Pipeline: Hazards

Fall, 2017

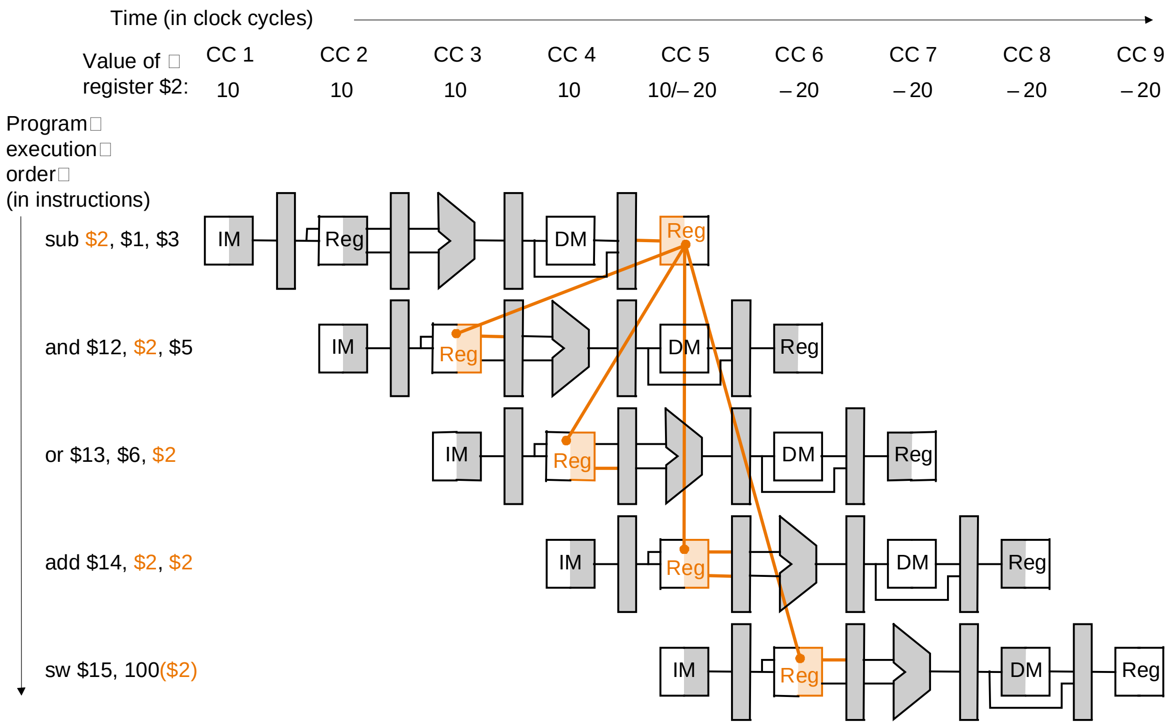
These slides are adapted from notes by Dr. David Patterson (UCB)

Announcements

- Midterm Exam is on 10/23/2017 in class.
- Midterm Review is on 10/18/2017.

Data Hazards

- Data hazards occur when data is used before it is stored



The use of the result of the SUB instruction in the next three instructions causes a data hazard, since the register is not written until after those instructions read it.

Data Hazards

Execution Order is:
Instr$_I$
Instr$_J$

Read After Write (RAW)

Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Data Hazards

Execution Order is:
Instr_I
Instr_J

Write After Read (WAR)

Instr_J tries to write operand *before* Instr_I reads i

– Gets wrong operand

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

– Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - –All instructions take 5 stages, and
  - – Reads are always in stage 2, and
  - – Writes are always in stage 5

Data Hazards

Execution Order is:
Instr_I
Instr_J

Write After Write (WAW)

Instr_J tries to write operand *before* Instr_I writes it

– Leaves wrong result (Instr_I not Instr_J)

