

Topic: Pipelining



CSE 30: Computer Organization and Systems Programming

Diba Mirza

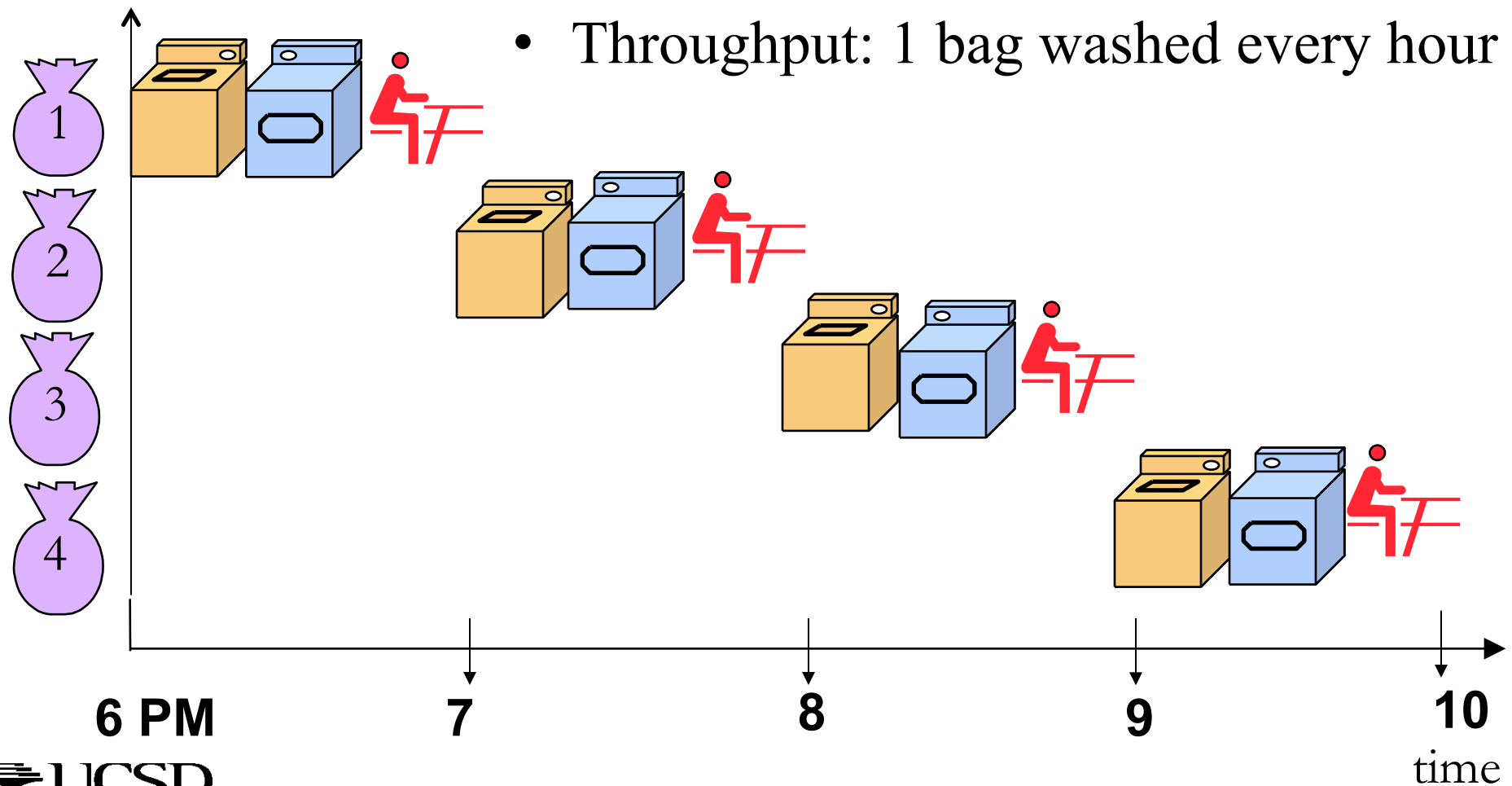
Dept. of Computer Science and Engineering
University of California, San Diego

Pipelining

- Pipelining is a way of increasing the throughput of a processor i.e. No. of instructions executed per second
- This is different from the actual time required to execute one instruction which is the instruction latency

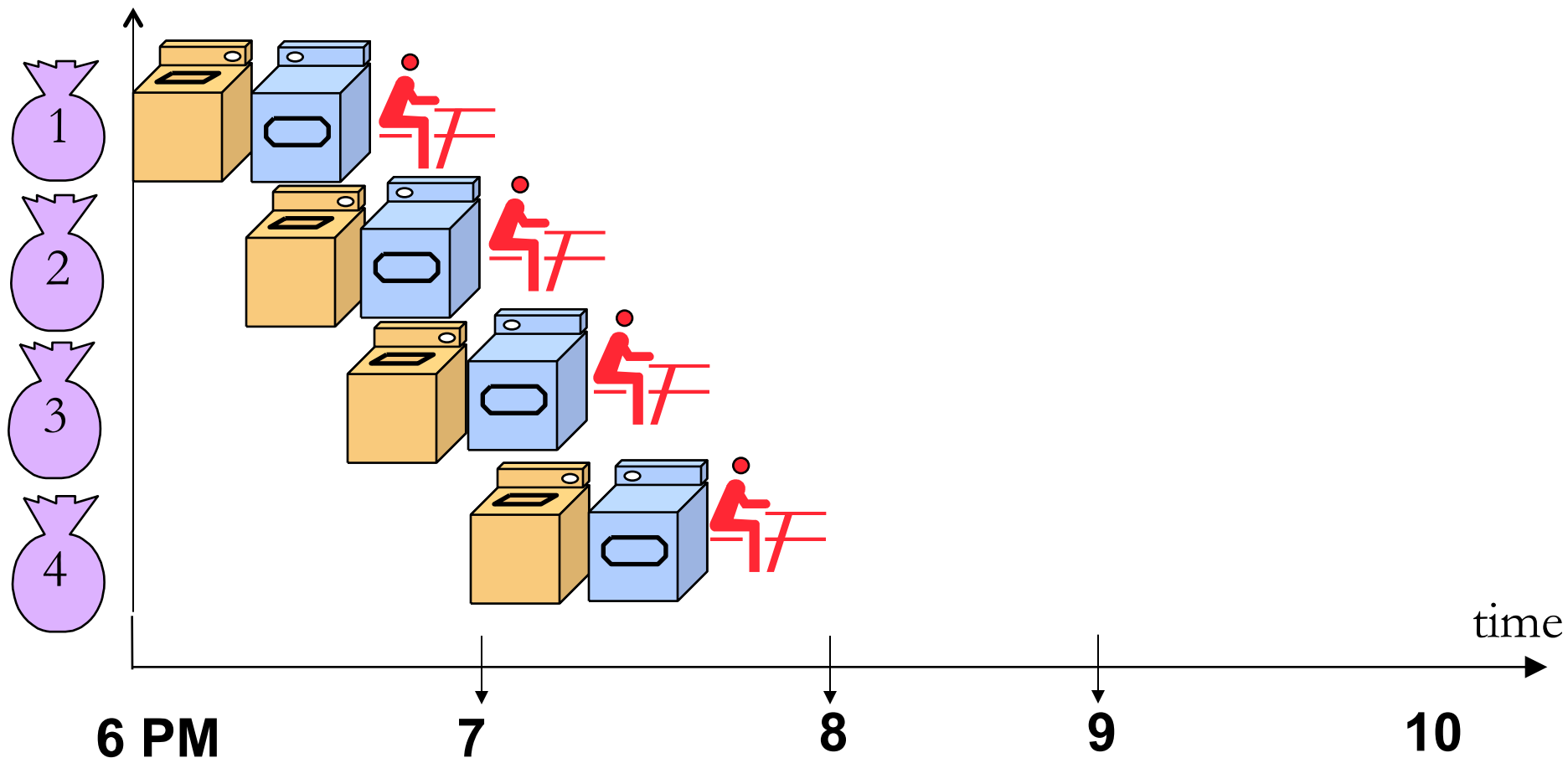
Pipelining Analogy: Laundry

- Recognize: The Laundry can be broken into subtasks each of which uses an independent resource



