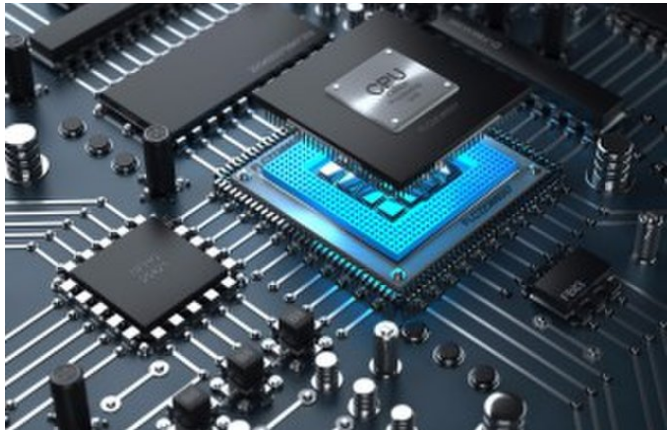


Pipelining 2

CS 154: Computer Architecture

Lecture #15

Winter 2020

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB

Administrative

- Talk is on **MONDAY, MARCH 9th** in our usual class
  - Will take attendance…

- Final Exam info:
  - **Tuesday, March 17th at 12:00** (**not 12:30!!!**) in this classroom
    - Arrive 10 mins early – randomized seating…
  - Cumulative Exam
  - Will allow some notes – exact details to follow
  - Study guide/example Qs will be issued by this weekend

Re: Labs

- Lab 7 still due on Sunday
- Lab 8 will be issued soon
- There IS a lab THIS Friday
- Re: lab NEXT Friday...

Lecture Outline

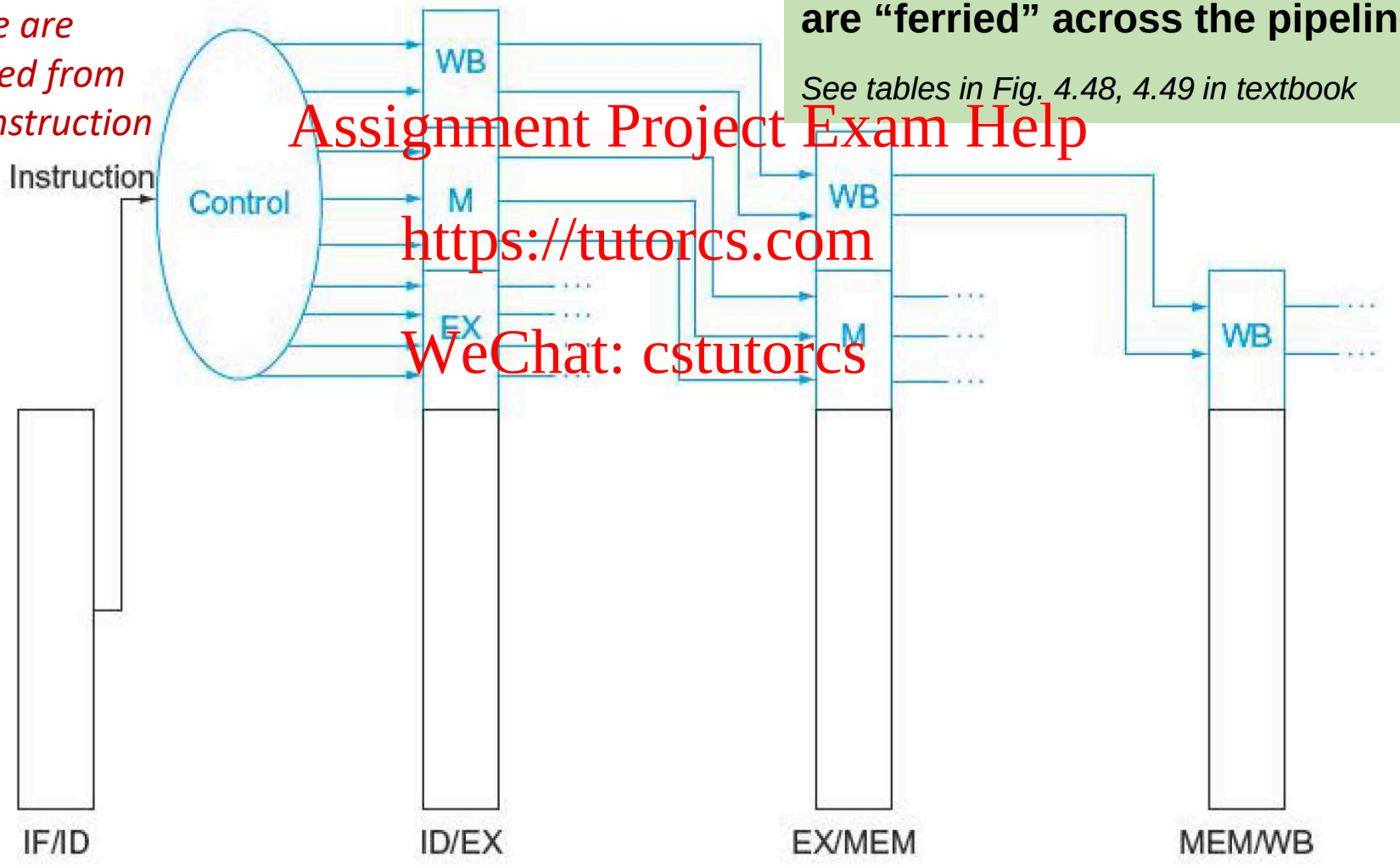
- Data Hazards in Pipelining: Forwarding vs Stalling
- Control Hazards: Branch Prediction

