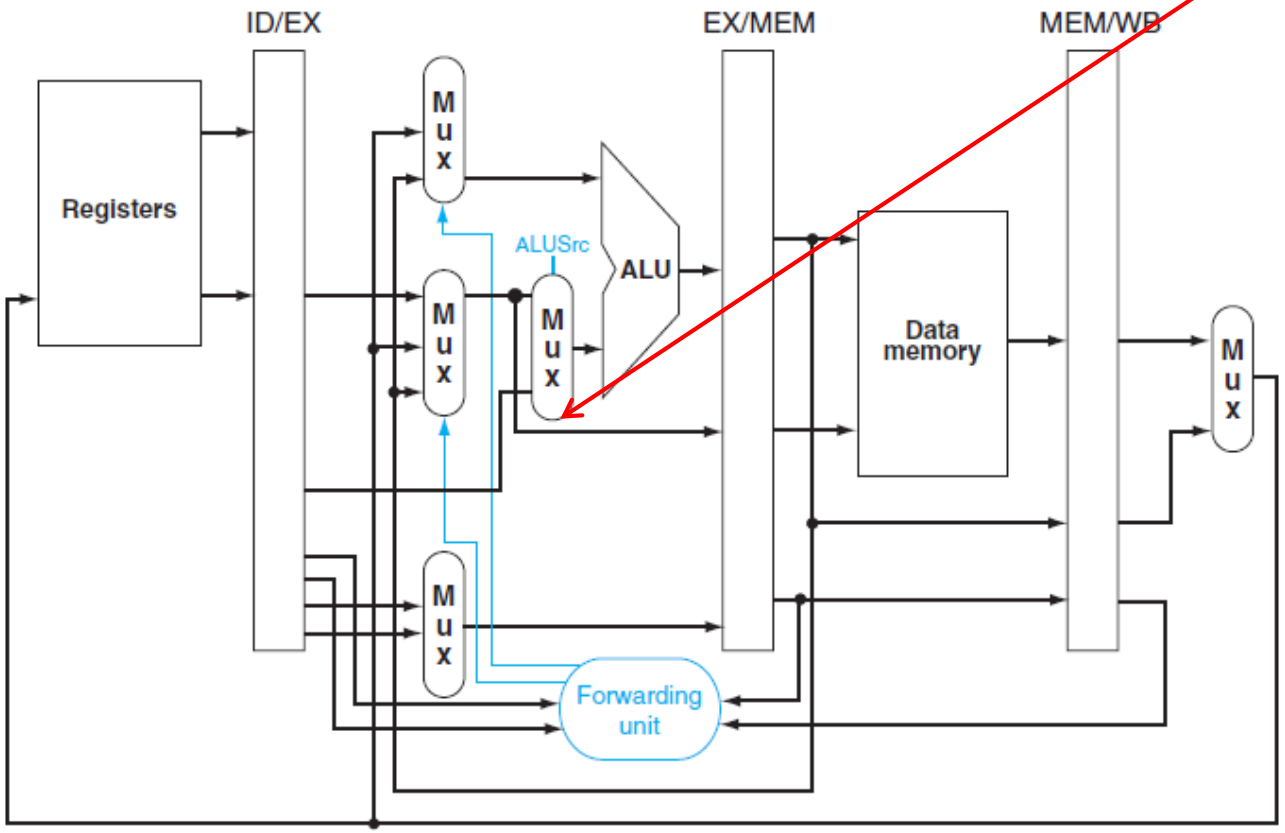
# EEL 4768
# Computer Architecture

## Pipelined Datapath

# Outline

- Forwarding Implementation
- Control Hazards

# Forwarding to EX: Implementation



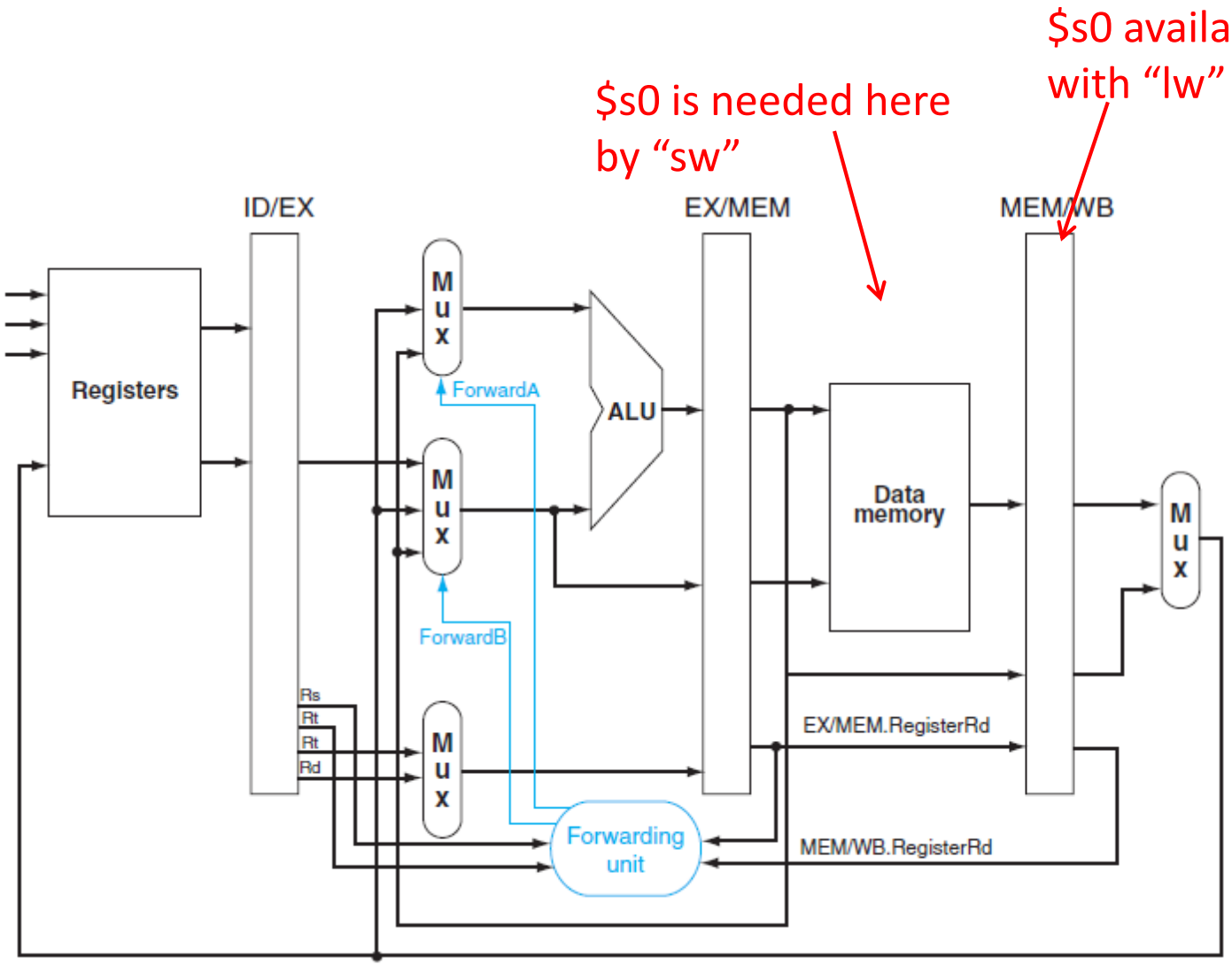

This multiplexer allows the 16-bit sign-extended number to go into the ALU.

This MUX is controlled by the main control unit. It decides between [Register] or [16-bit number].

The second MUX is controlled by the forwarding unit. It brings the correct register value.

# ‘lw’ followed by ‘sw’

- Need to forward from the MEM stage since it contains the most recent result
- We need to add a forwarding unit for: MEM/WB → MEM stage