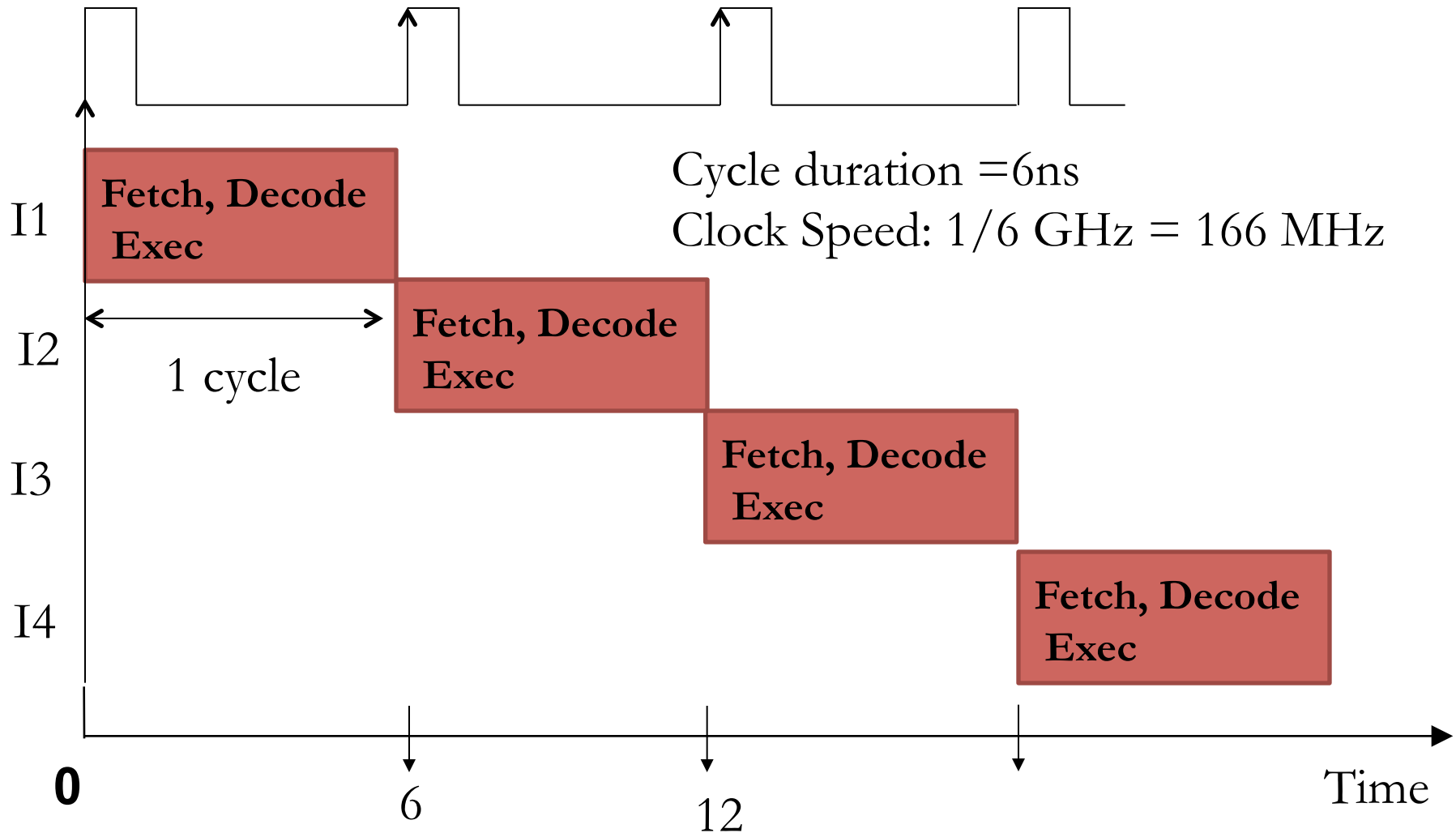
Pipelined Laundry

- Pipelined Throughput: 1 bag every 20 min
- 3x faster than non-pipelined case



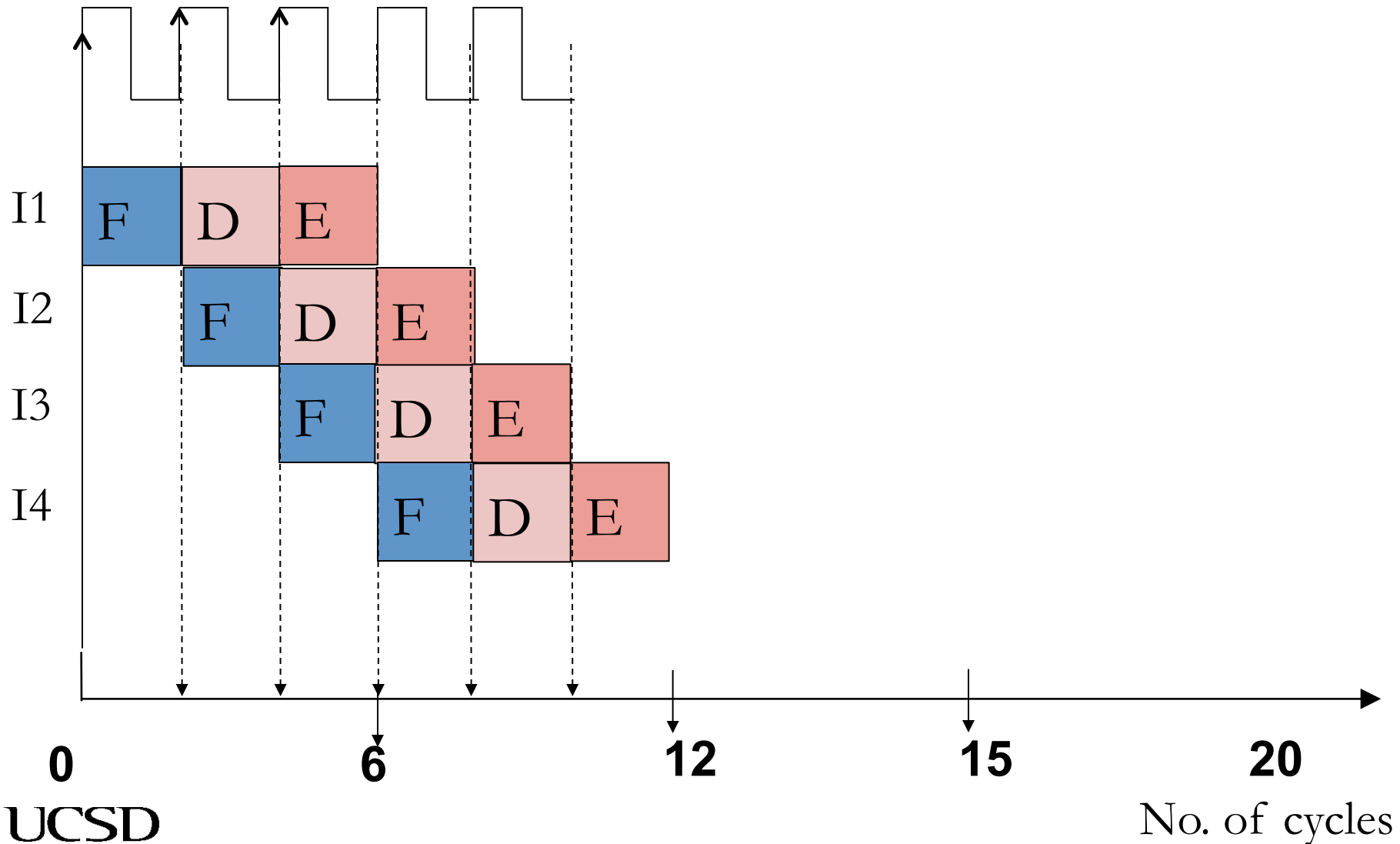
Application of Laundry (Analogy) to Instructions