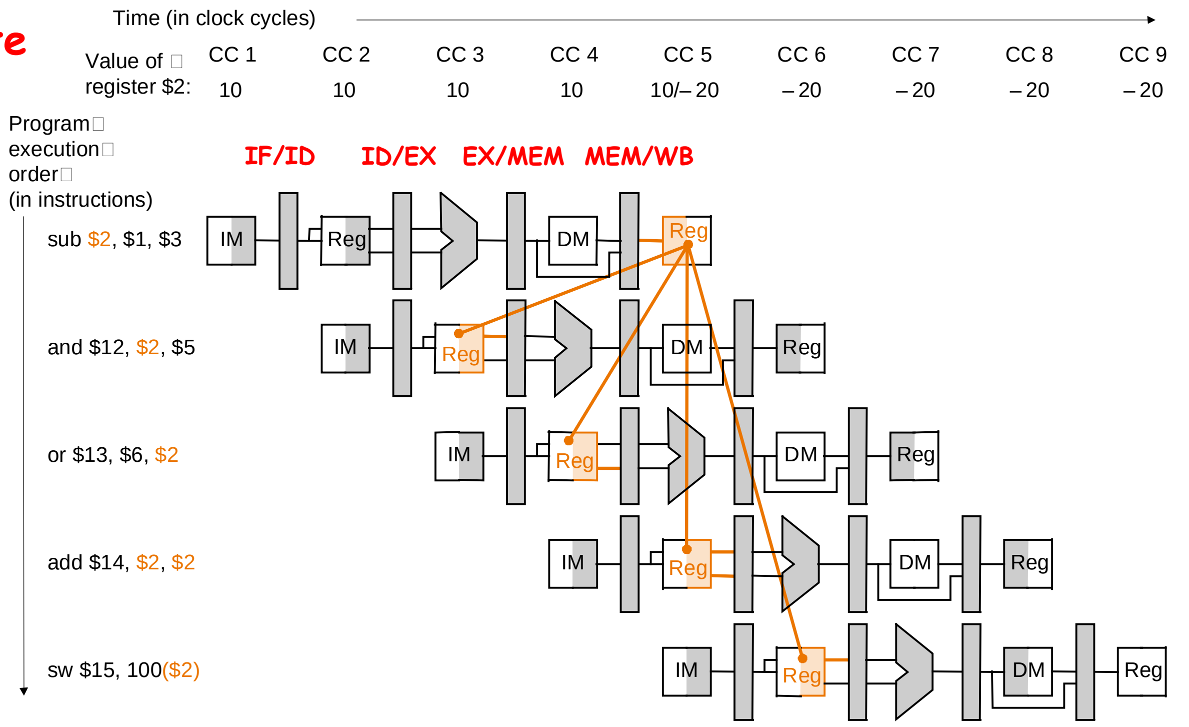
```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an “output dependence” by compiler writers
  This also results from the reuse of name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
  - –All instructions take 5 stages, and
  - – Writes are always in stage 5

•Will see WAR and WAW in later more complicated pipes

Data Hazard Detection in MIPS

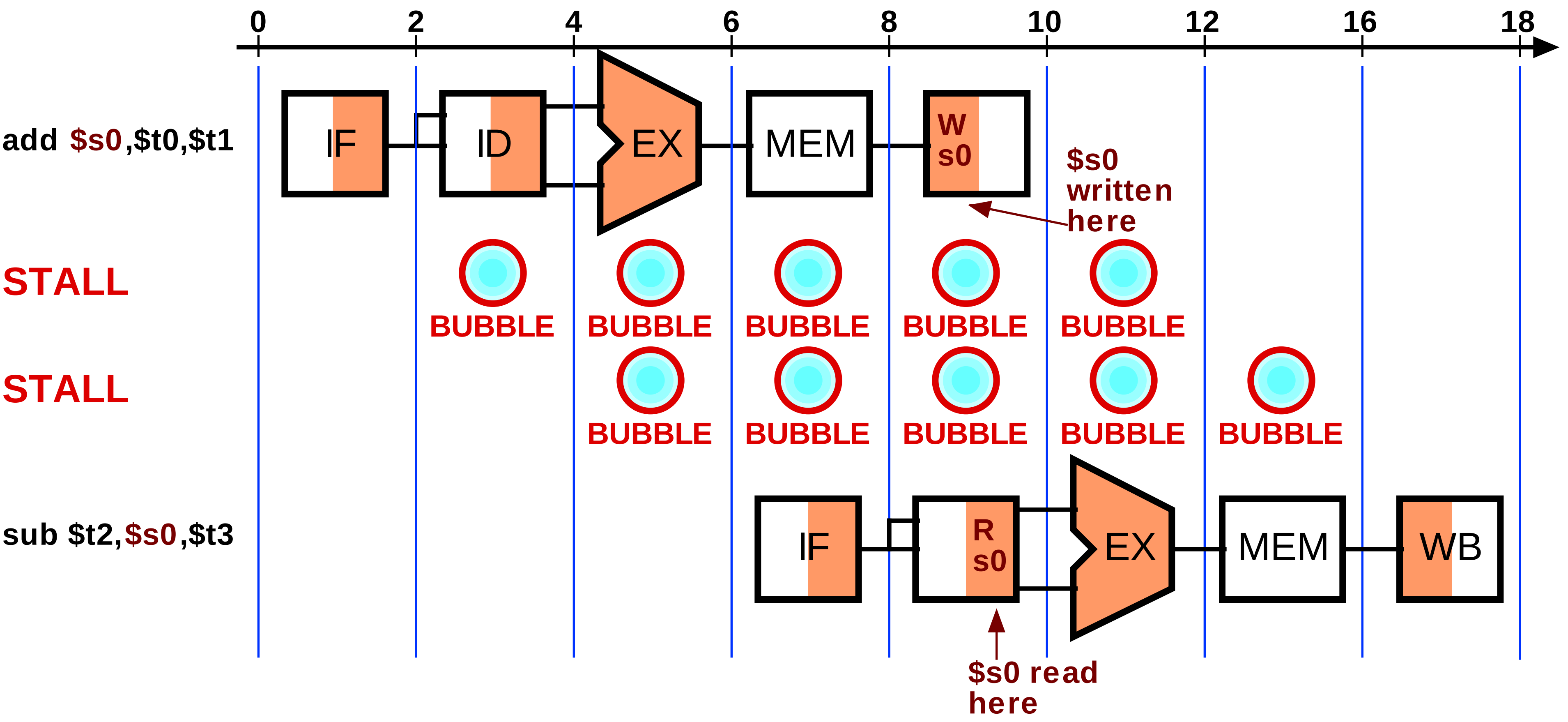
Read after Write



Data Hazards

- Solutions for Data Hazards
  - Stalling
  - Forwarding:
    - connect new value directly to next stage
  - Reordering

Data Hazard - Stalling



Data Hazards - Stalling

Simple Solution to RAW

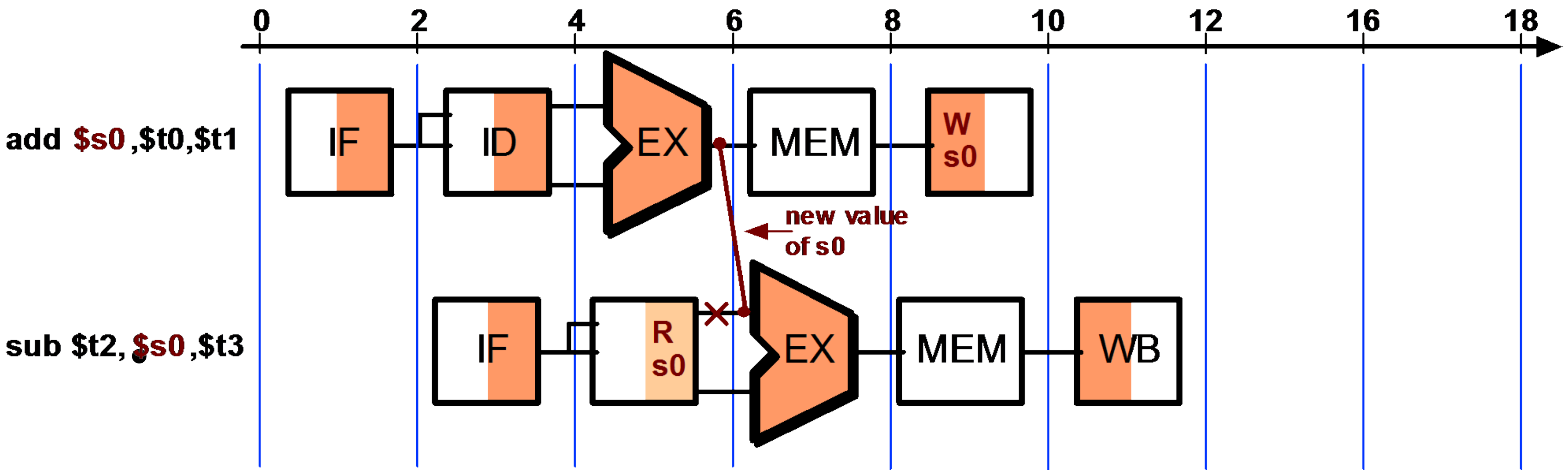
- Hardware detects RAW and stalls
- Assumes register written then read each cycle
  + low cost to implement, simple
  -- reduces IPC
- Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/shortcircuit (We will use the word “forward”)
- Use data before it is in the register
  + reduces/avoids stalls
  -- complex
- Crucial for common RAW hazards

Data Hazards - Forwarding

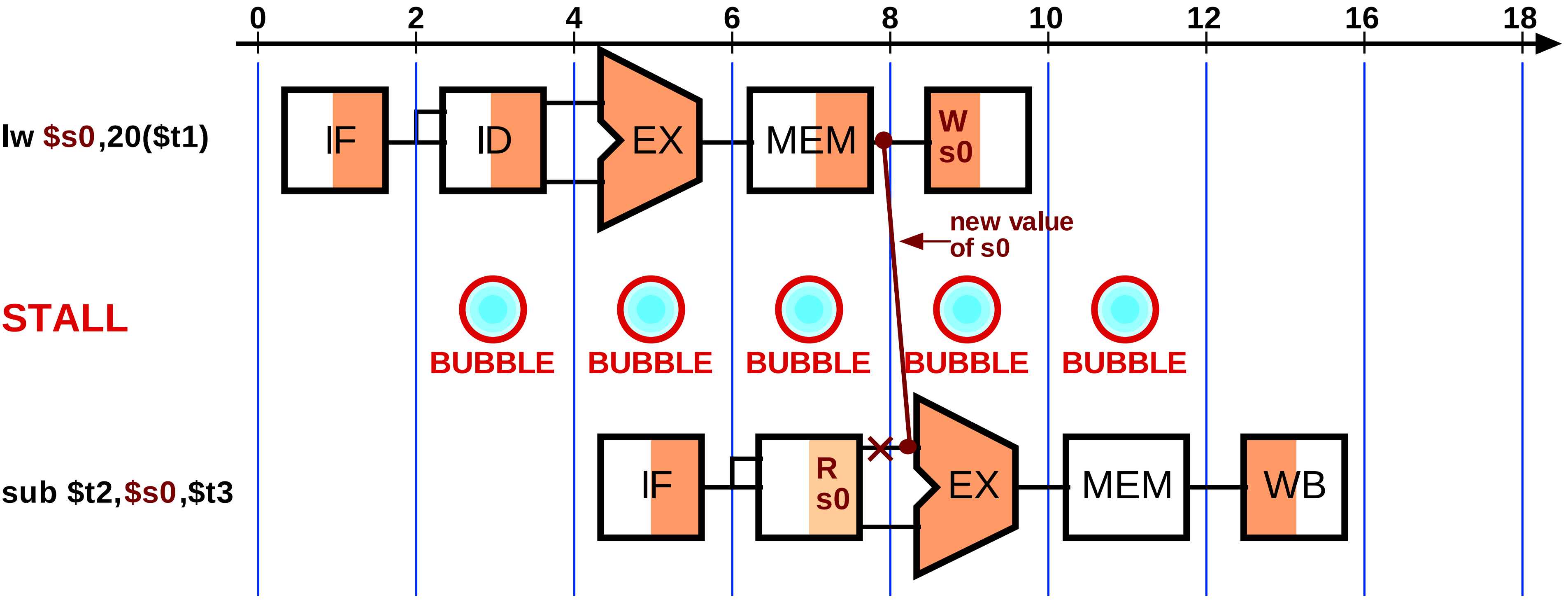