Control Lines for the Last 3 Pipeline Stages

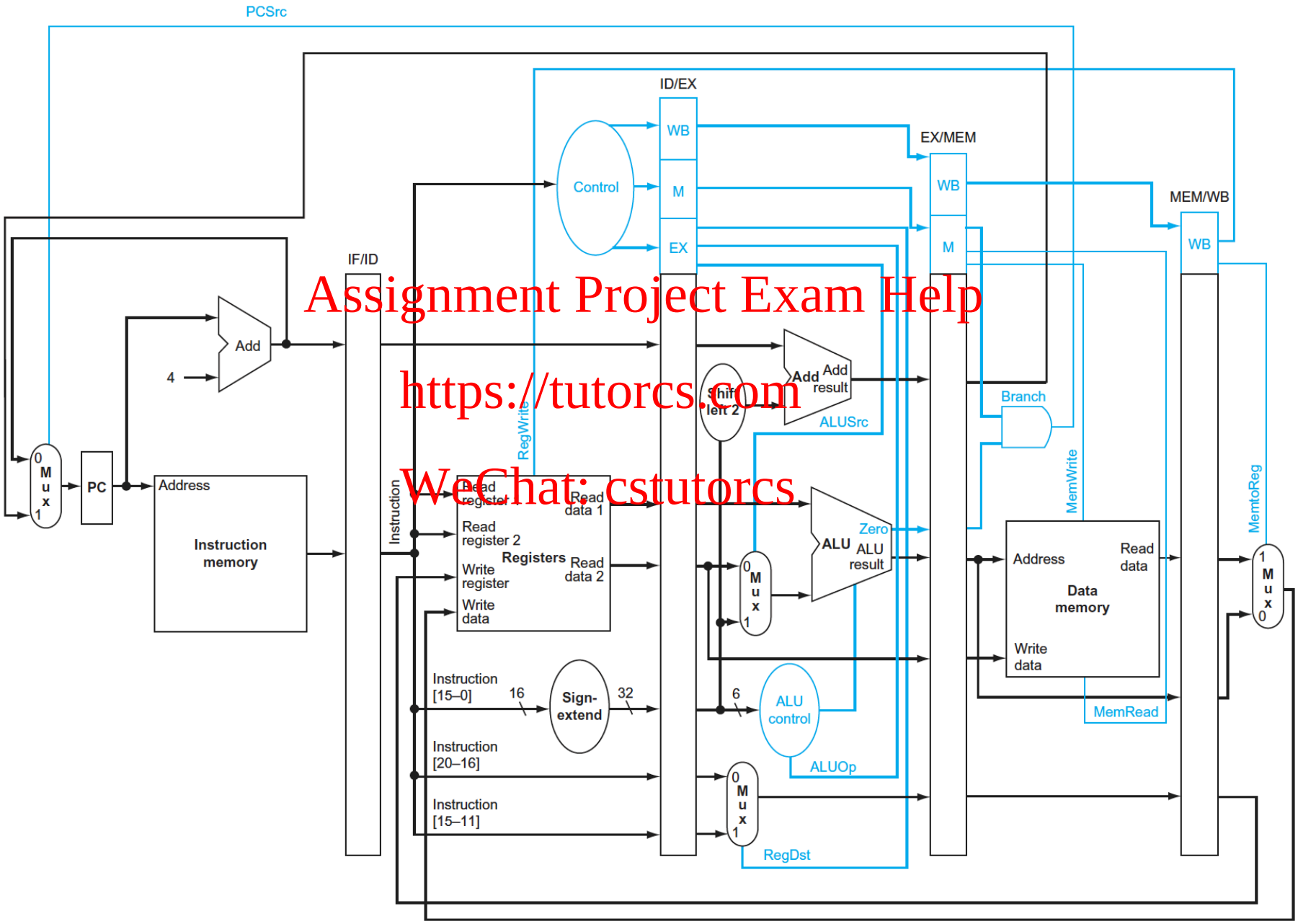
Same control signals that we learned earlier, but this time they are "ferried" across the pipelines

See tables in Fig. 4.48, 4.49 in textbook

These are derived from the instruction



Pipelined Datapath Showing Control Signals



Another Look at Data Hazards

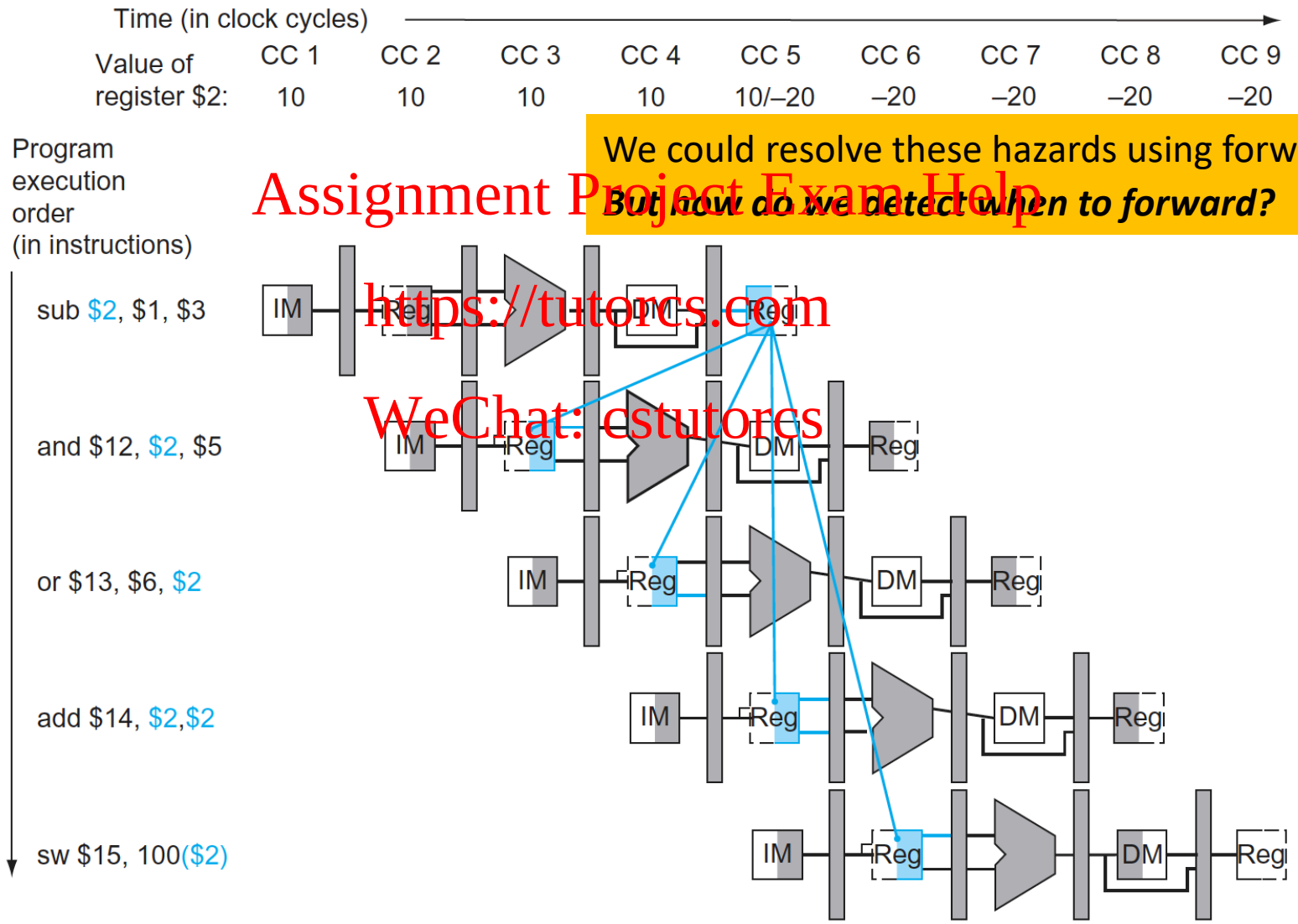
- Consider this sequence:

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```


- All of the instructions after **sub** are dependent on the result in register **$2** of the first instruction.

Let's say that reg. $2 has value *10* at first, but that the sub instruction changes that to *-20*.

This diagram shows that the instructions that would get the correct value of *−20* are **add** and **sw**, while the **AND** and **OR** instructions would get the incorrect value 10!