- Processing an instruction: i) Fetch ii) Decode iii) Execute



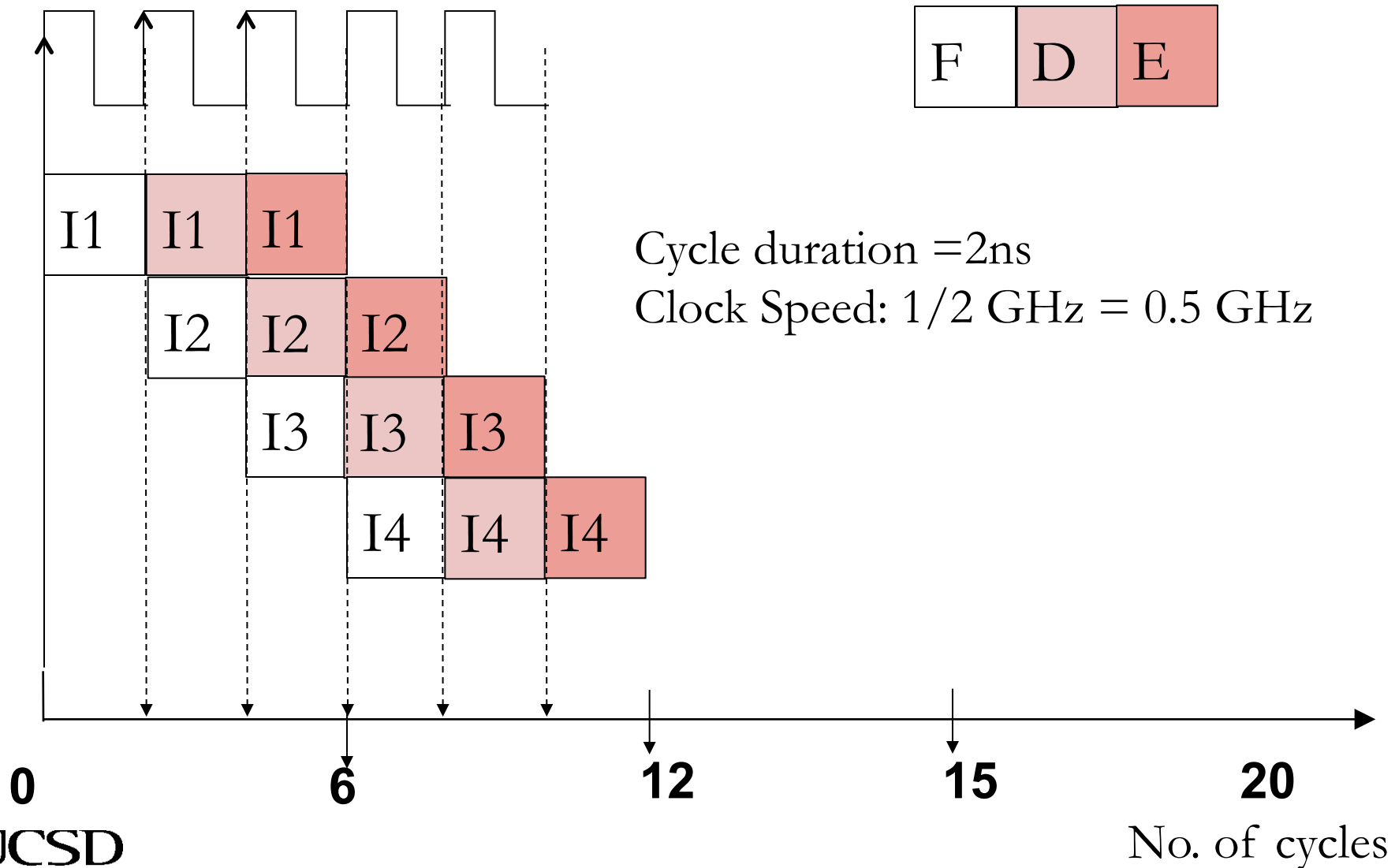
Application of Laundry (Analogy) to Instructions