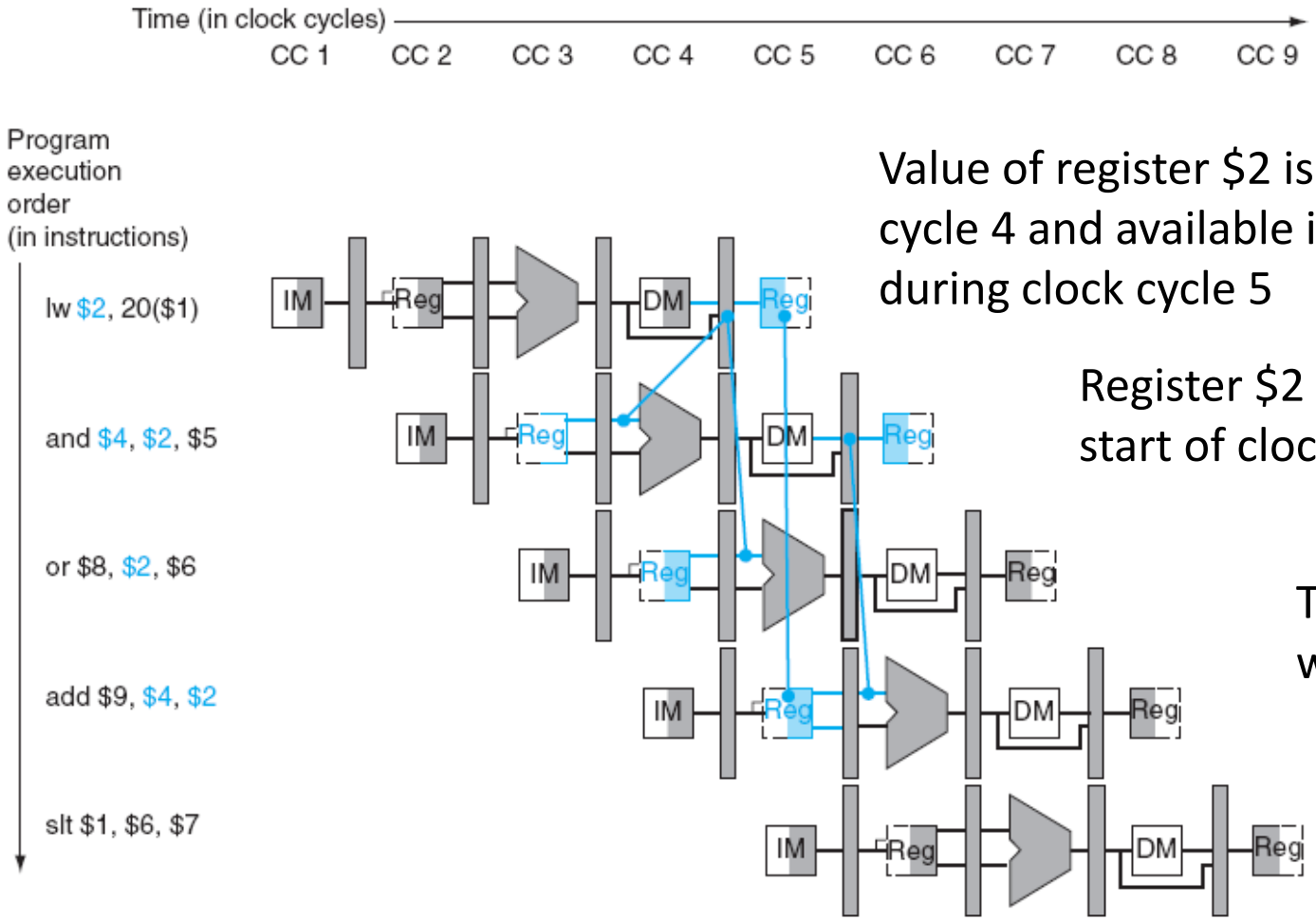



Example:

```
lw $s0, 0($a0)
sw $s0, 0($a1)
```

# Case where Forwarding is not Sufficient

- Load word (lw) changes register $2
- The following instruction takes register $2 as operand




Value of register $2 is obtained by lw in clock cycle 4 and available in the stage register during clock cycle 5

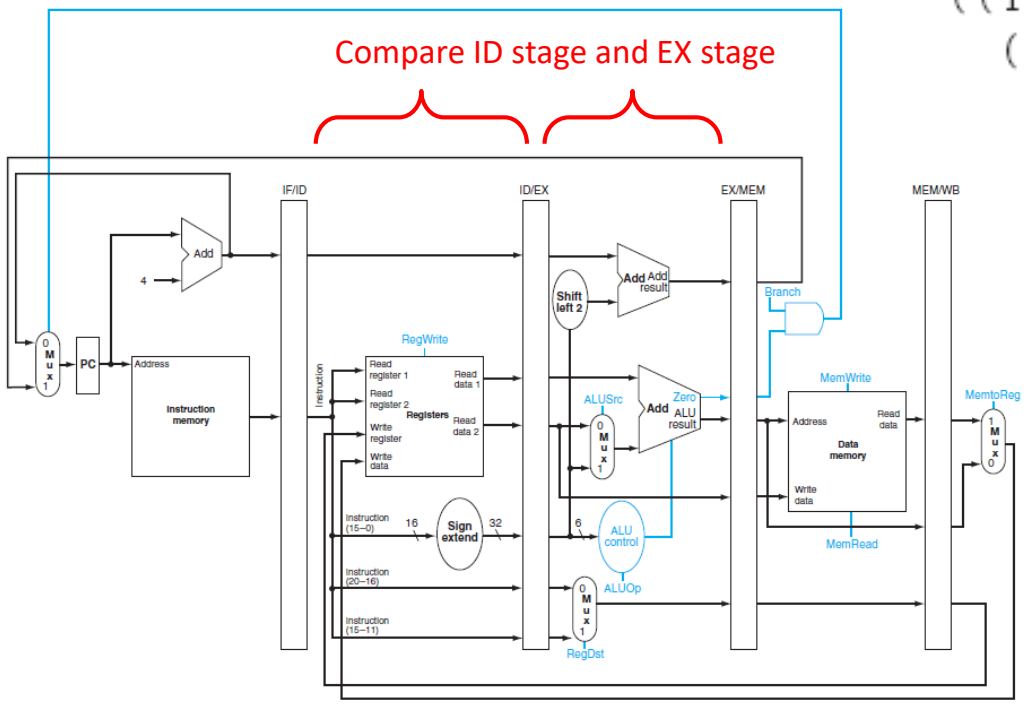
Register $2 is needed by and at the start of clock cycle 4