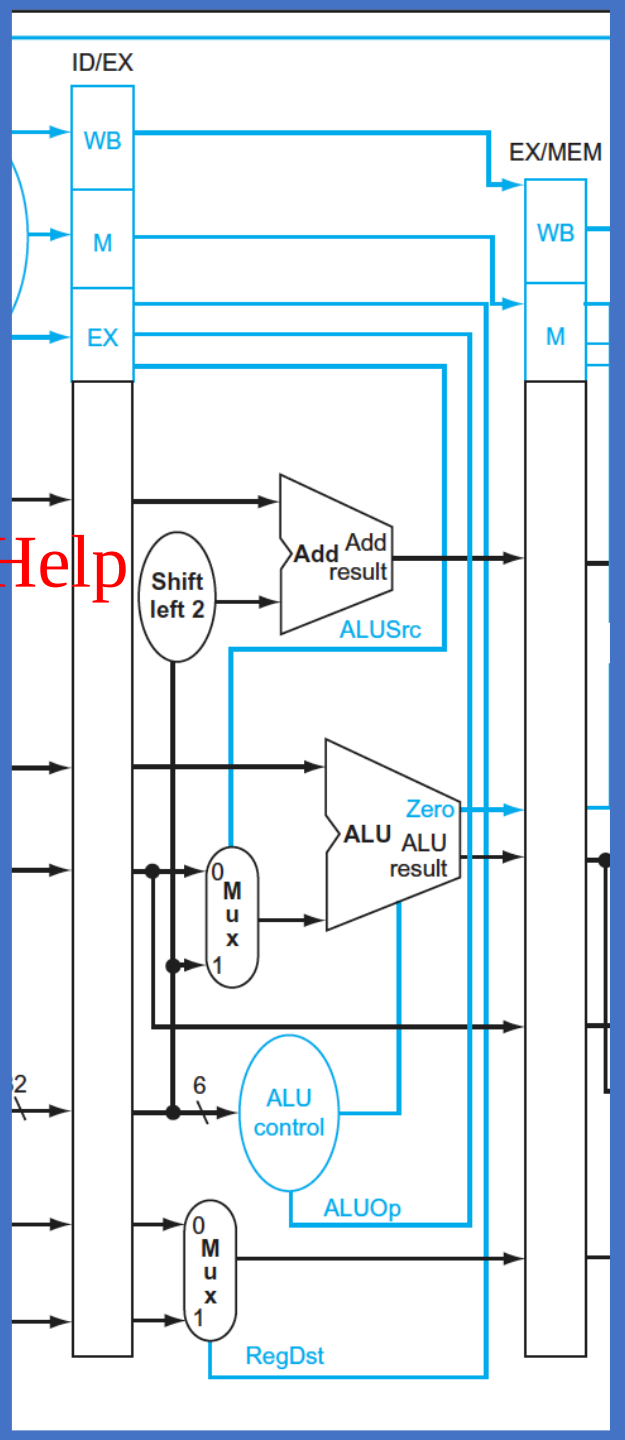


We could resolve these hazards using forwarding

But how do we detect when to forward?

Forwarding vs. Stalling

- We could stall using bubbles… (inefficient)
- …or we could ***forward*** the data as soon as it is available to any units that need it before it is available to read in the **WB** stage
- Let's only consider forwarding to an operation in the **EX** stage
  - That is, either an ALU operation or an address calculation



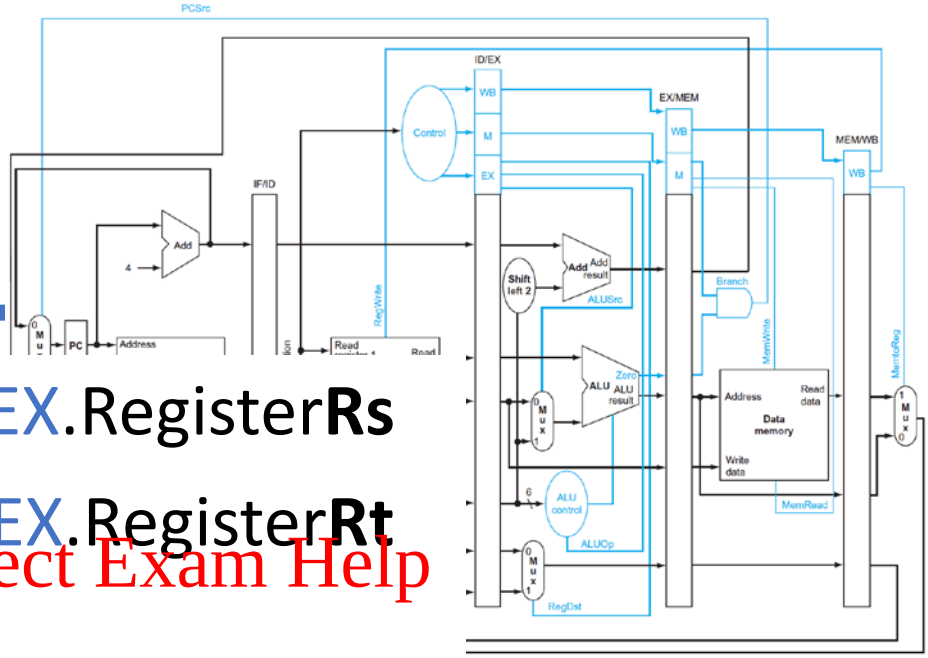
A Word on Some Notation to Use…

Example: **ID/EX.RegisterRs**

- Refers to the number of a register whose value is found in the pipeline register **ID/EX**
  - The 1st part is the name of the pipeline register (**ID/EX**)
  - the 2nd part is the name of the field in that register (**Rs**)

- ALU operand register numbers in EX stage are given by **ID/EX.RegisterRs** and **ID/EX.RegisterRt**

Data Hazards Occur When…



1a. EX/MEM.Register**Rd** = ID/EX.Register**Rs**

1b. EX/MEM.Register**Rd** = ID/EX.Register**Rt**

2a. MEM/WB.Register**Rd** = ID/EX.Register**Rs**

2b. MEM/WB.Register**Rd** = ID/EX.Register**Rt**

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

1st hazard here is on register $2, between the result of **sub** and **AND** instructions.

This hazard can be detected when the **AND** instruction is in the **EX** stage and the **sub** instruction is in the **MEM** stage, so this is ***hazard 1a:***
EX/MEM.RegisterRd = ID/EX.RegisterRs = $2

Detecting the Need to Forward

- These comparisons are **not enough**, though!
  - Some instructions don't write to registers

- Solution: see if the **RegWrite** control signal will be active

- Additionally, check that:

1a. EX/MEM.Register**Rd** = ID/EX.Register**Rs** ≠ **$zero**

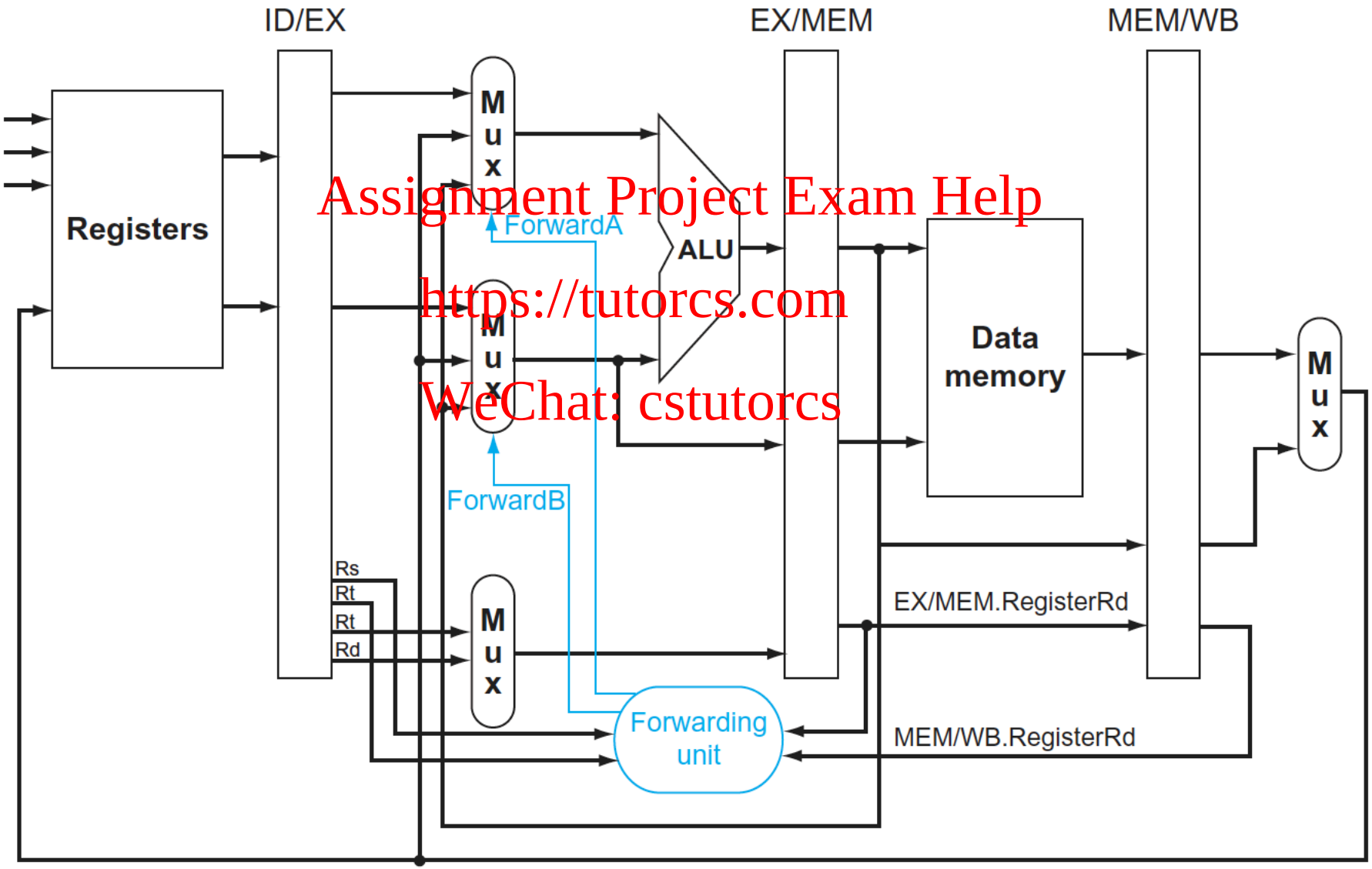
1b. EX/MEM.Register**Rd** = ID/EX.Register**Rt** ≠ **$zero**

2a. MEM/WB.Register**Rd** = ID/EX.Register**Rs** ≠ **$zero**

2b. MEM/WB.Register**Rd** = ID/EX.Register**Rt** ≠ **$zero**

See **Fig. 4.55** in textbook for full explanation of the mux selects **ForwardA** and **ForwardB**

Forwarding Paths (simplified)



Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Double Data Hazards!

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

- Consider the sequence:

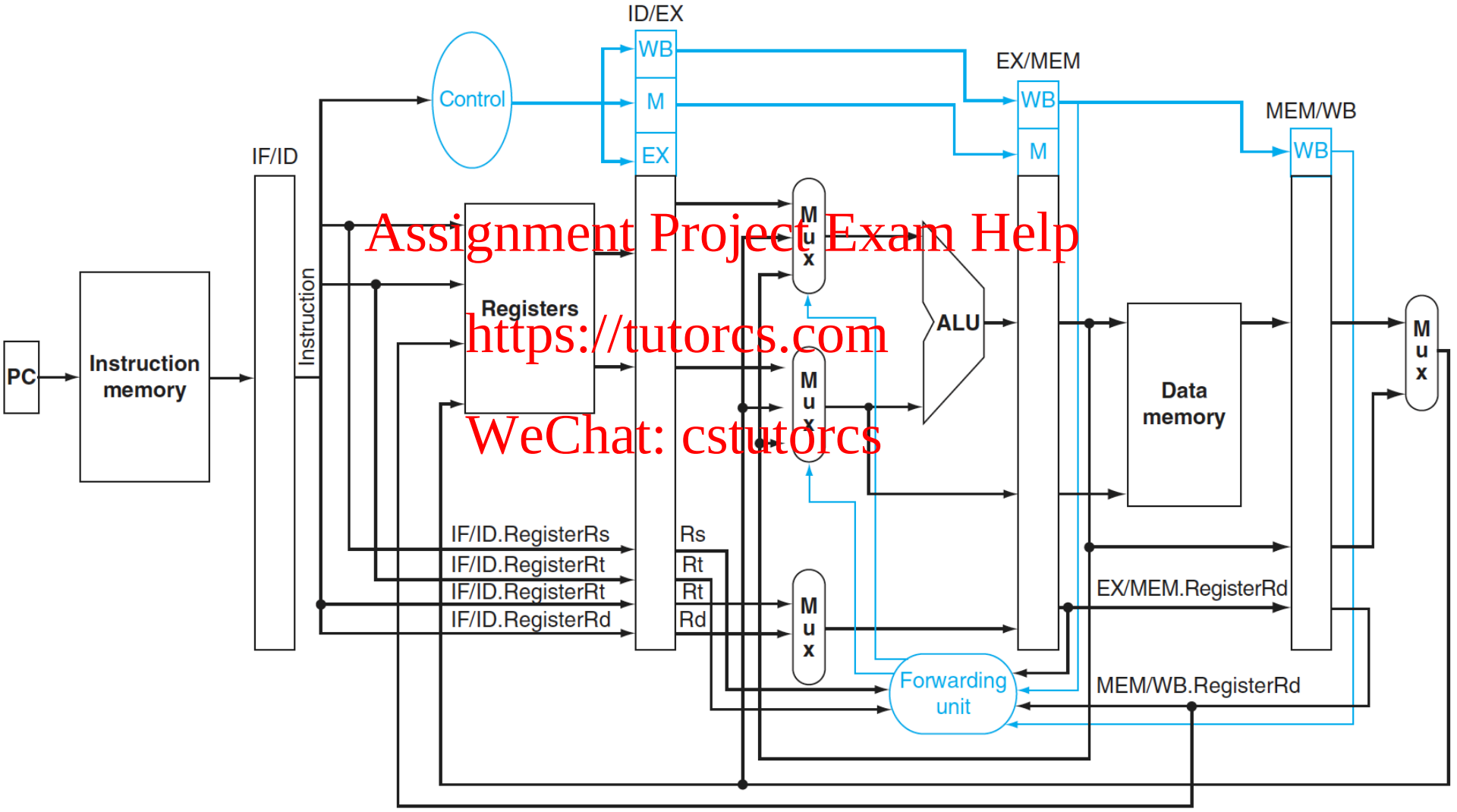
```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
```


- Both hazards conditions occur at once!
- We want to use the most recent value in **$1**
- We have to revise the **MEM** hazard condition
  - Only forward if **EX** hazard condition isn't true