- The clock period is reduced by a factor of 3

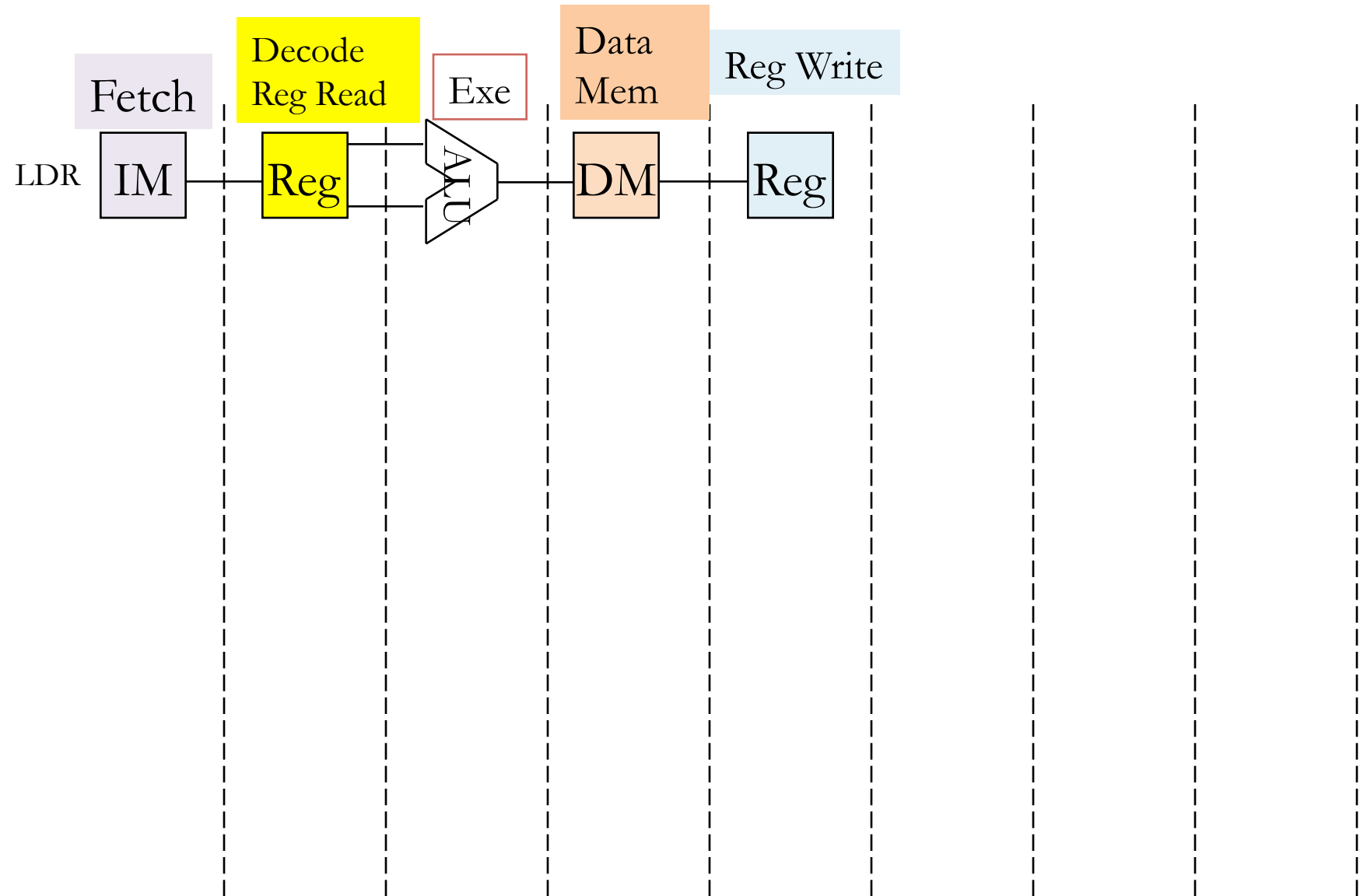


Application of Laundry (Analogy) to Instructions

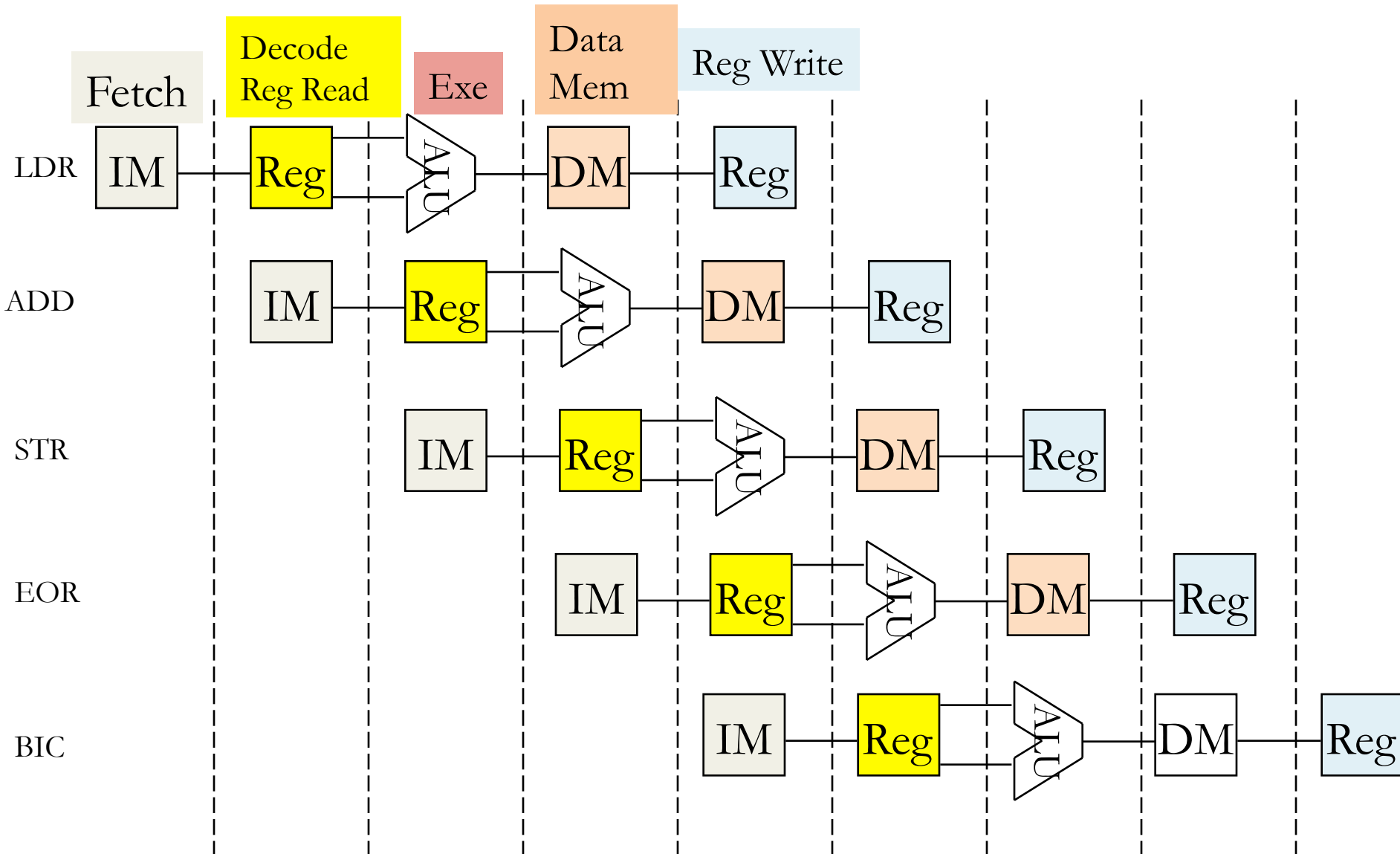
- The clock period is reduced by a factor of 3



A 5-Stage Pipeline



Program Execution in a 5-Stage Pipeline



Pipelining Hazards

Situations that prevent starting the next instruction in the next cycle

- **Structural hazard**
 - A required resource is busy
- **Data hazard**
 - One of the stages of executing a particular instruction needs to wait for previous instruction to complete
- **Control hazard (next lecture)**
 - Deciding on control action depends on previous instruction

Structural Hazards

Conflict for use of a resource

- Load/store requires data (memory) access
- Instruction fetch requires memory access

LDR

ADD

SUB

BIC

Cycle 1

Cycle 2

Cycle 3

Cycle 4

Cycle 5

Cycle 6

Cycle 7

Cycle 8

If instruction and data memory had a single port of access, in which cycle would a hazard occur?

- A. Cycle 2
- B. Cycle 3
- C. Cycle 4
- D. Cycle 5
- E. Cycle 2 and 5

LDR

ADD

SUB

BIC

Cycle 1

Cycle 2

Cycle 3

Cycle 4

Cycle 5

Cycle 6

Cycle 7

Cycle 8

If instruction and data memory had a single port of access, in which cycle would a hazard occur?