Therefore, forwarding won't solve the problem

# Stall the pipeline for 1 clock cycle

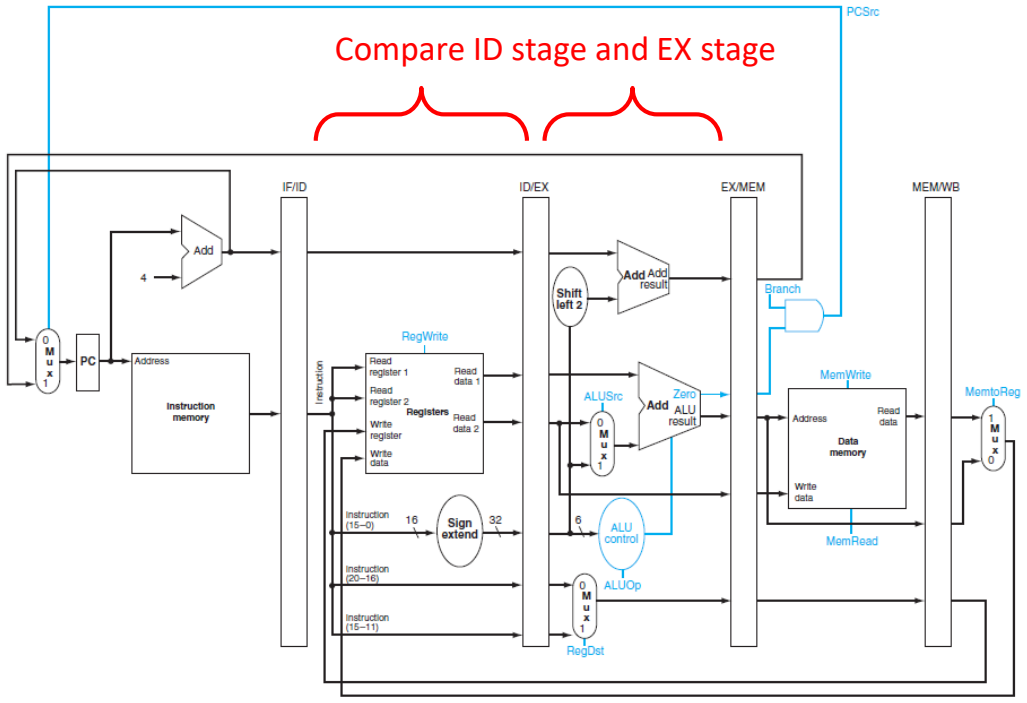
- We use a *hazard detection unit* to detect this situation:
  - Load word followed by an R-type with data dependency
- We detect the hazard in the ID stage
  - The R-type with data dependency is in ID stage; load is in EX stage
  - If the instruction in EX stage is load word (lw) and it will produce data that is used by either operands in ID stage, stall the pipeline