- Key idea: connect new value directly to next stage
- Still read s0, but ignore in favor of new result



Problem: what about load instructions?

Data Hazards - Forwarding

- STALL <u>still</u> required for load - data avail. after MEM
- MIPS architecture calls this <u>delayed load</u>, initial implementations required compiler to deal with this



Data Hazards

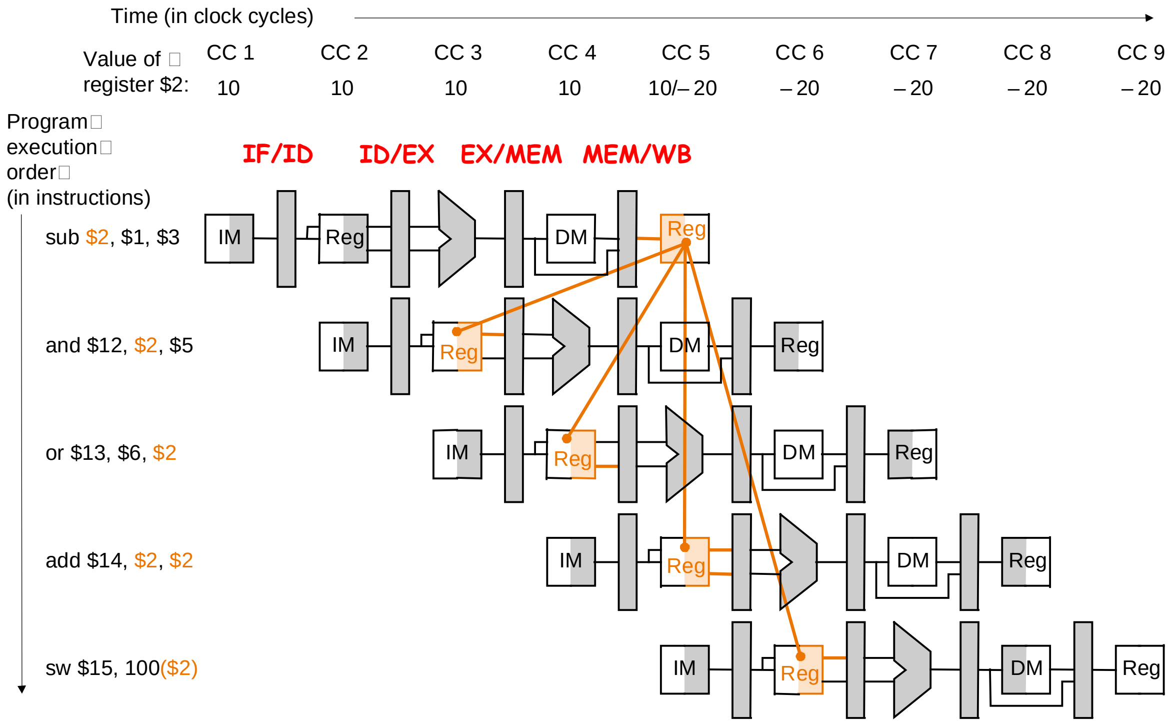
This is another representation of the stall.

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall?	IF	ID	EX	MEM	WB

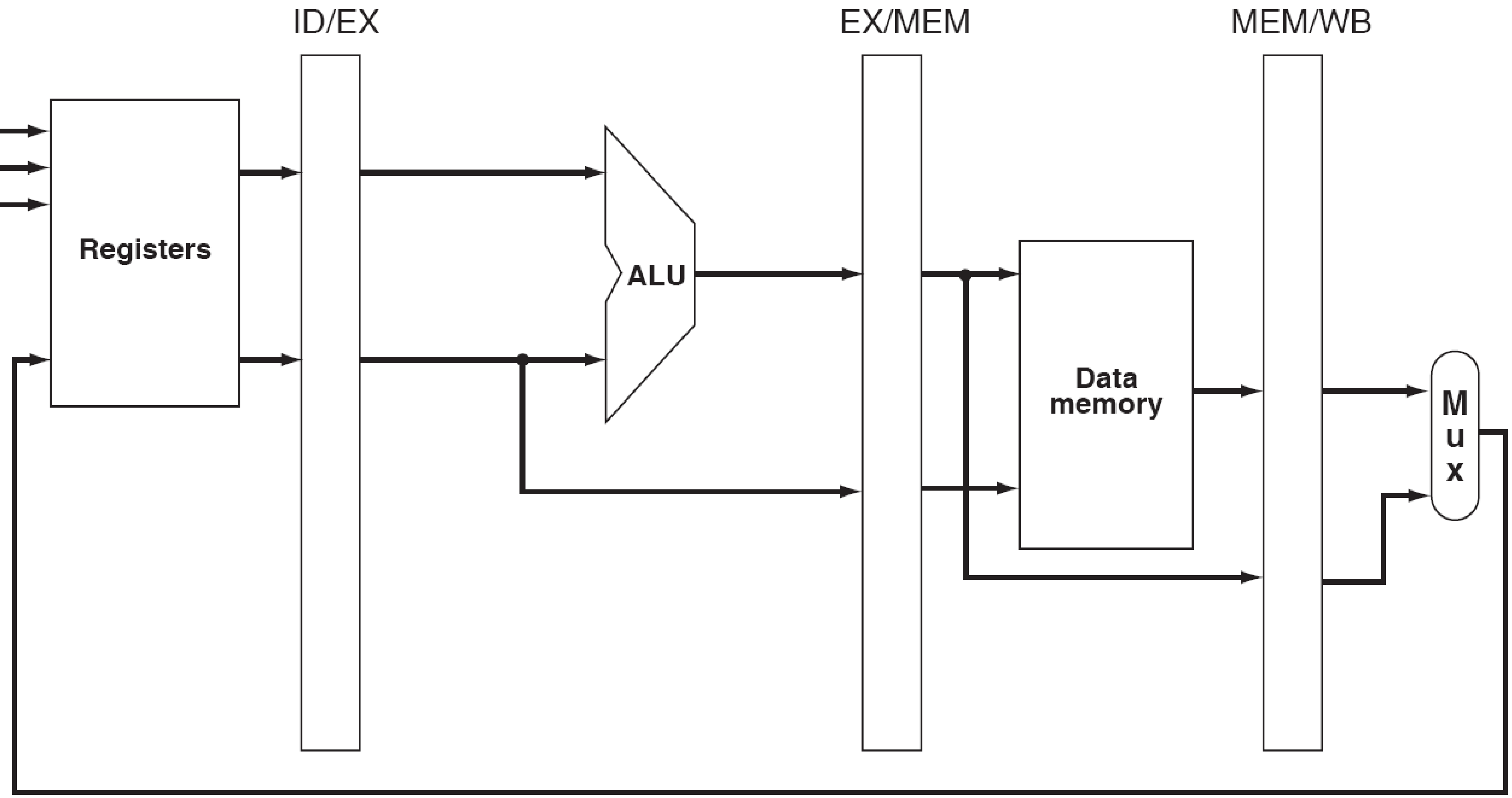
Forwarding

Key idea: connect data internally before it's stored



How would you design the forwarding?

No Forwarding

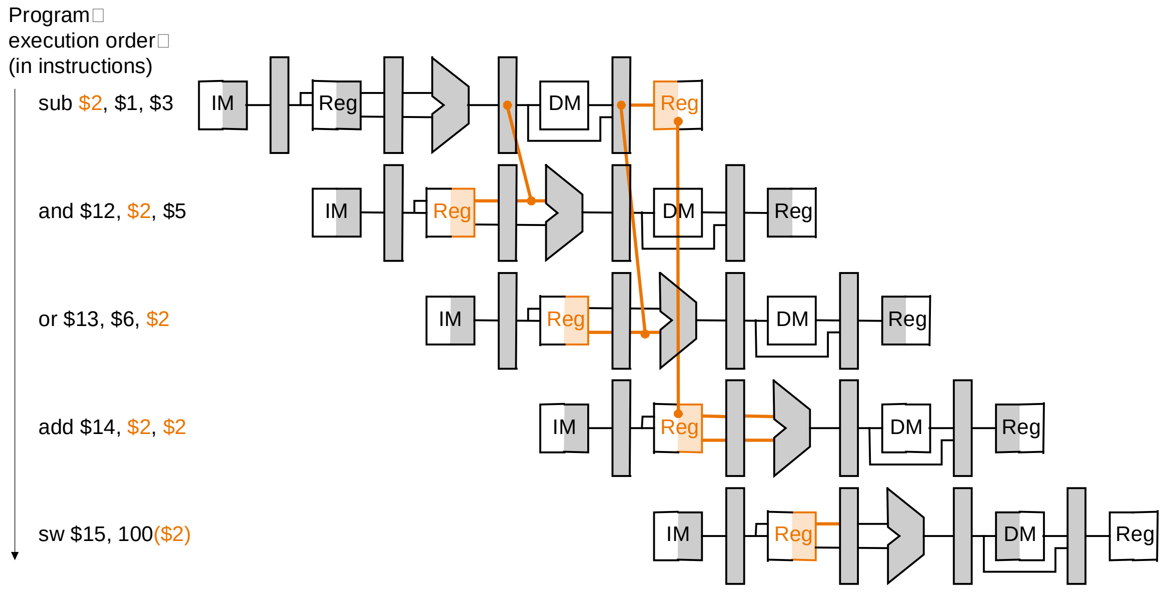


Data Hazard Solution: Forwarding

- Key idea: connect data internally before it's stored

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X



Assumption:

- **The register file forwards values that are read and written during the same cycle.**

Data Hazard Summary

- Three types of data hazards
  - RAW (MIPS)
  - WAW (not in MIPS)
  - WAR (not in MIPS)
- Solution to RAW in MIPS
  - Stall
  - Forwarding
    - Detection & Control