A. Cycle 2

B. Cycle 3

C. Cycle 4; DM and IM conflict

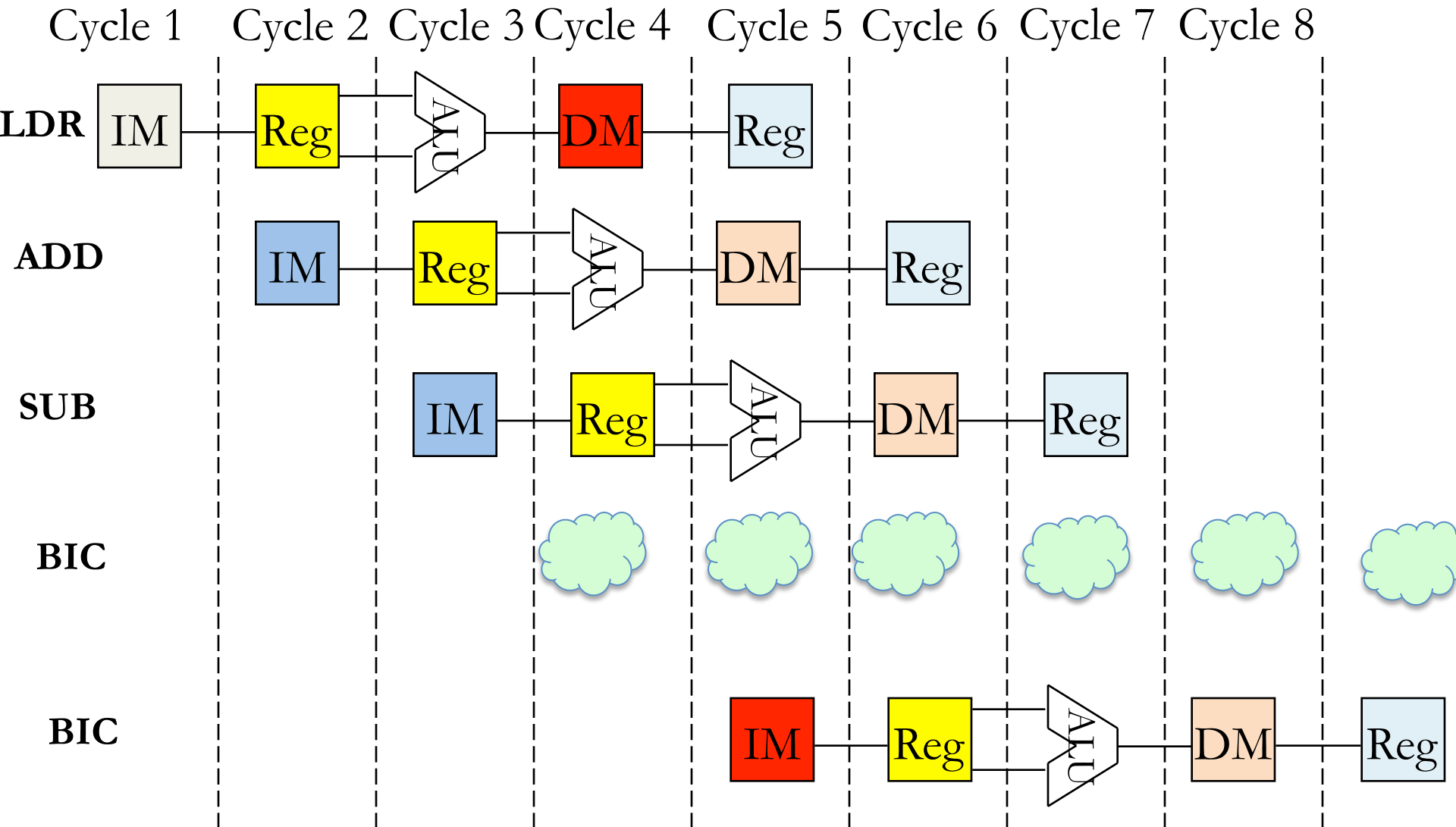
D. Cycle 5

E. Cycle 2 and 5

Solving Structural Hazards

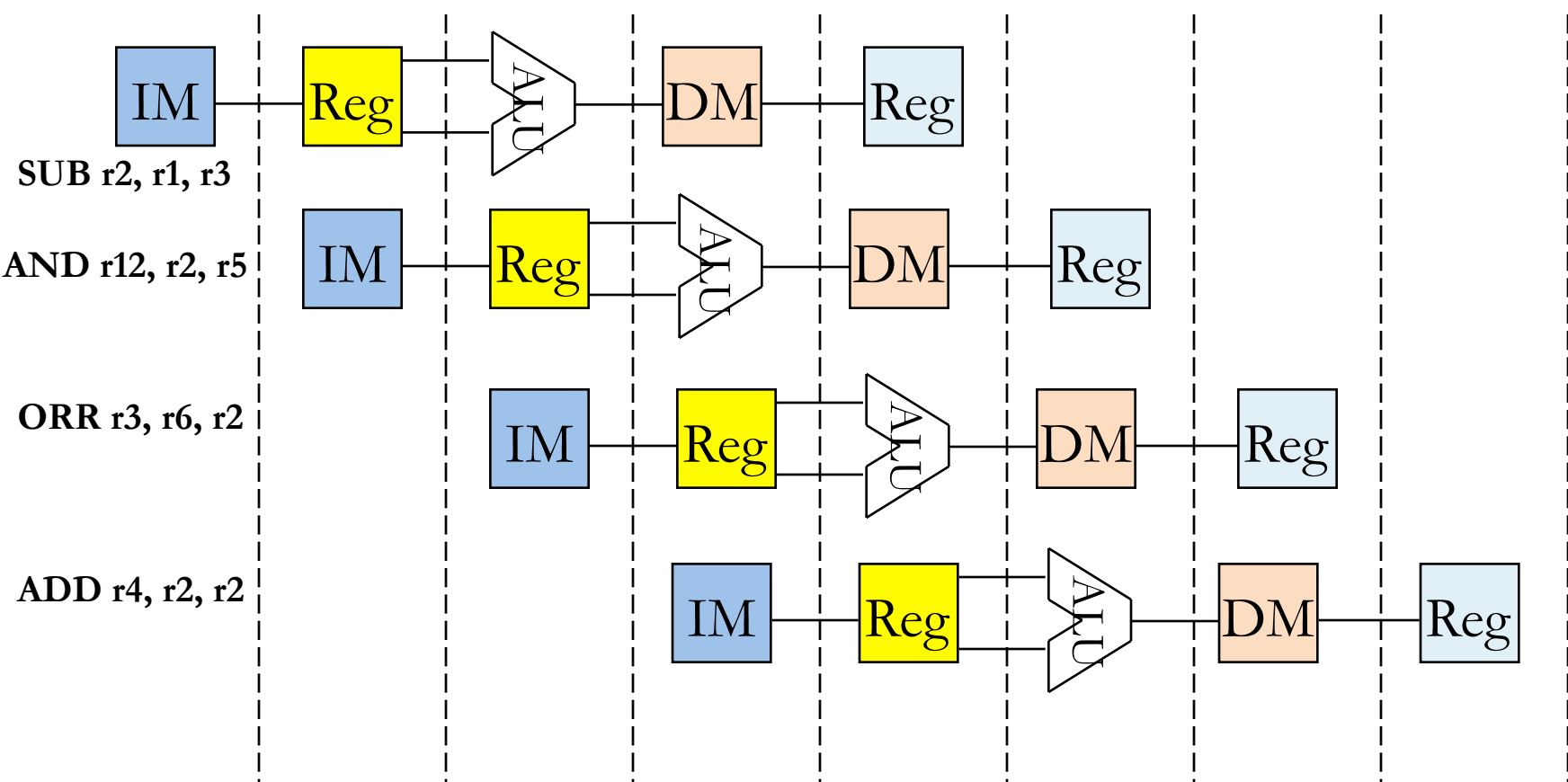
- A. *Stall* instruction fetch for the cycle with Load/store data fetch
- B. Separate ports of access for instruction/data memories
- C. Both
- D. Neither

Stalling: Buuuuubbles!