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
```



# Stalling the Pipeline

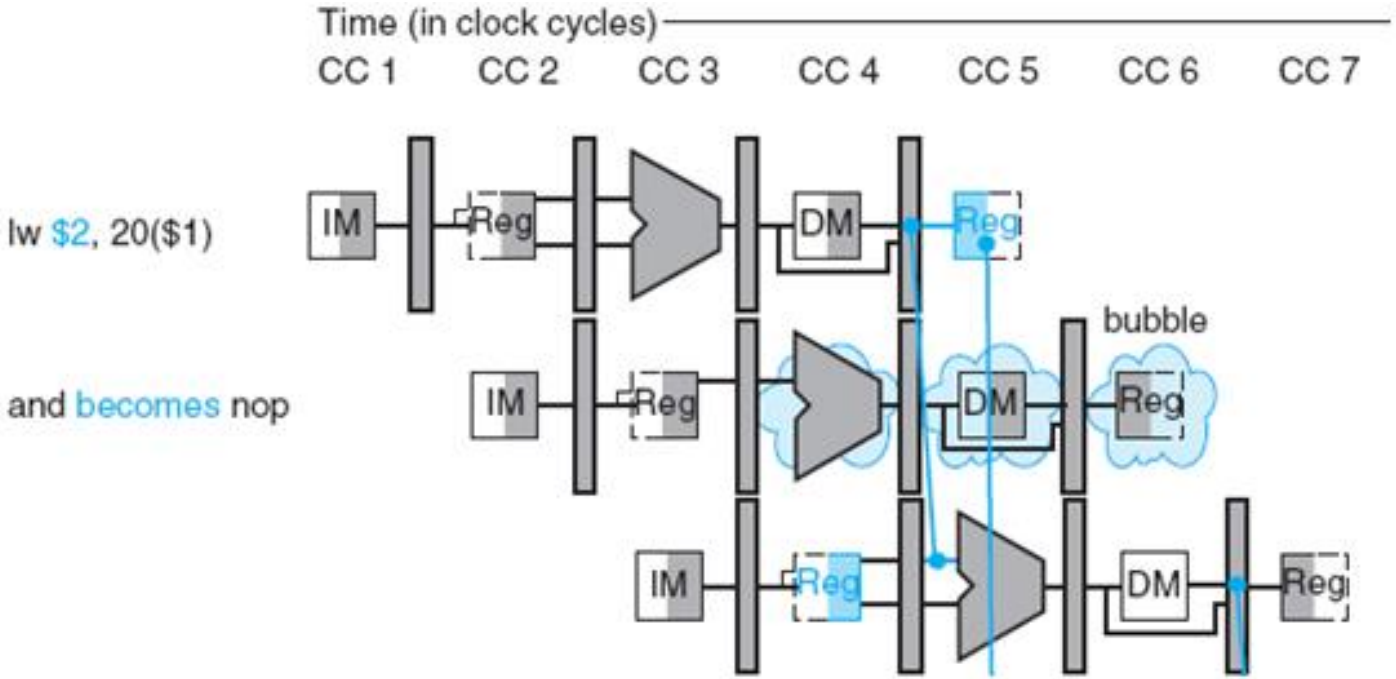
- If the instruction in ID stage is stalled, the instruction in the IF stage should also be stalled
- We accomplish this by preventing the IF/ID register and the PC register from changing
- Accordingly, the instruction in ID stage will not go to EX stage in the next clock cycle



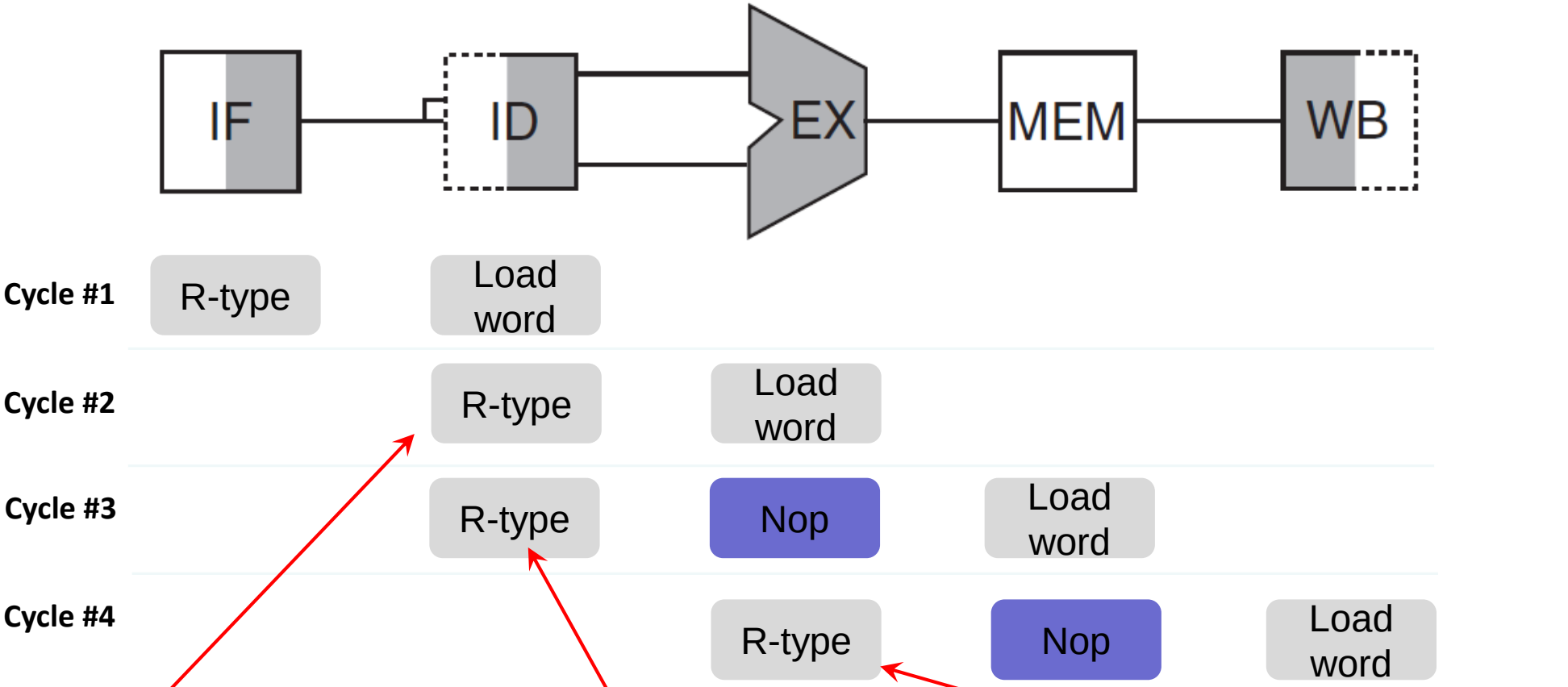

- Instead, we send a ‘nop’ instruction to the EX stage
- We do this by setting all the control signals to 0 in the ID/EX stage register

# Inserting a Nop Instruction (or bubble)

- CC 3: ‘and’ instruction is in the ID stage
- CC 4: ‘and’ instruction stays in the ID stage
  - A bubble is sent to the EX stage
- CC 5: ‘and’ instruction goes to the EX stage and receives forwarded data from the MEM/WB stage register



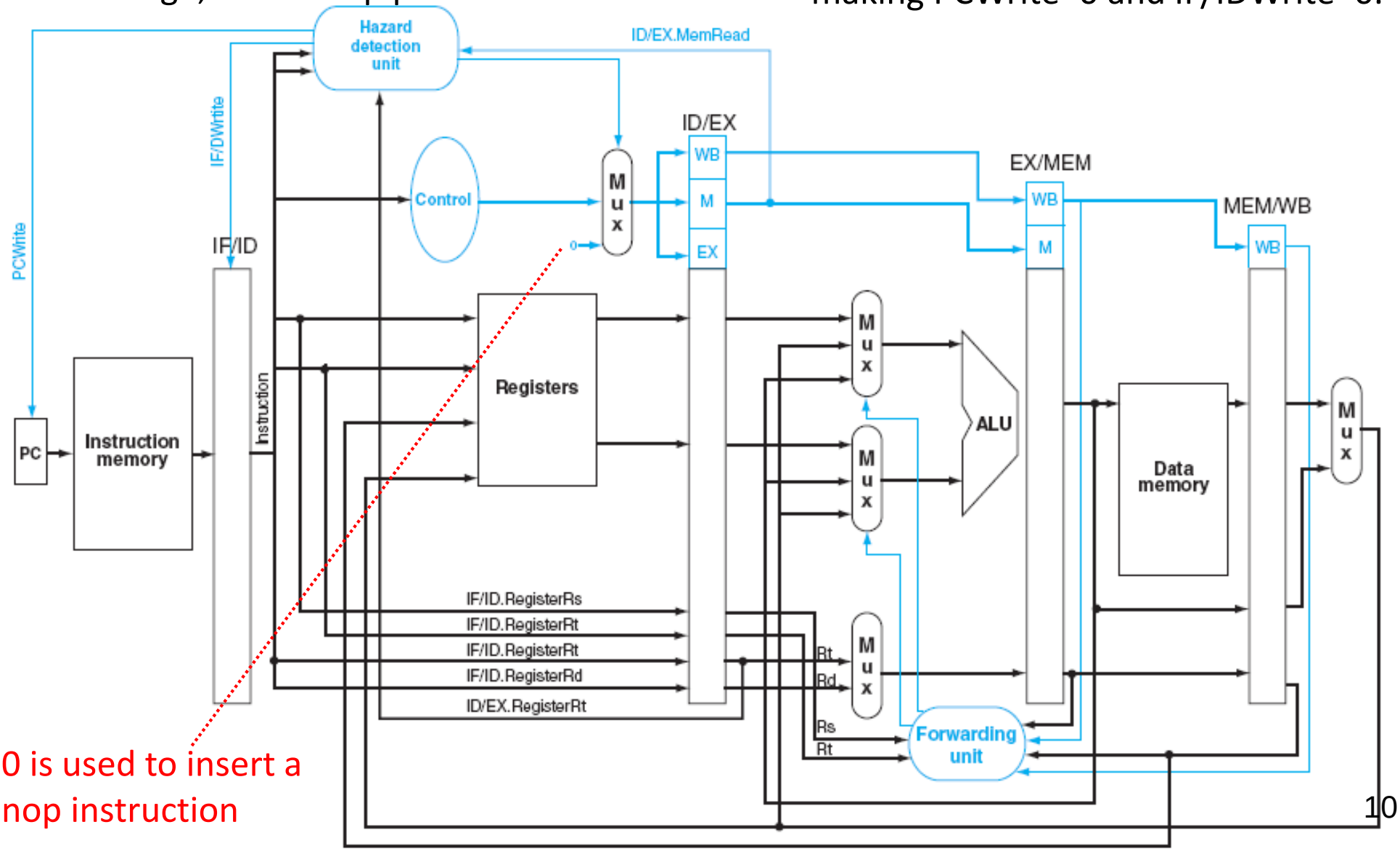
# ‘lw’ followed by R-type with data dependency



# Pipelined Datapath: Hazard Detection Unit

The hazard detection unit checks if there is a load in the EX stage that interferes with an operand in the ID stage; stalls the pipeline if so.