    - A stall is needed if read a register after a load instruction that writes the same register.
  - Reordering

Control Hazards

A *control hazard* is when we need to find the destination of a branch, and can’t fetch any new instructions until we know that destination.

A branch is either

- Taken: PC <= PC + 4 + Imm
- Not Taken: PC <= PC + 4

Control Hazards

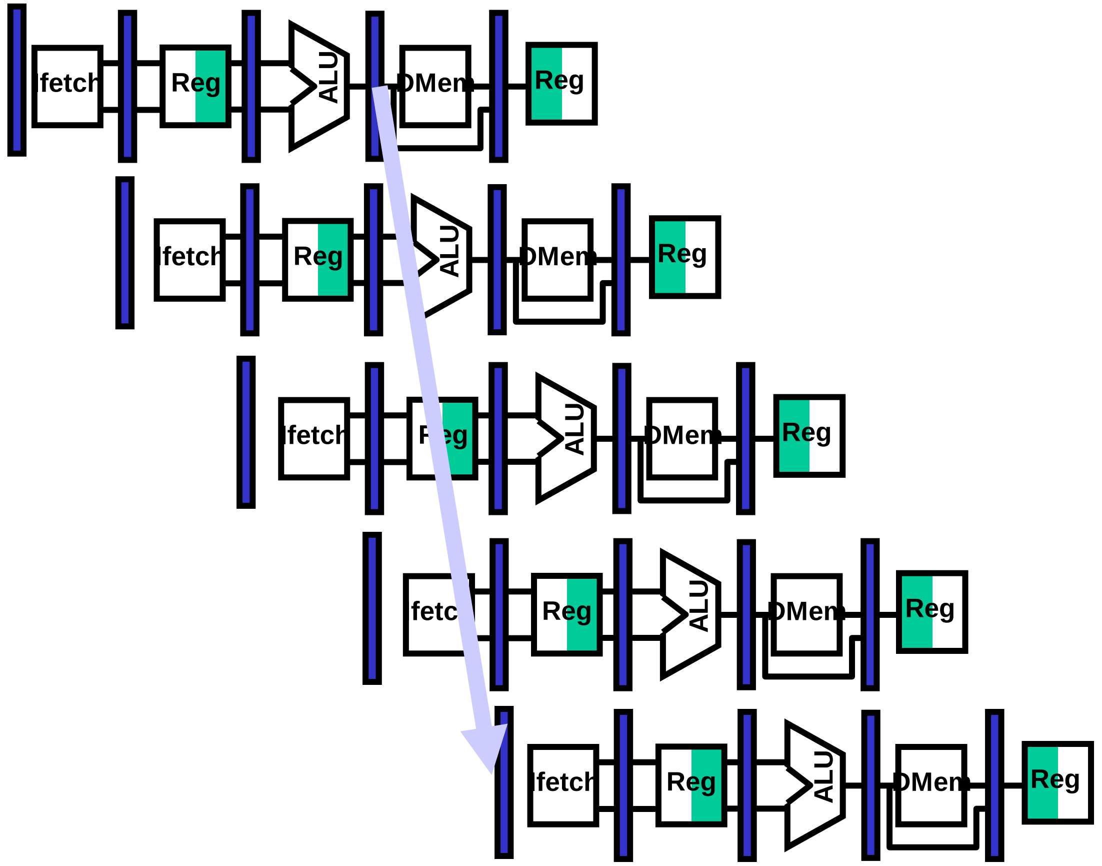
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11

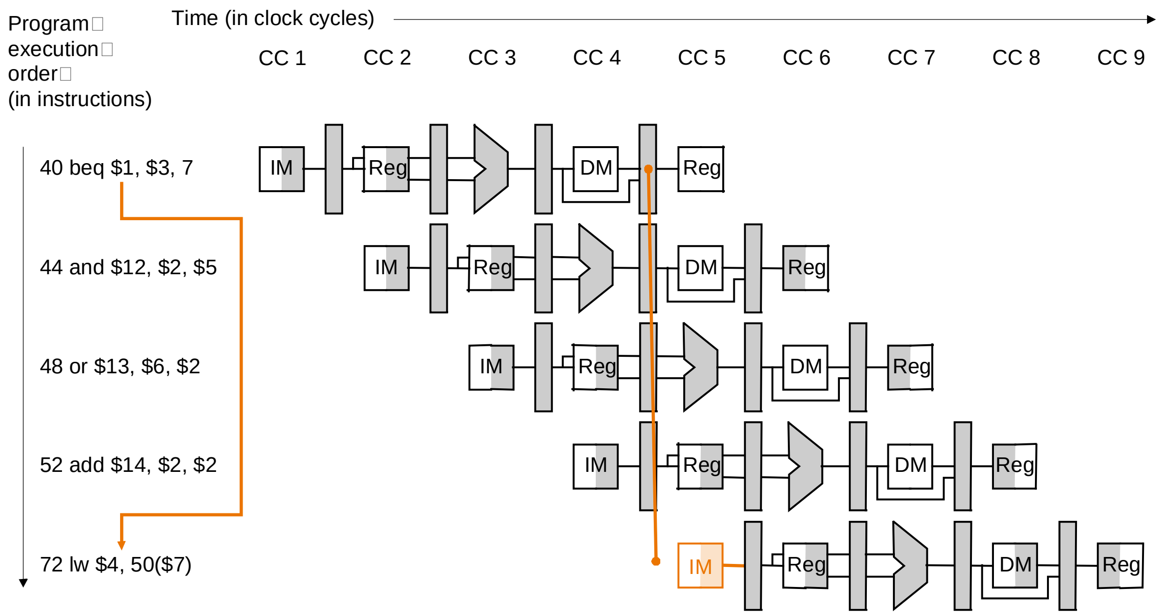


The penalty when branch take is 3 cycles!

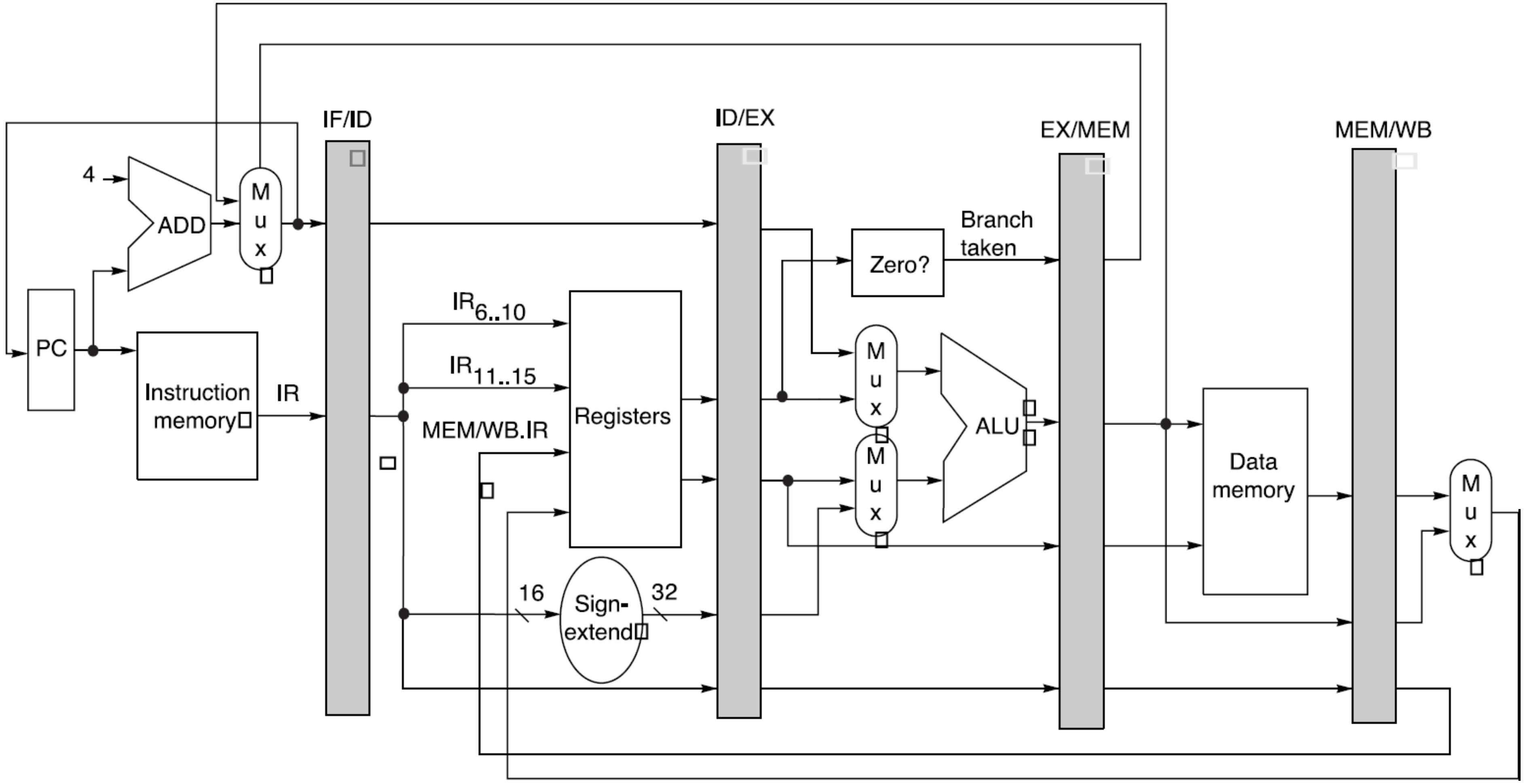
Three Stage Stall

Branch Hazards

- Just stalling for each branch is not practical
  Common assumption: branch not taken
- When assumption fails: flush three instructions



Basic Pipelined Processor



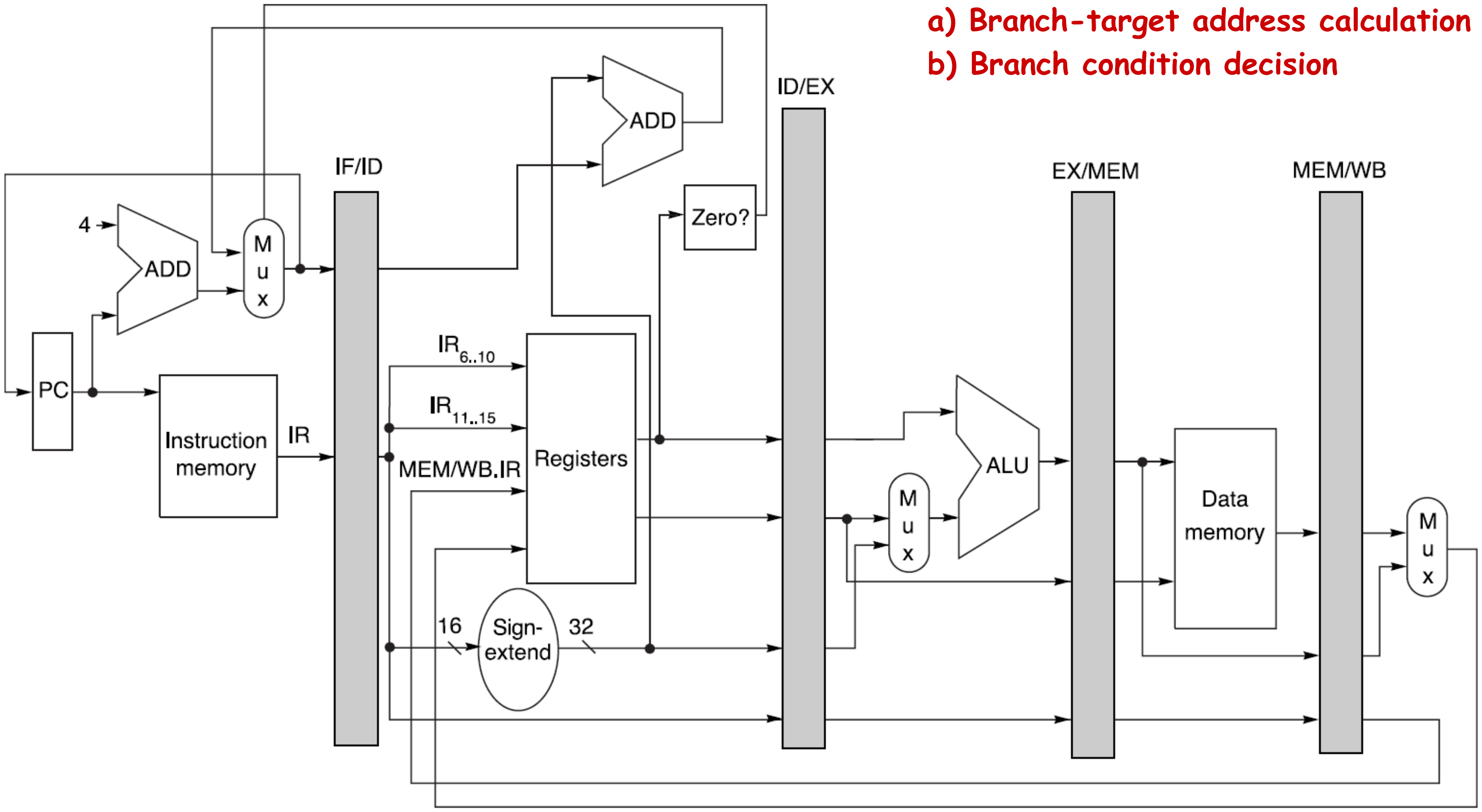
In our original Design, branches have a penalty of 3 cycles

Reducing Branch Delay

Move following to ID stage

a) Branch-target address calculation

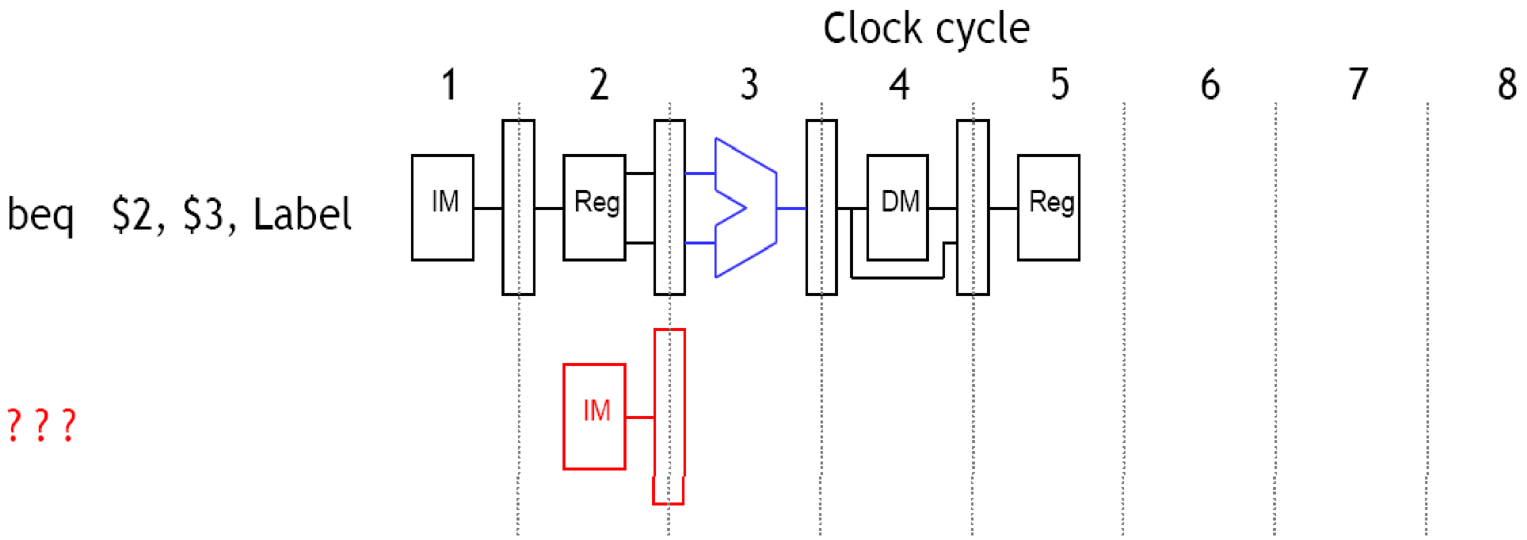
b) Branch condition decision



Reduced penalty (1 cycle) when branch take!

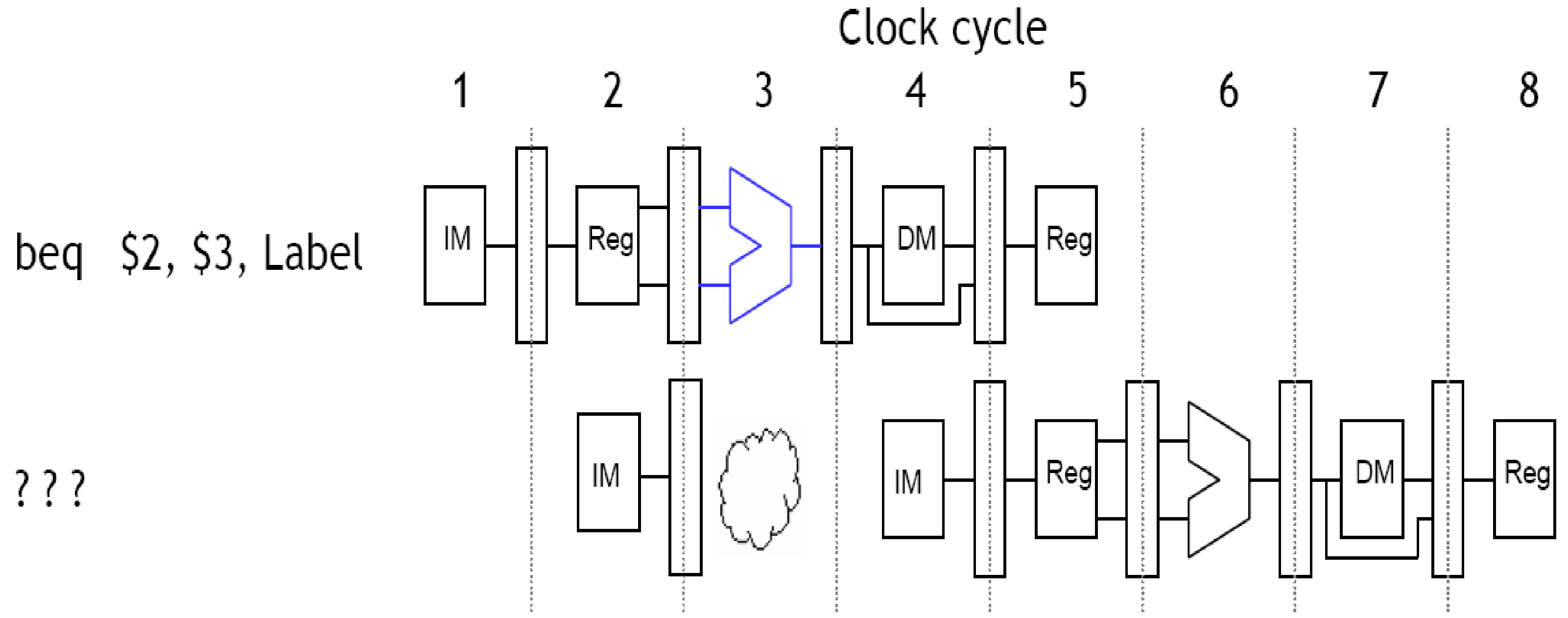
Branches