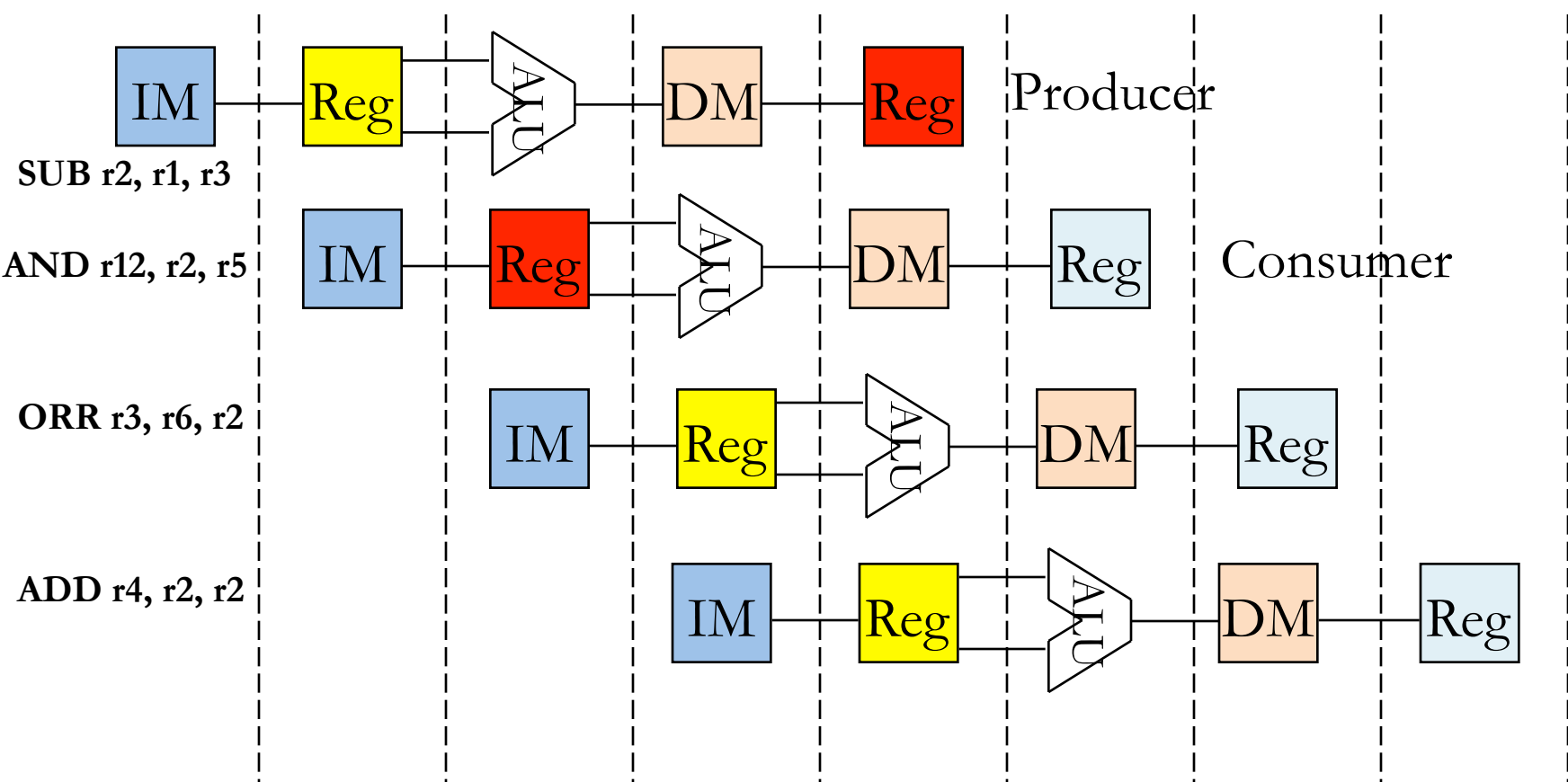
Solving Structural Hazards

Instead of stalling, in most processors separate ports (memories) used for accessing instruction and data memories



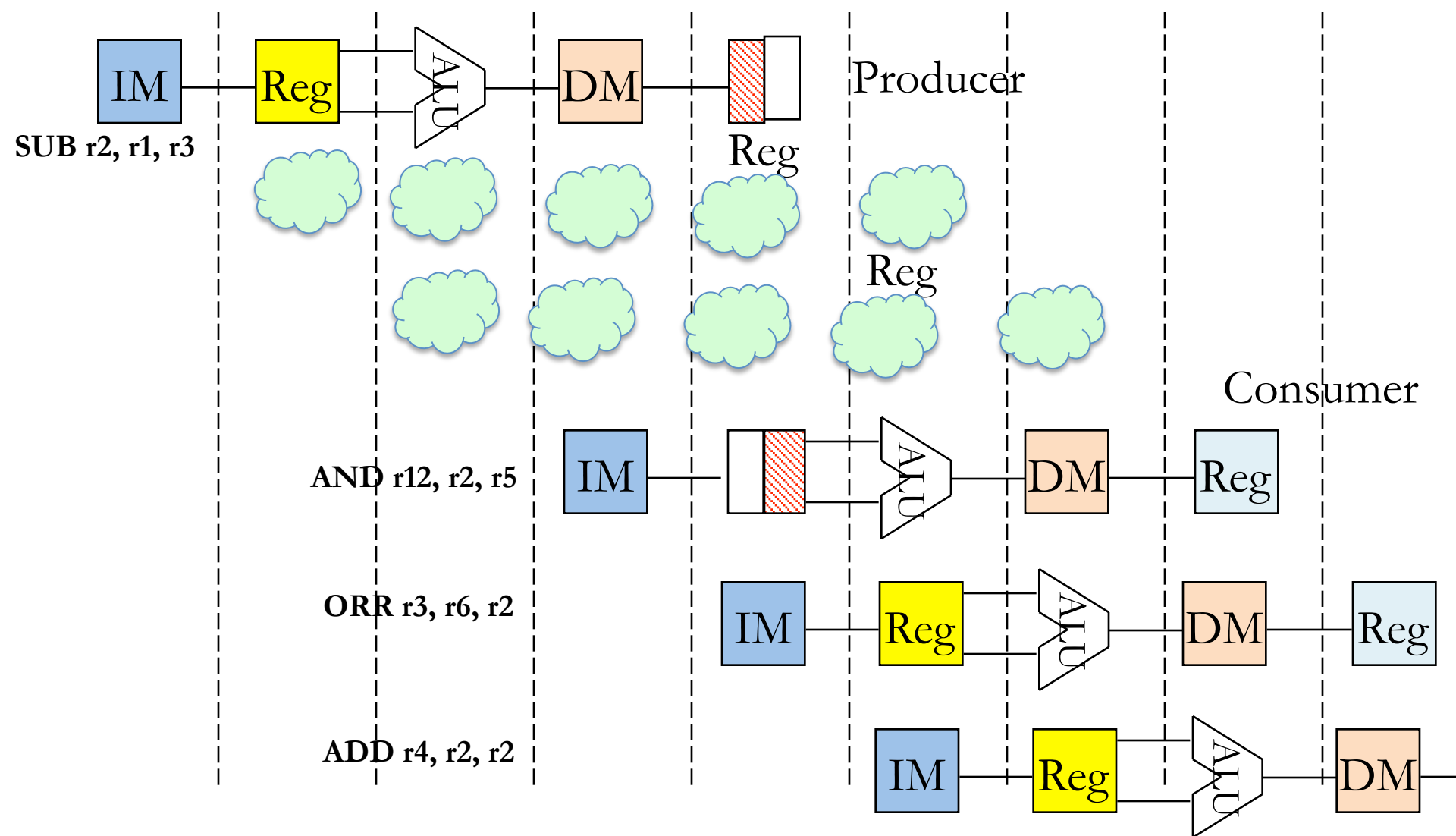
What just happened here which is problematic (BEST ANSWER)?

- A. The register file is trying to read and write the same register
- B. The ALU and data memory are both active in the same cycle
- C. A value is used before it is produced
- D. Both A and B
- E. Both A and C



What just happened here which is problematic (BEST ANSWER)?