The pipeline is stalled for 1 clock cycle by making PCWrite=0 and IF/IDWrite=0.




0 is used to insert a nop instruction

# Branch/Control Hazard

- Happens when the ‘beq’ enters the pipeline: We don’t know which instruction should be executed next
  - It’s either the one at PC+4 (branch not taken)
  - Or it’s the one at the branch address (branch taken)

- Predict branch untaken:
  - We continue to fetch the next instructions
  - We know the branch decision at the moment when ‘beq’ enters the MEM stage
  - Assuming that ‘beq’ was at address X, when ‘beq’ is in the MEM stage, the instruction at X+4 is in the EX stage, the instruction at X+8 is in the ID stage and the instruction at X+12 is in the IF stage
  - If the branch should be taken, we flush these 3 instructions from the IF, ID and EX stages
- Con: Costs 3 CC if taken!

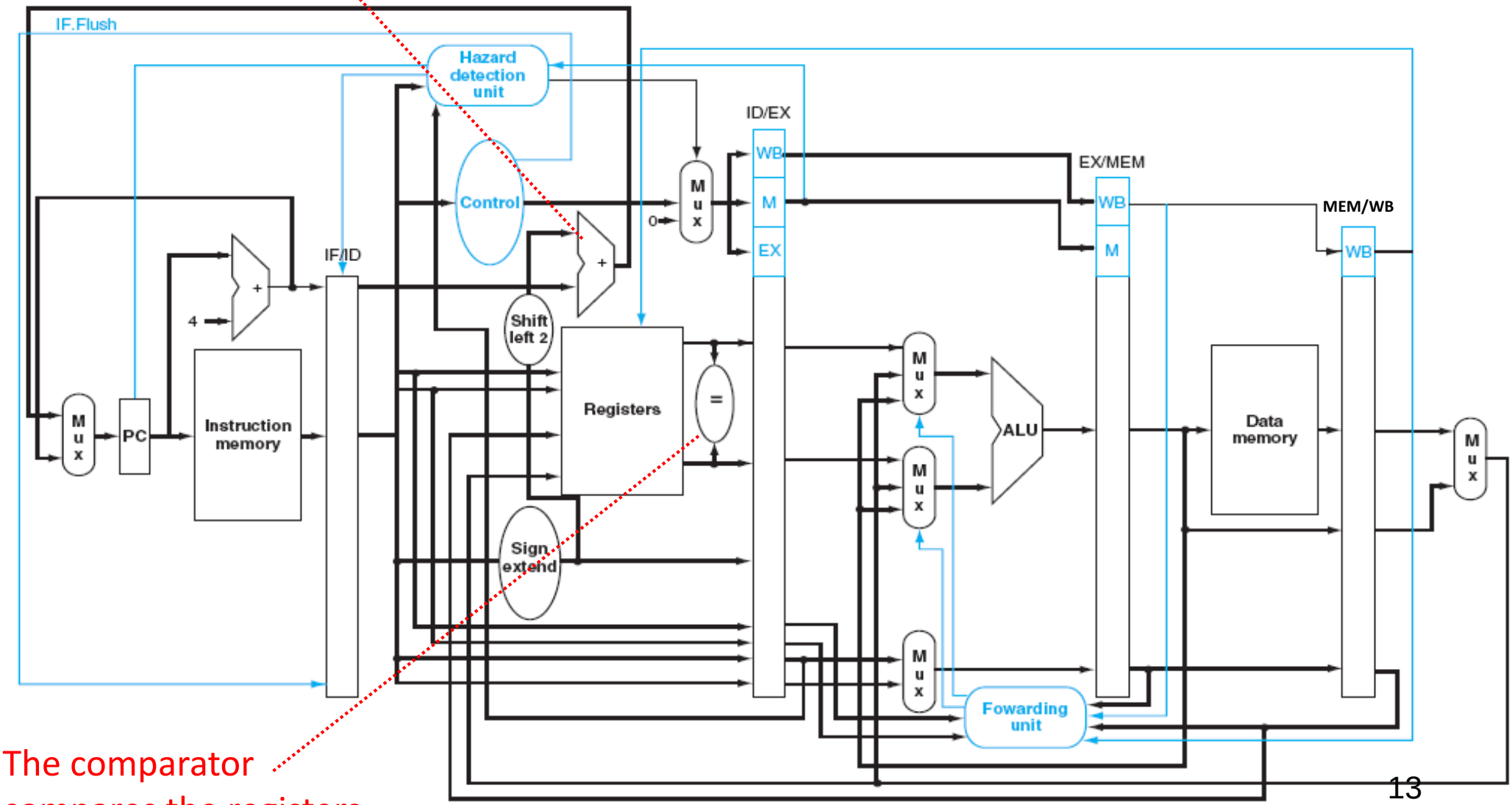
# Another way to deal with branch

- Do the branch in the ID stage:
  - The registers are read in the ID stage; we add a comparator hardware to know if they are equal
  - The comparator XOR the registers bit-by-bit, we get zero if they are equal
  - If the branch is taken, we only need to flush the instruction in the IF stage
- Con:
  - Requires a data forwarding unit in the ID stage in case ‘beq’ had dependency on preceding instructions
  - Also, we might need to stall the pipeline if a preceding instruction will produce a result that’s not available yet (instruction in EX is R-type or load)

# Executing the Branch in ID Stage

The branch address is found

This diagram is not complete: A forwarding unit is needed in the ID stage; it forwards from MEM stage




The comparator compares the registers

# Forwarding Cases For Doing ‘beq’ in ID Stage

- How should we forward to the ID stage?
- There are three possibilities: EX→ID, MEM→ID, WB→ID

- For the case below, ‘add’ is in EX stage and hasn’t computed t0 yet, therefore, it can’t be forwarded
- A ‘nop’ is needed and the forwarding could be done from MEM→ID