Revised Forwarding Condition

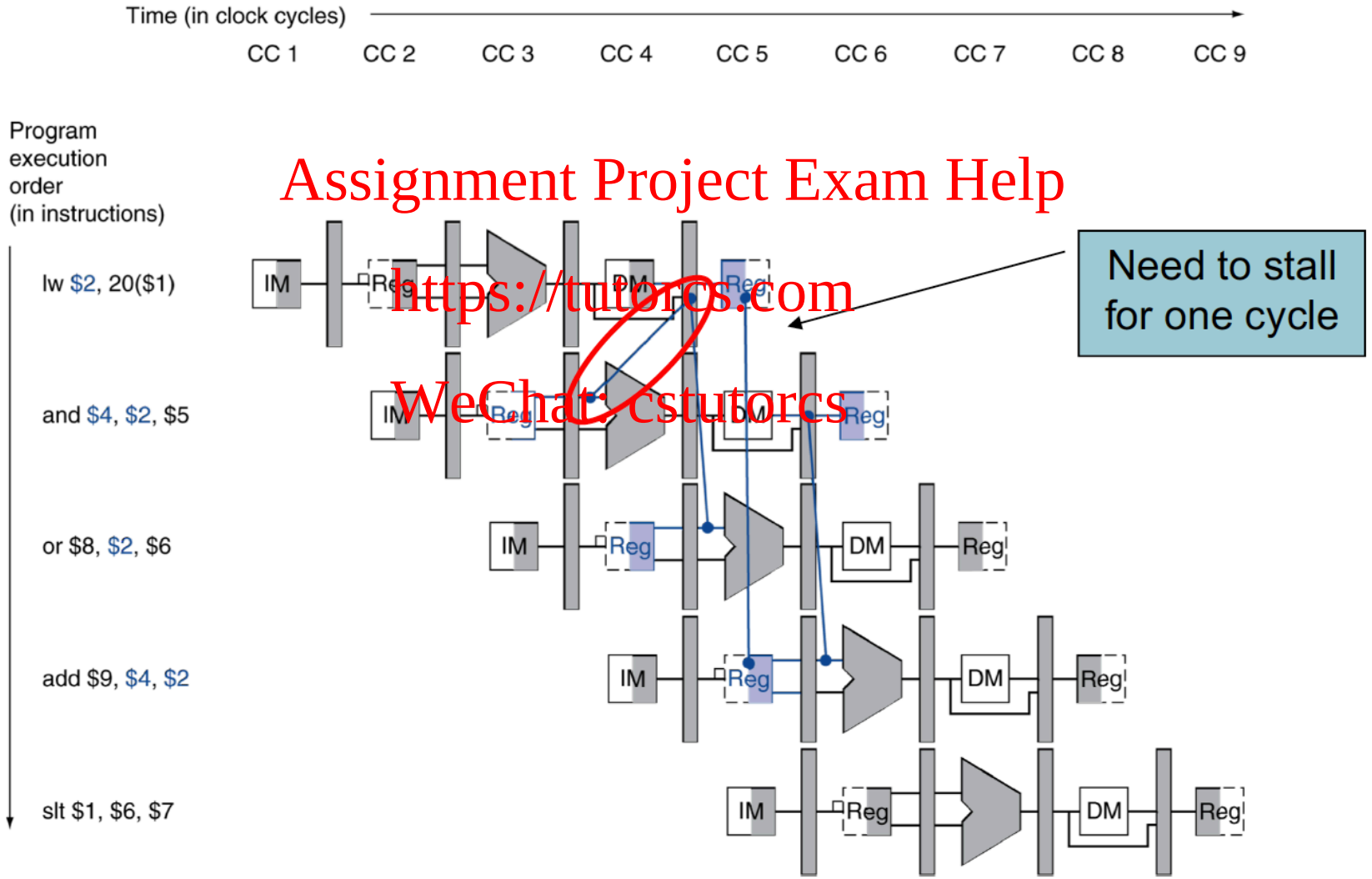
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Pipelined Datapath Modified To Resolve Hazards Via Forwarding



Load-Use Data Hazard

A case where forwarding will not work is when an instruction tries to read a register following a load instruction that writes the same register.



Hazard Detection

- We also need a “Hazard Detection Unit”!
- It operates during the **ID** stage so that it can insert the stall between the load and its use.
- ALU operand register numbers in ID stage are given by **IF/ID.RegisterRs, IF/ID.RegisterRt**

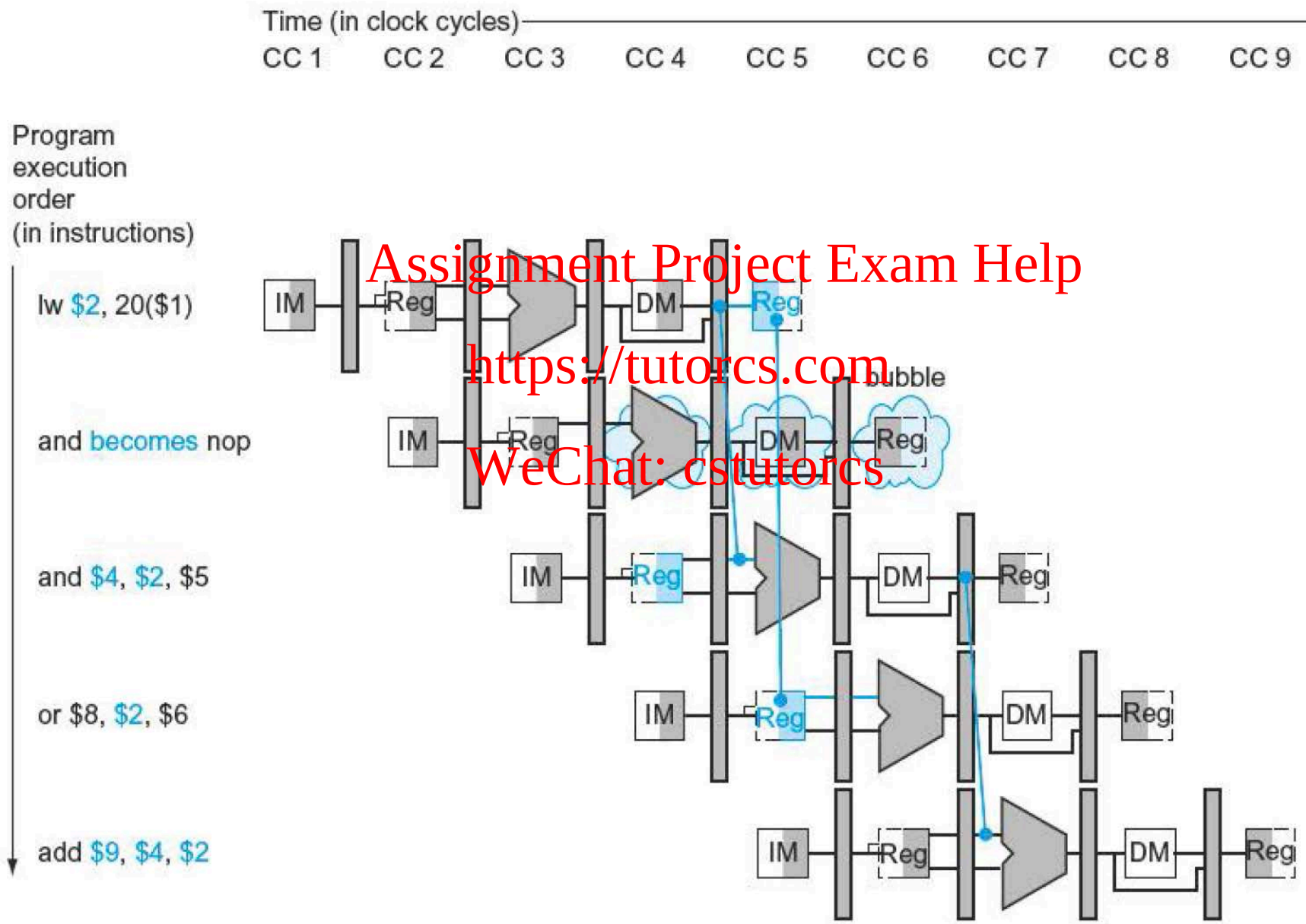
- Has one thing to check:

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

How to Stall the Pipeline

- Force control values in **ID/EX** register to 0
  - EX, MEM and WB do a **nop** (no-operation)
- Prevent update of **PC** and **IF/ID** register
  - Current instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows **MEM** to read data for **lw**
  - Can subsequently forward to **EX** stage

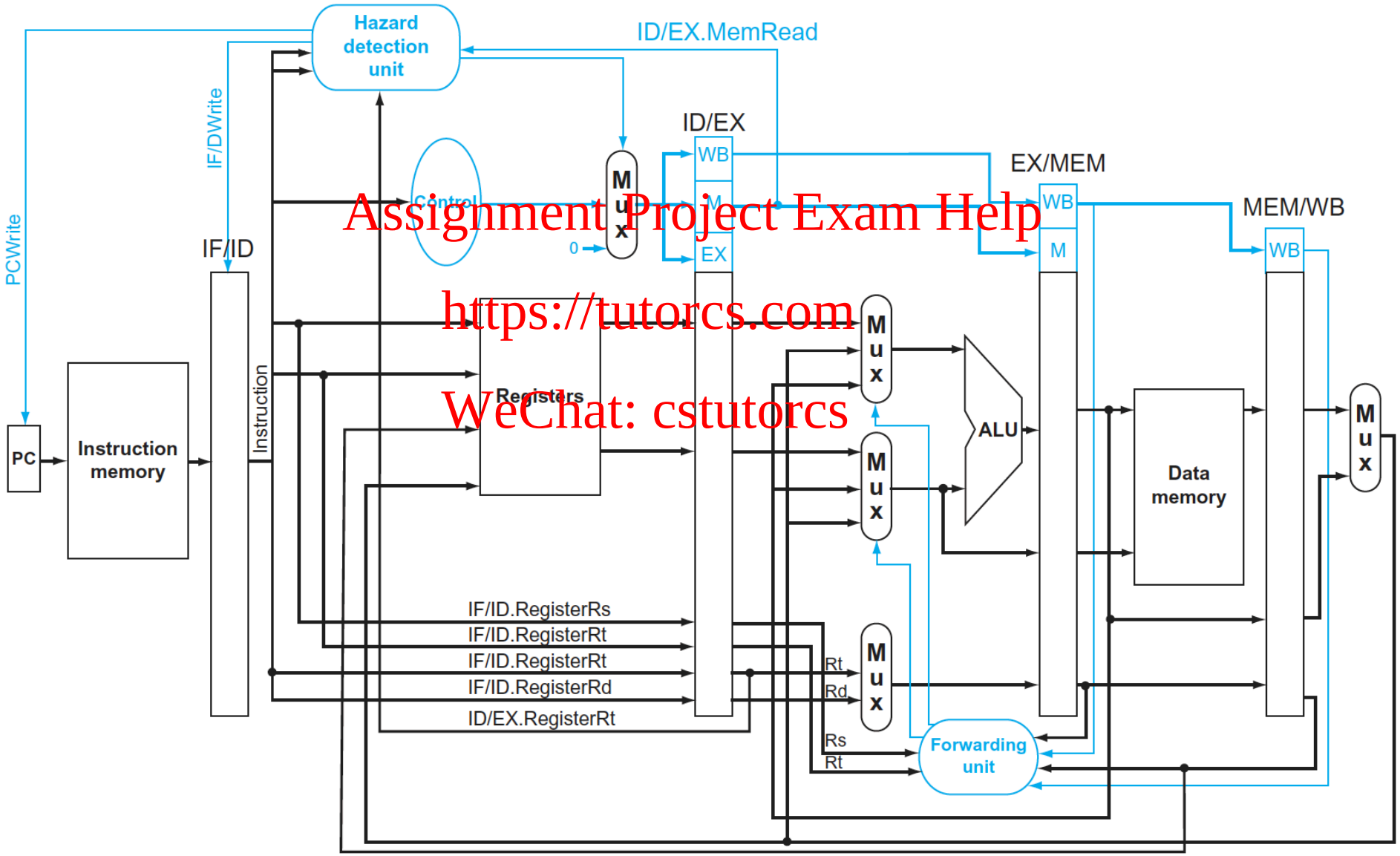
How Stalls Are Actually Done…



Compiler vs Hardware Stalls

- Compilers usually rely on the hardware to resolve hazards
  - Sometimes compilers take measures to avoid some types of hazards
- BUT a compiler *must understand the pipeline* to achieve the best performance.
  - Otherwise, unexpected stalls will reduce the performance of the compiled code.

Pipelined Datapath Modified
To Resolve Hazards Via Forwarding OR Stalling



Control Hazards

- Pipeline hazards involving branches

- An instruction must be fetched at every clock cycle to sustain the pipeline

  - Buy the decision about whether to branch doesn't occur until the **MEM** pipeline stage

  - Stalling until the branch is complete is too slow

- Control hazards occur less frequently than data hazards
  - We end up using simpler schemes.

Prediction Scheme 1: Assume branch **not** taken

- Continue execution down the sequential instruction stream.

- If the branch **is taken**, the instructions that are being fetched and decoded **must now be discarded** (flushed)
  - Execution continues at the branch target.

- If the branch **is untaken** half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards

Prediction Scheme 1.5: Reduce the Delays

- That is, reduce the cost of the taken branch
- *Main idea*: if we move the branch execution earlier in the pipeline (from **MEM**), then fewer instructions need be discarded.

- Many branches rely only on simple tests (equality or sign)
  - Such tests do not require a full ALU operation
  - Can be done with at most a few gates.

- For more complex branch decisions use a separate instruction that uses an ALU to perform a comparison

What Needs to be Done?

2 actions have to occur:

- Computing the branch target address earlier
  - Easy fix: we move the branch adder from the **EX** stage to the **ID** stage
- Evaluating the branch decision earlier
  - Harder to do…
  - For branch equal, we would compare the two registers read during the **ID** stage to see if they are equal.
    - Can be done with 1 XOR and 1 OR gate (32b gates).
  - Implies additional forwarding and hazard detection hardware…