- Most of the work for a branch computation is done in the EX stage.
  - The branch target address is computed.
  - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what's called a control hazard.



Stalling is one solution

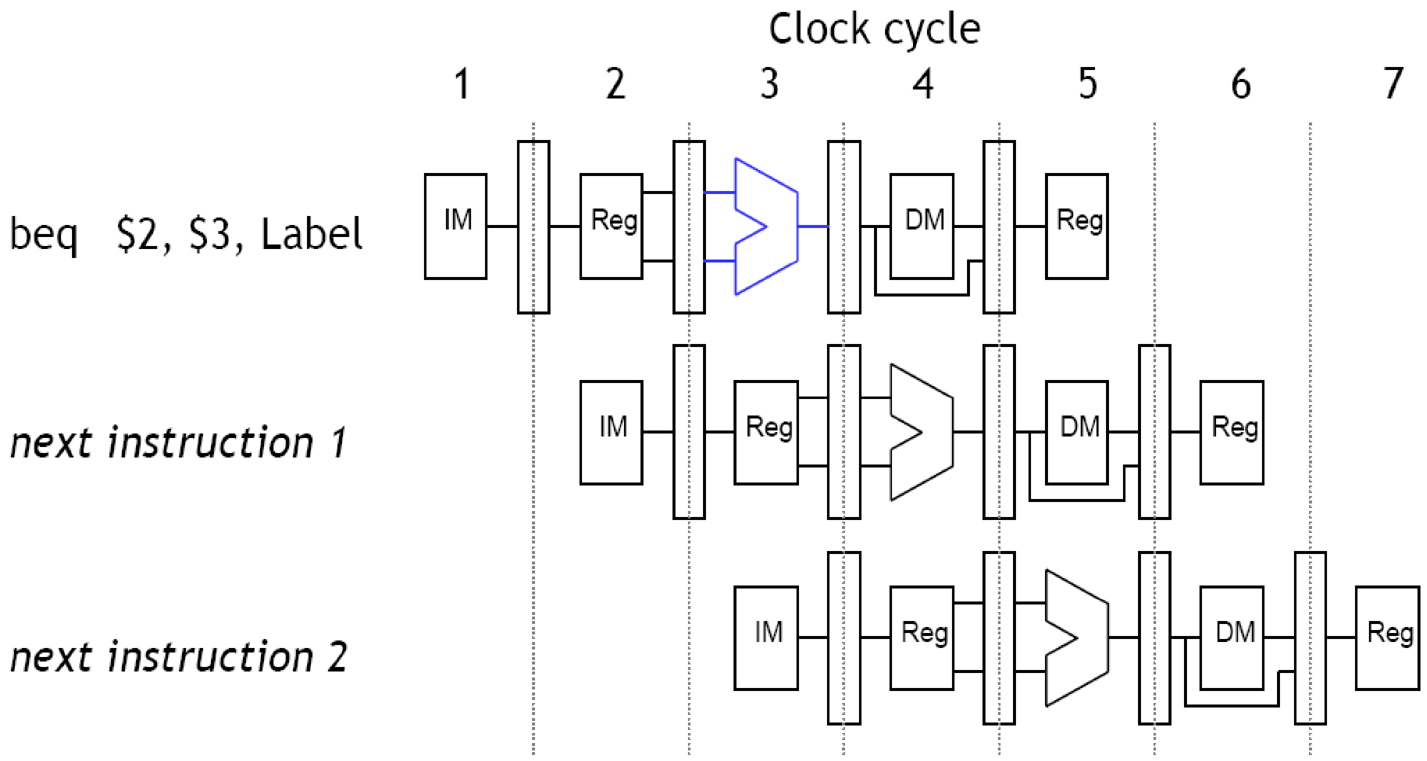
- Again, stalling is always one possible solution.



- Here we just stall until cycle 4, after we do make the branch decision.

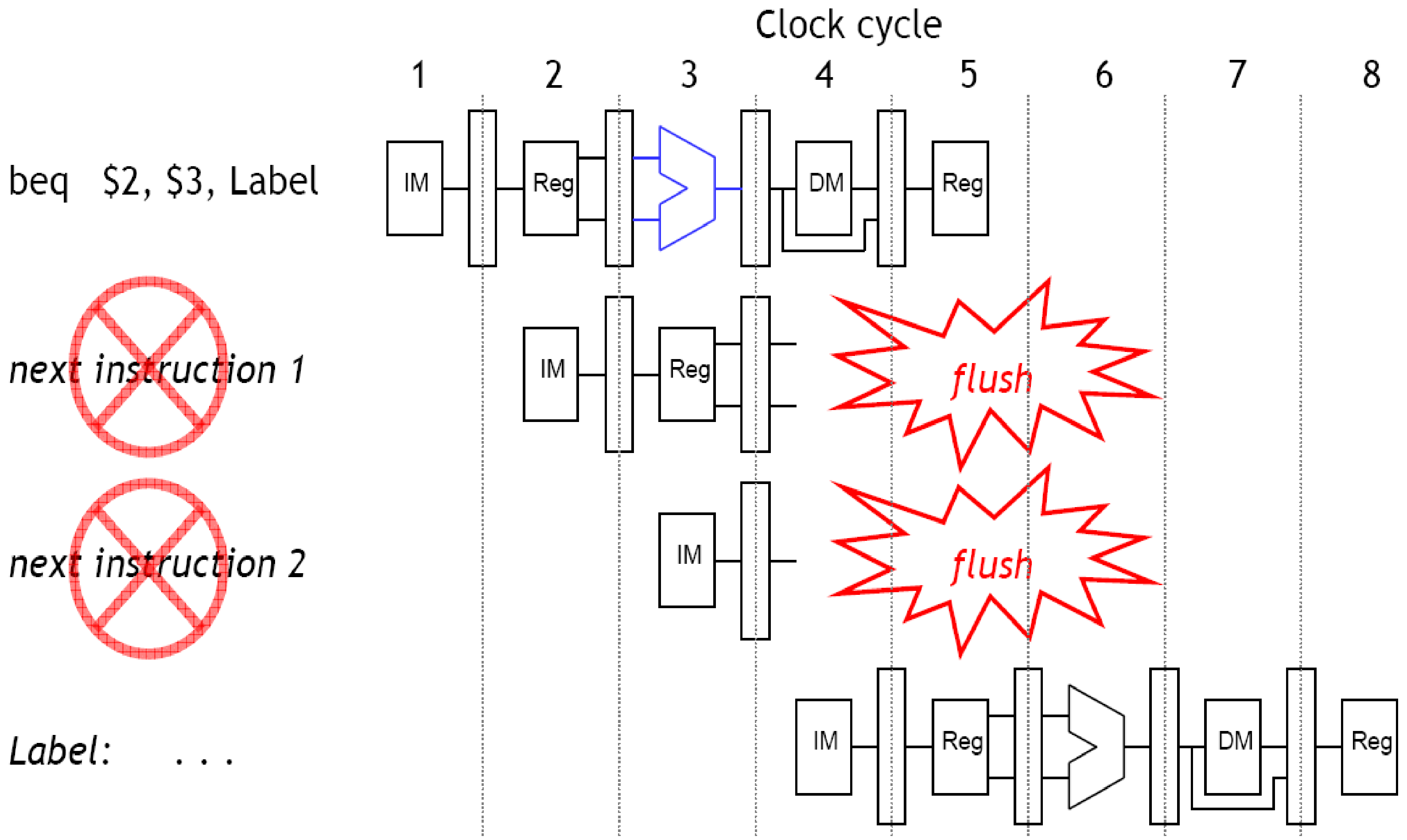
Branch prediction

- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is *not* taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or flush, those instructions and begin executing the right ones from the branch target address, Label.

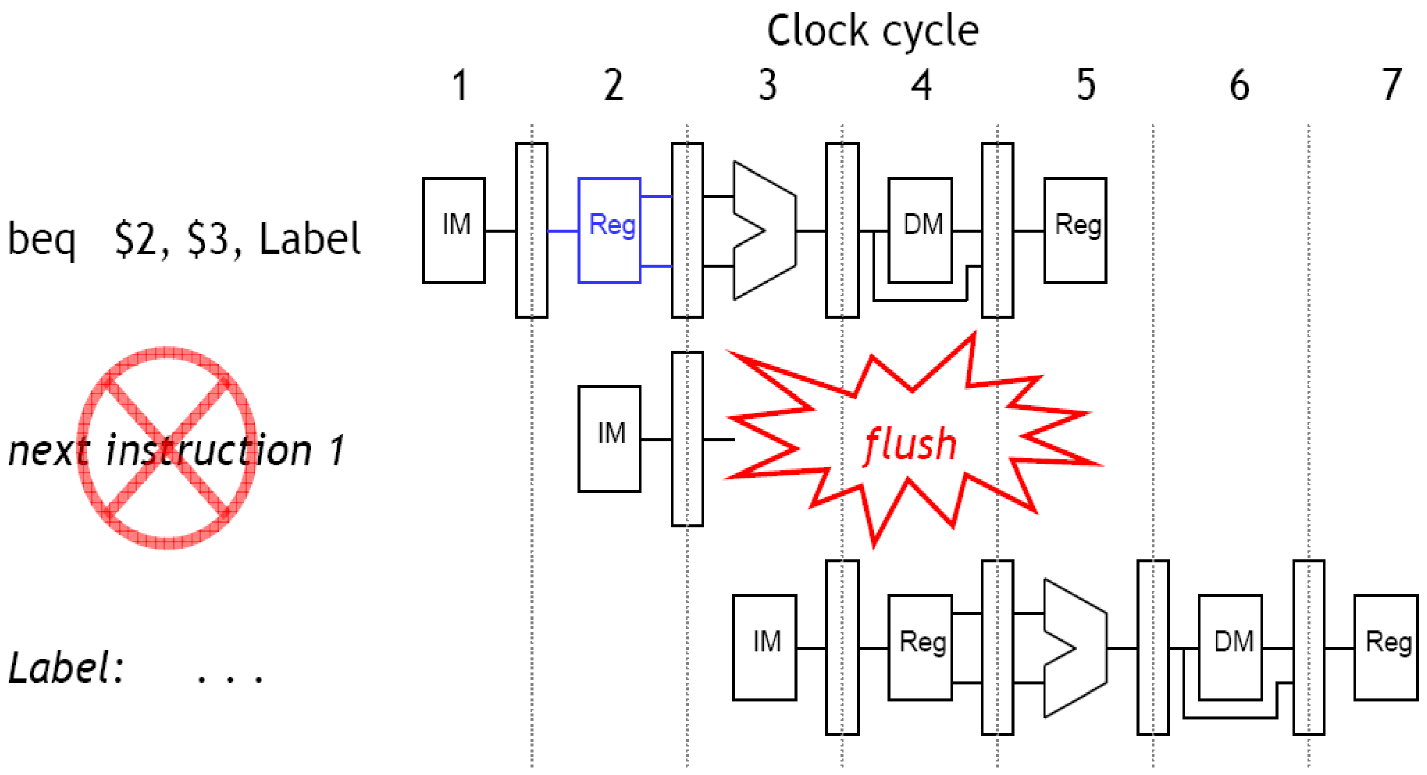


Performance gains and losses

- Overall, branch prediction is worth it.
  - Mispredicting a branch means that two clock cycles are wasted.
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for *every* branch.
- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance.
  - Most CPUs predict branches dynamically—statistics are kept at run-time to determine the likelihood of a branch being taken.