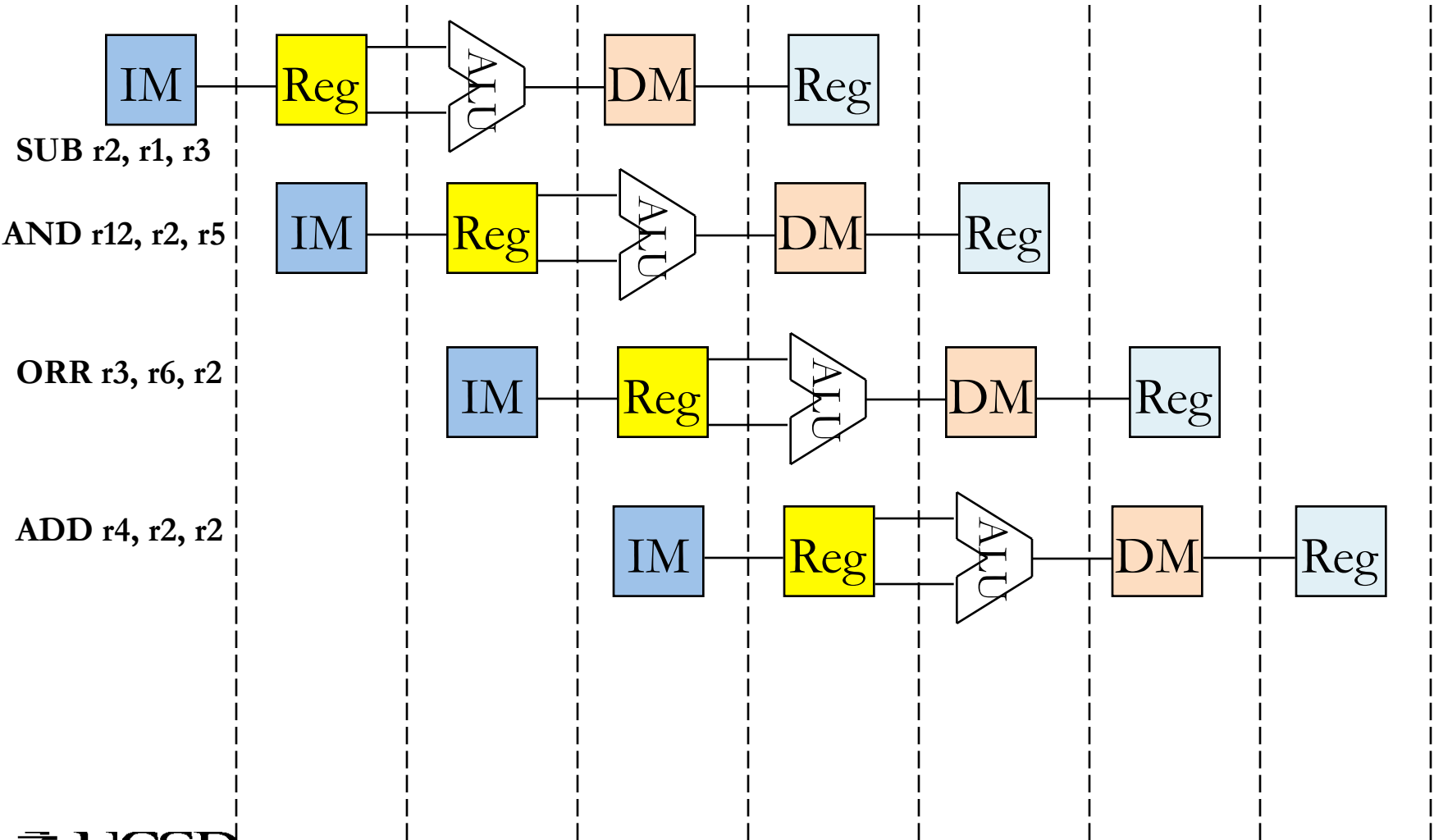
- A. The register file is trying to read and write the same register
- B. The ALU and data memory are both active in the same cycle
- C. A value is used before it is produced: Consumer needs value that hasn't been produced yet: Data hazard
- D. Both A and B
- E. Both A and C



2 instruction gap between producer and consumer

Data Hazards

When a result is needed in the pipeline before it is available, a “data hazard” occurs.

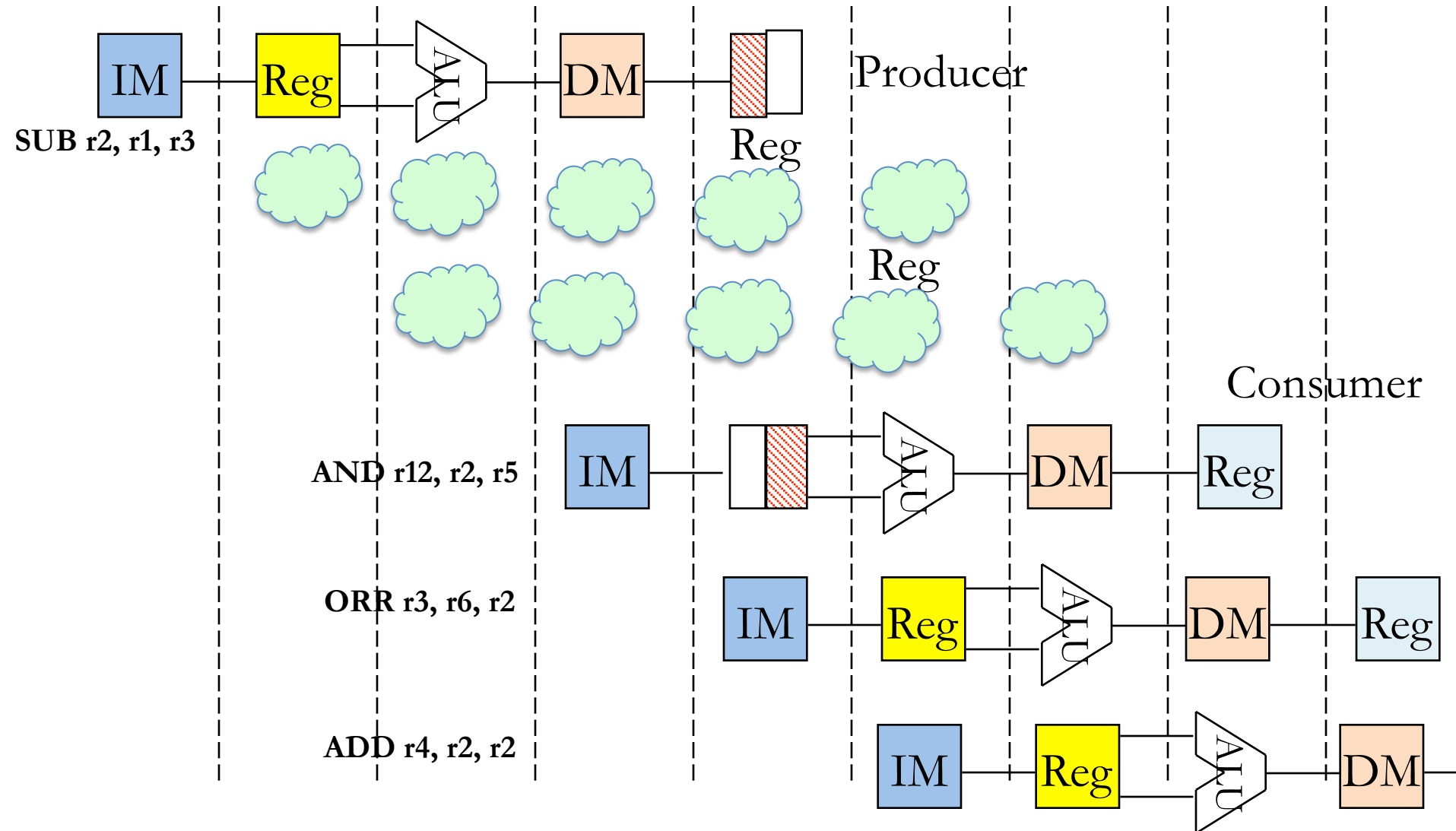


Data Hazards can be solved by

- A. Not running the second instruction until the data is ready (Stalling)
- B. Sending the calculated value straight from the ALU to the next instruction, skipping the registers
- C. Reordering instructions

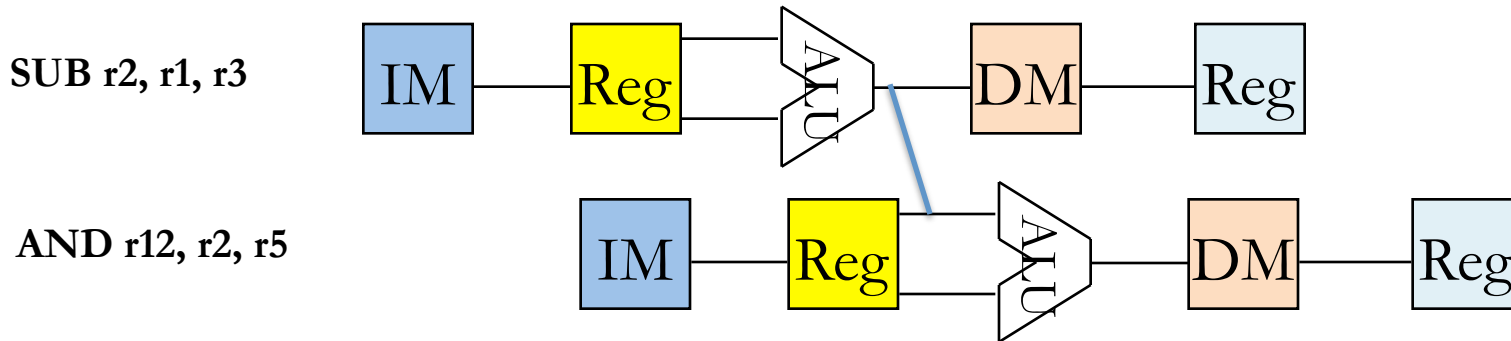
Solving data hazards: Stalling

2 instruction gap between producer and consumer



Solving data hazards: Forwarding (aka Bypassing)