```
add     t0, t1, t2      // in EX stage (t0 not ready)
beq     t0, t1          // in ID stage
```

- The code with nop is shown below and MEM→ID forwarding is done

```
add     t0, t1, t2      // in MEM stage (t0 is ready)
nop
beq     t0, t1          // in ID stage
```

# Forwarding Cases For Doing ‘beq’ in ID Stage

- In the case below, when ‘beq’ is in ID stage, the ‘add’ is in MEM stage and has t0
- Therefore, t0 can be forwarded from MEM→ID

```
add     t0, t1, t2      // in MEM stage (t0 is ready)
or      t5, t6, t7
beq     t0, t1          // in ID stage
```

- The case below doesn’t need forwarding since t0 is written by ‘add’ in the 1st half of the cycle and then t0 is read by ‘beq’ in the 2nd half of the cycle

```
add     t0, t1, t2      // in WB stage
or      t5, t6, t7
and     t4, t6, t7
beq     t0, t1          // in ID stage
```

# Forwarding Cases For Doing ‘beq’ in ID Stage

- In the code below, t0 can’t be forwarded since t0 is not ready

```
lw      t0, 0(s0)       // in EX stage (t0 not ready)
beq     t0, t1          // in ID stage
```

- In the code below, t0 can’t be forwarded since it’s not ready

```
lw      t0, 0(s0)       // in MEM stage (t0 not ready)
or      t5, t6, t7
beq     t0, t1          // in ID stage
```

- In the code below, the ‘lw’ write t0 in the 1st half of the cycle and the ‘beq’ reads t0 in the 2nd half of the cycle

```
lw      t0, 0(s0)       // in WB stage
and     t4, t5, t6
or      t5, t6, t7
beq     t0, t1          // in ID stage
```

# Forwarding Cases For Doing ‘beq’ in ID Stage

*Conclusion of these scenarios:*

**R-type followed by ‘beq’**

- Separate them by one instruction (or nop) and forward MEM→ID

```
and     t0, t5, t6
beq     t0, t1
```

**Load followed by ‘beq’**

- Separate by two instructions (or two nops) (or 1 instruction and 1 nop) and no forwarding is needed

```
lw      t0, 0(s0)
beq     t0, t1
```

# Control Hazards: Compiler Help

- Branch Prediction
  - What if the compiler arranges the code such that most branches are not taken?

**Example** (assume the loop iterates many times)

- Assuming 'predict branch untaken', which code is better?
  - Code #1 is better because during the iterations, the beq doesn't branch
  - In Code #2, there's a nop penalty for each iteration of the loop

**Code #1**

```
Loop:  beq   t0, t1, Exit
       …
       …
       …
       j     Loop
Exit:
```

**Code #2**

```
Loop:  …
       …
       …
       …
       beq   t0, t1, Loop
```

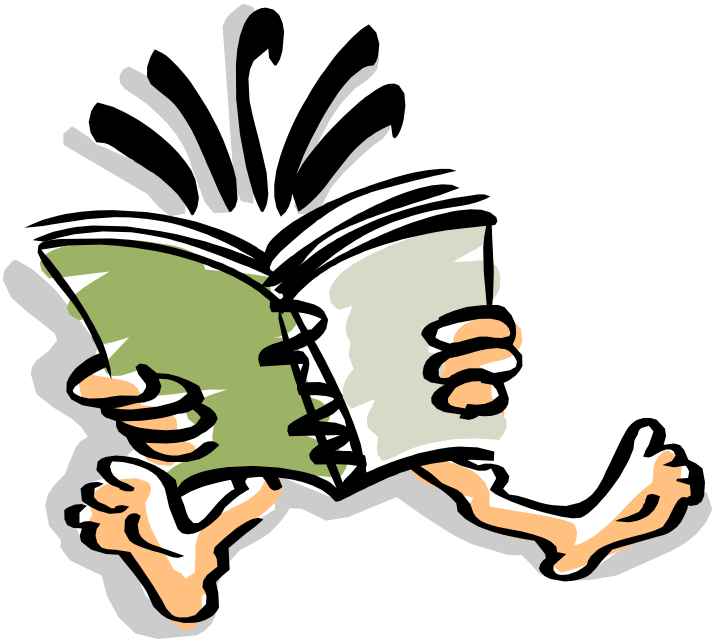
# Control Hazards: Compiler Help

- The combo of (compiler-based prediction) and (predict branch untaken) are suitable for a simple pipelines

- More sophisticated pipelines are:
  - **deeper** (large number of stages, e.g.: 17-stage pipeline)
  - **multiple-issue** (it's like the equivalent of two or more pipelines side-by-side) -- Instruction Level Parallelism (ILP)

- For complex pipelines, the **branch penalty is many cycles** and better prediction approach is needed

# Readings



- H&P COD
  - Chapter 4