- The pipeline structure also has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must also be careful that instructions do not modify registers or memory before they get flushed.

Implementing branches

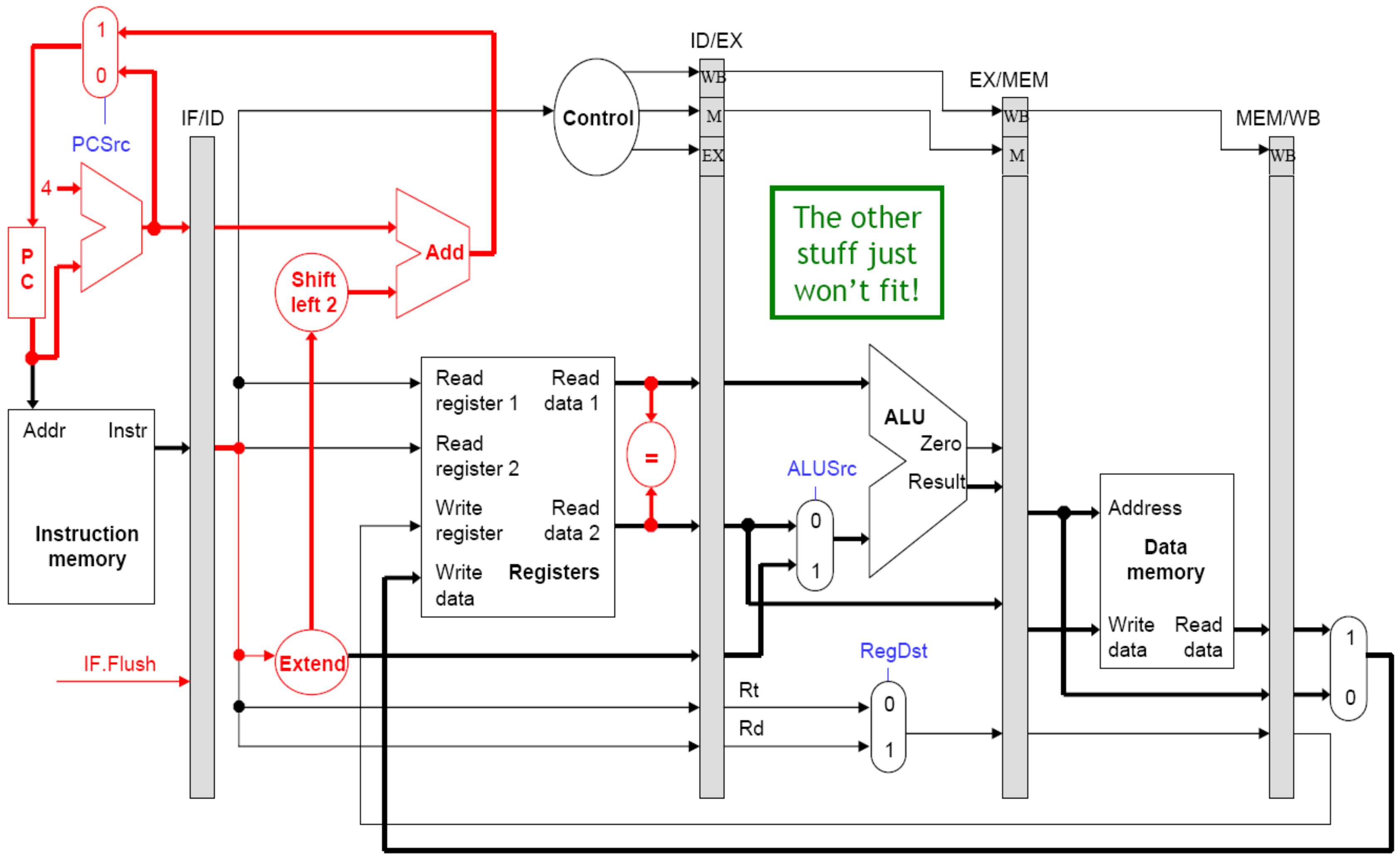
- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a BEQ.
  - We can add a small comparison circuit to the ID stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.



Implementing flushes

- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
  - MIPS uses sll $0, $0, 0 as the nop instruction.
  - This happens to have a binary encoding of all 0s: 0000 .... 0000.
- Flushing introduces a bubble into the pipeline, which represents the one-cycle delay in taking the branch.
- The IF.Flush control signal shown on the next page implements this idea, but no details are shown in the diagram.

Branching *without* forwarding and load stalls



Branch Behavior in Programs

- Based on SPEC benchmarks on DLX
  - Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
  - About 75% of the branches are forward branches
  - 60% of forward branches are taken
  - 80% of backward branches are taken
  - 67% of all branches are taken