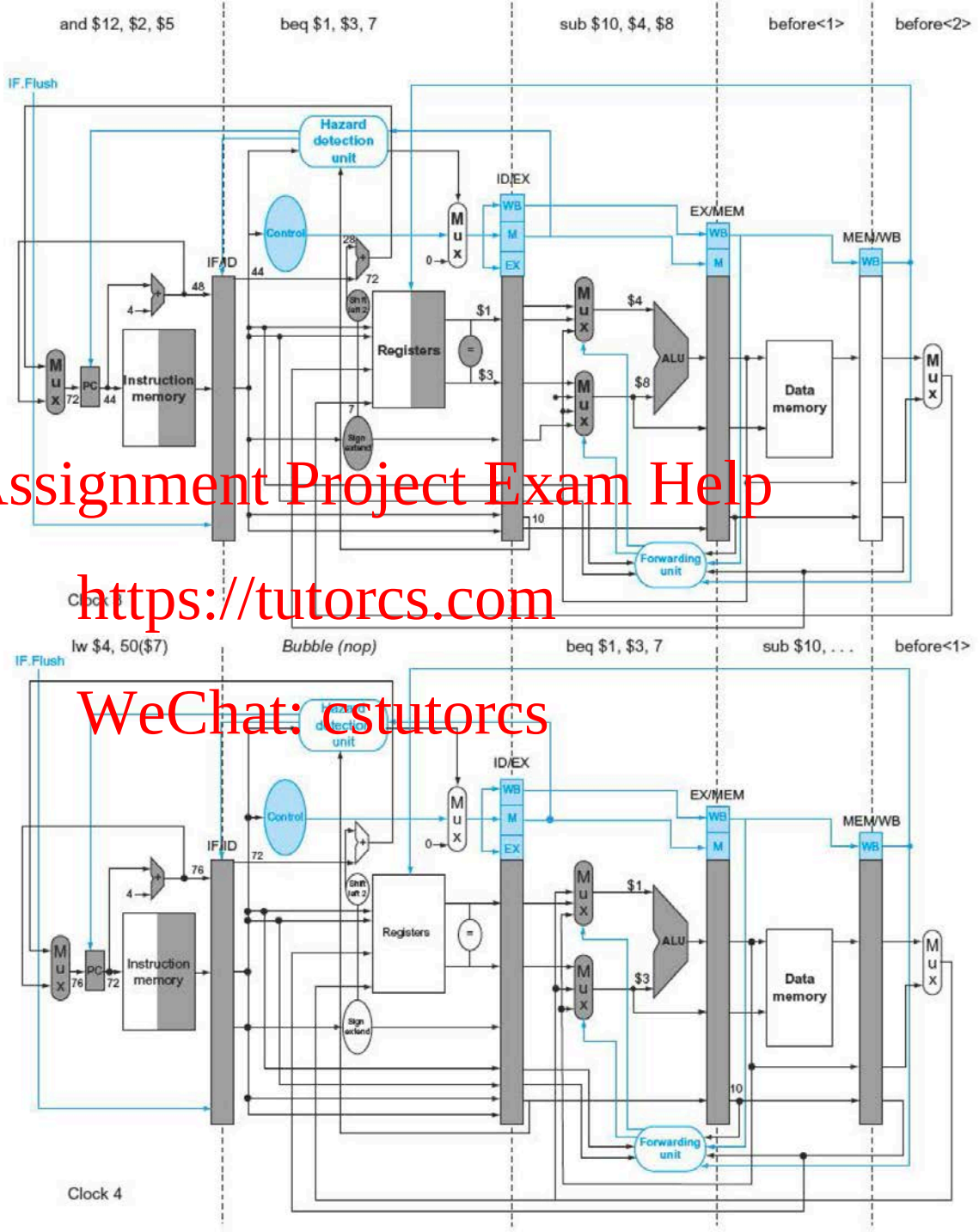
Example

- Consider this code:

```
36 sub $10, $4, $8
40 beq $1, $3, 7  # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```


- Assume that, in this example, the branch will be taken

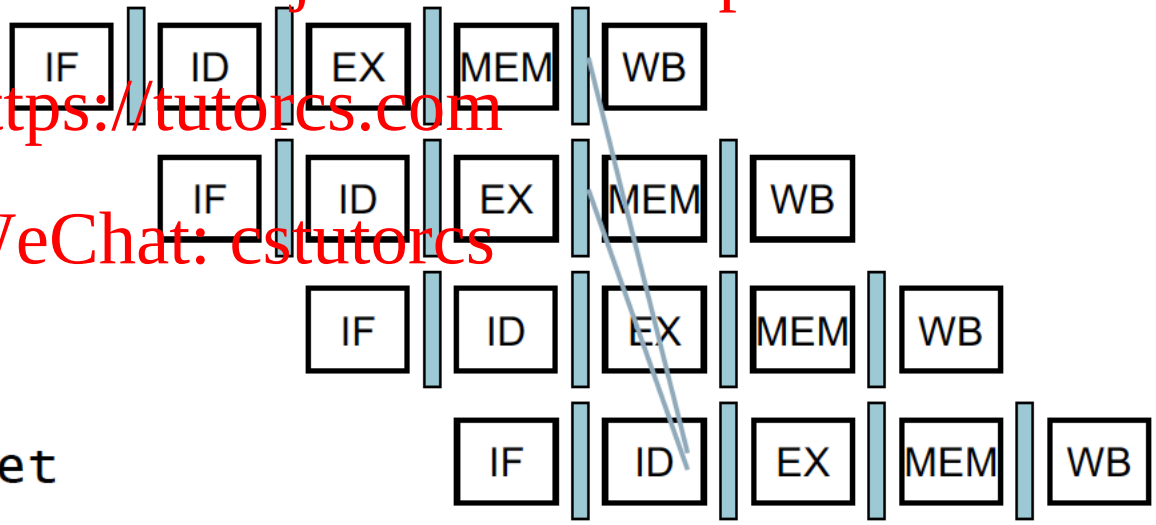
```
36 sub $10, $4, $8
40 beq $1, $3, 7 # I
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```



