 3 instructions is a heavy penalty!
  - Make branch decision earlier
    - Calculate the target address in the ID stage
    - Compare register contents in the ID stage
      - OR the results of parallel XOR tests on each bit (FAST, since no carry logic required)
  - Require forwarding to the register file and hazard detection if the branch is dependent upon results of an R-type or LOAD instruction still in the pipeline
  - Only one instruction then need be flushed
  - The control signal IF.Flush is used to flush the instruction in the IF/ID register
    - Sets the instruction field of IF/ID register to 0 (nop)

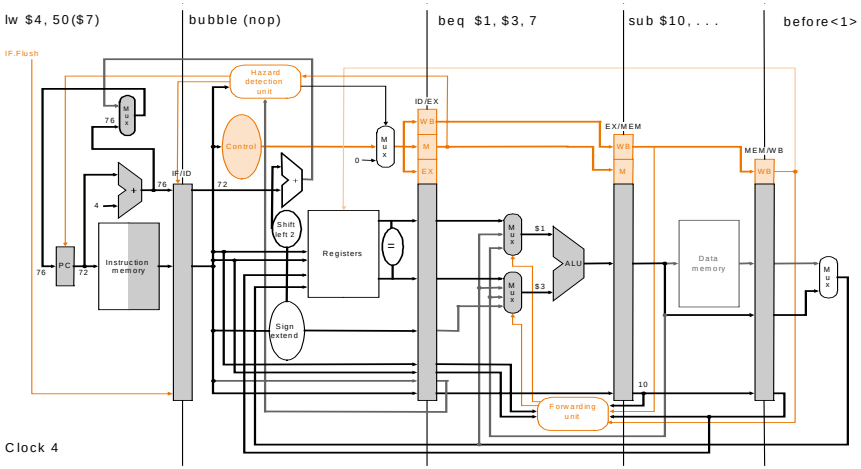
Static prediction – Branch not taken (cont.)



Handling misprediction(1)



Handling misprediction(2)



Dynamic prediction

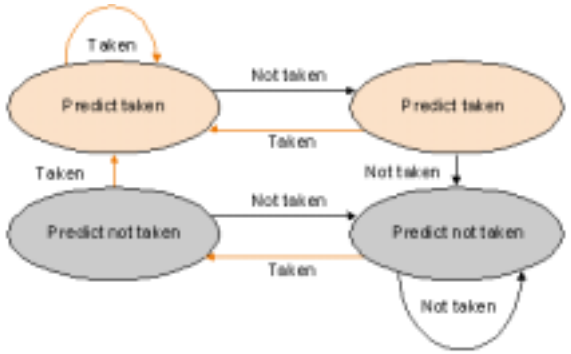
- **The prediction is made on the fly, depending on the history of the branch behavior**
- **The history is stored in a table or buffer, called the branch history table or branch prediction buffer**
- **Like a cache, the table consists of a portion of the branch instruction address and a branch history bit field**

1-bit prediction scheme

- The history bit field contains only 1 bit
- Prediction takes the bit value recorded from the last evaluation
  - If the prediction is wrong, the bit is changed
- Advantage
  - Simple
- Disadvantage
  - Double mis-prediction if branch is almost always taken:
  - Consider loop that is executed several times
    - With 1 bit scheme, we always predict incorrectly when we exit the loop
    - But, when we execute loop again, we mispredict on the first iteration since the bit was set to "not taken" at the end of the last execution
  - A 2-bit scheme is often used to overcome this problem

2-bit prediction

- History field has two bits
- A prediction must be wrong twice before it is changed
- Will mispredict once per loop invocation



Delaying a branch

- A compiler approach
- Avoids the uncertainty of a branch decision by inserting an independent instruction (immediately after the branch instruction) that can always be executed irrespective of the branch decision (does not need to be flushed)
- Is becoming less popular as pipelines become longer and issue more instructions per cycle
- There is a preference for using the increasing availability of transistors to implement dynamic predictors instead

Scheduling a branch delay slot