- Why are branches (especially backward branches) more likely to be taken than not taken?

Summary

- Three kinds of hazards conspire to make pipelining difficult.
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven't been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a "true" hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.

Written Assignment 2 (1)

When designing memory systems it becomes useful to know the frequency of memory reads versus writes and also accesses for instructions versus data.

We assume the following notation:

IC = number of dynamically executed instructions
f_F = frequency of instruction fetch = 100% by definition
f_L = frequency of load instructions = 26% from given information
f_S = frequency of store instructions = 10% from given information. Please find the percentage of all memory accesses for data

$$\text{Percentage of all memory accesses for data} = \frac{\text{Number of data accesses}}{\text{Number of memory accesses}}$$

$$= \frac{(f\_L + f\_S) * A * IC}{(f\_F+f\_L+f\_S) * A * IC} = \frac{26+10}{100+26+10} = 26.5$$

Written Assignment 2 (2)

What are the economic arguments (i.e., more computers sold) for and against changing instruction set architecture in desktop and server markets?
What about embedded markets?

Desktop: if change provide support for new functions, such as multimedia, then change may enable growth in market share for that segment.

Server: if the new ISA provides better performance, then the trend towards using open source software for servers reduces the cost of change because re-compilation is possible.

Embedded: new ISA allows more and different I/O in the base chip, reducing the need for support chips, and thus, system cost.

Sample Question

- For the following code identify all data dependence between instructions. Please use the format “instruction X depends on instruction Y over register Z”.

```
1    L.D       F0, 0 (R1)
2    ADD.D     F4, F0, F2
3    S.D       F4, 0(R1)
4    L.D       F0, -8(R1)
5    ADD.D     F4, F0, F2
6    S.D       F4, -8(R1)
```

Instruction 2 depends on instruction 1 (instruction 1 result in F0 used by instruction 2), Similarly, instructions (4,5)
Instruction 3 depends on instruction 2 (instruction 2 result in F4 used by instruction 3), Similarly, instructions (5,6)