Using Simple Forwarding in Branches

- If a comparison register is a destination of a 2nd or 3rd preceding ALU instruction


```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```



- Can resolve using forwarding

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant, so dynamic prediction is used, needing:

- Branch prediction buffer (aka branch history table)
- Indexing by recent branch instruction addresses

- Store outcome (taken/not taken)
- To execute a branch, then:
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction

1-bit Predictor

- A **branch prediction buffer** is a small memory indexed by the lower portion of the address of the branch instruction.

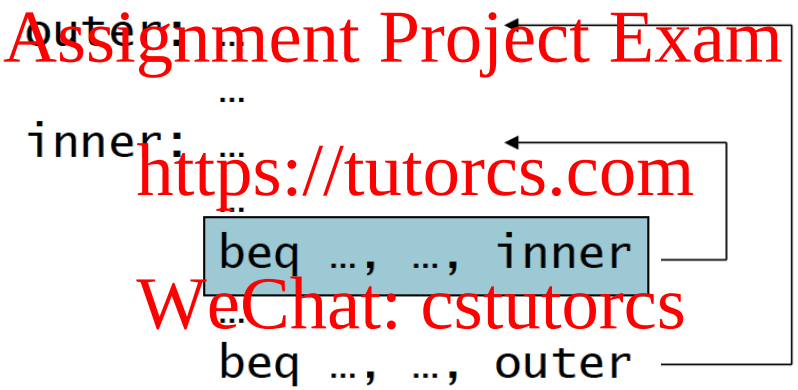
- The memory contains a bit that says whether the branch was recently taken or not.

  - Very simple
  - Remember: a prediction is just an "educated" guess – a hint we hope is correct

- If the hint turns out to be wrong:
  - the incorrectly predicted instructions are deleted
  - the prediction bit is inverted and stored back
  - the proper sequence is fetched and executed.

Downsides to a 1-bit Predictor

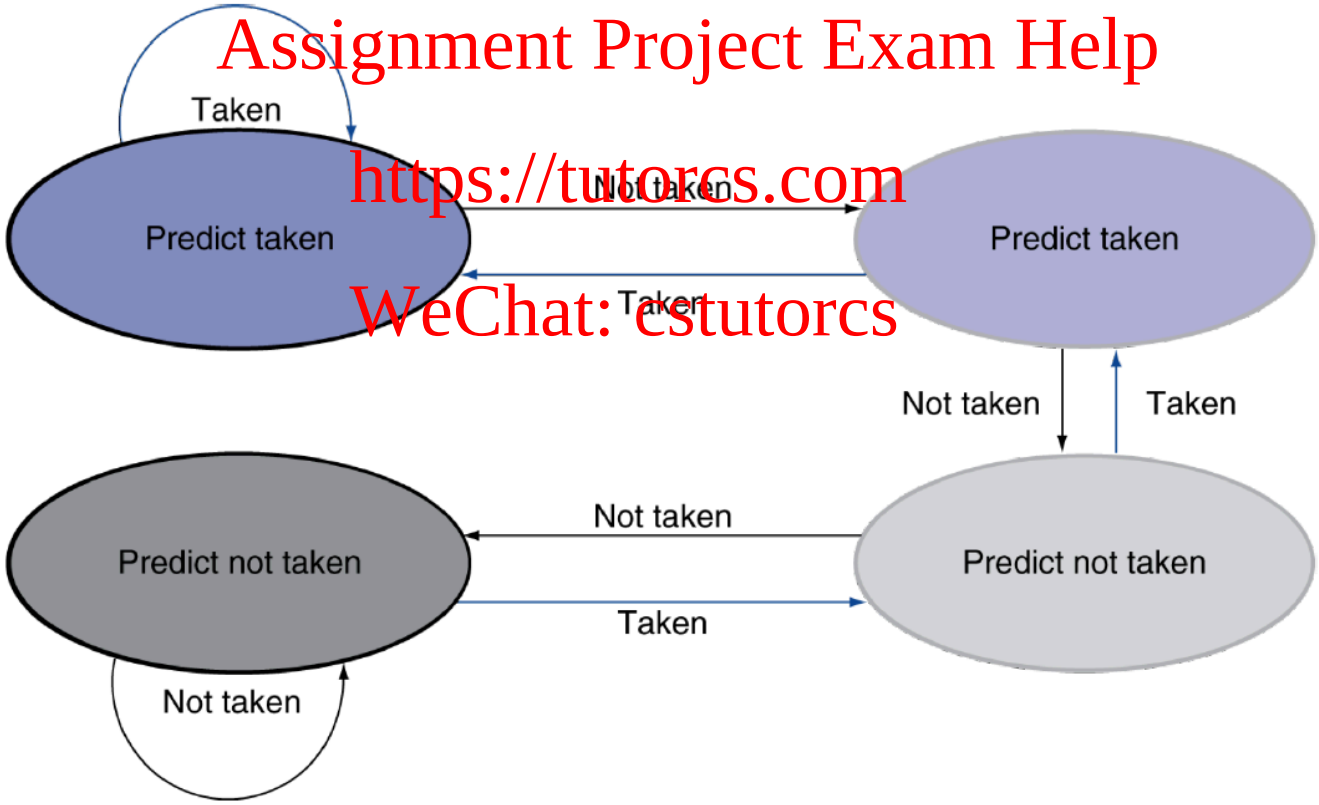
- You could get a sequence of wrong predictions!
- Example:



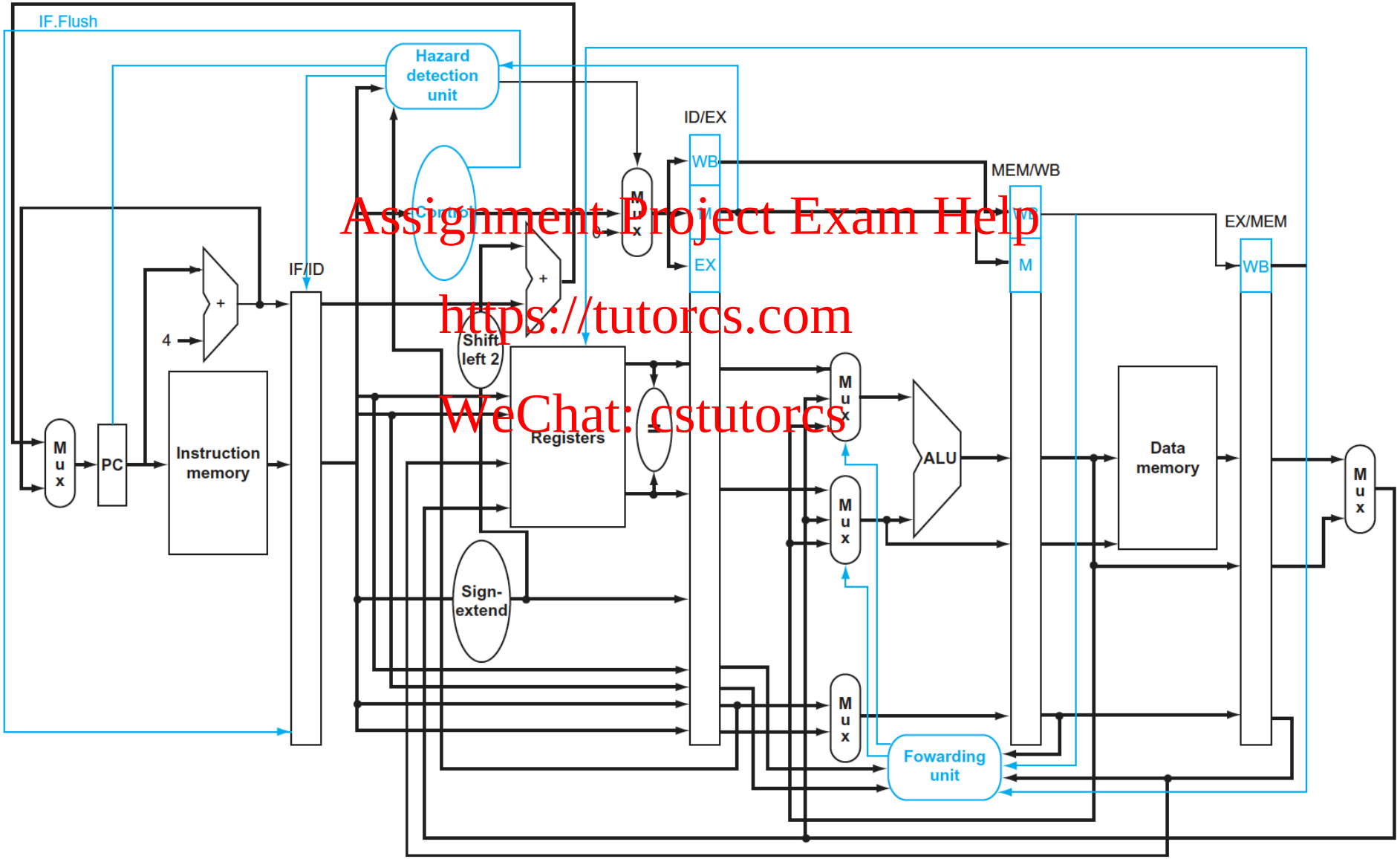
- Mis-predict taken on last iteration of the inner loop
- Then mis-predict (as in “do not take”) on first iteration of inner loop next time around

A Better Way? Use a **2-bit** Predictor

- Only change prediction on two successive mis-predictions



Final Pipelined Datapath
Showing Hazard and Forwarding Detection



YOUR TO-DOs for the Week

- Finish Lab 7 by Sunday

- New Lab 8 (last one!) will be issued Thursday

  - Due next week, which is last week of classes… ☹