- Use result when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath



Q: Would forwarding work (i.e. there is no need for any stalls) if our instructions were?

```
LDR r0, [r1]
```

```
SUB r2, r0, r3
```

A. Yes

B. No

Q: Would forwarding work (i.e. there is no need for any stalls) if our instructions were?

```
LDR r0, [r1]
```

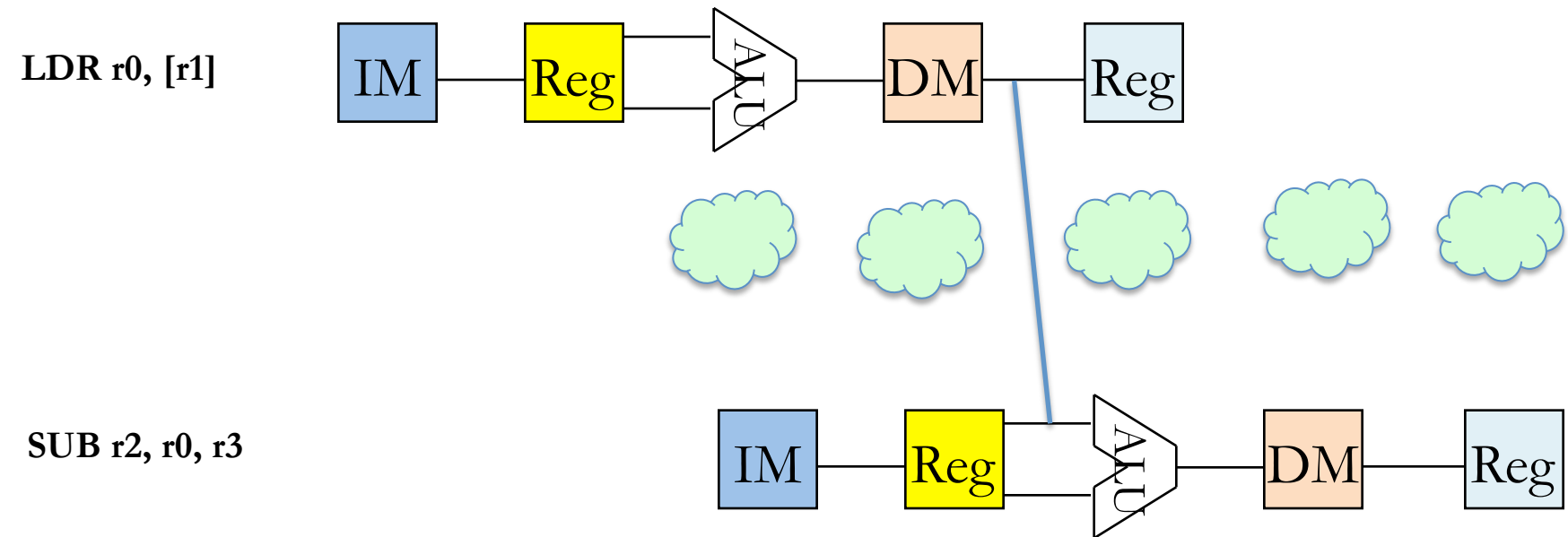
```
SUB r2, r0, r3
```

A. Yes

B. No

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!



Code Scheduling to Avoid Stalls

- How many stalls needed if forwarding was used?
 - A. One
 - B. Two
 - C. Three
 - D. Four
 - E. None

```
LDR r1, [r0]
LDR r2, [r0, #4]
ADD r3, r1, r2
STR r3, [r0]
LDR r4, [r0, #8]
ADD r5, r1, r4
STR r5, [r0]
```

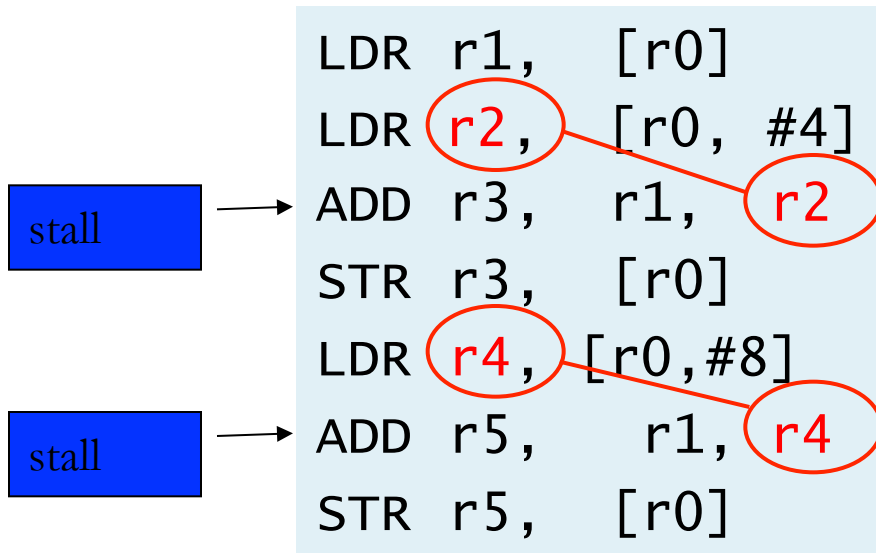
Code Scheduling to Avoid Stalls

- How many stalls needed if forwarding was used?
 - A. One
 - B. Two
 - C. Three
 - D. Four
 - E. None

```
LDR r1, [r0]
LDR r2, [r0, #4]
ADD r3, r1, r2
STR r3, [r0]
LDR r4, [r0, #8]
ADD r5, r1, r4
STR r5, [r0]
```

Code Scheduling to Avoid Stalls

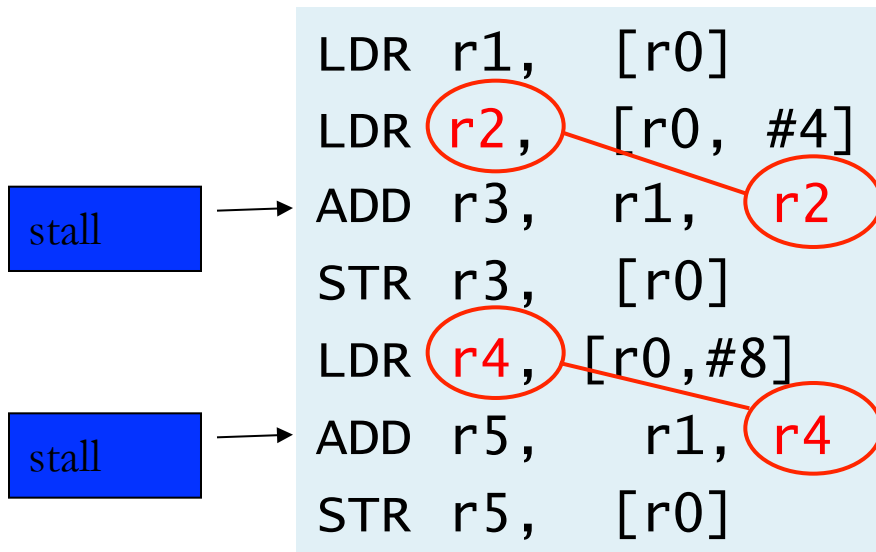
- How many stalls needed if forwarding was used?
 - A. One
 - B. Two: Two data hazards, need to stall for one cycle in each case)**
 - C. Three
 - D. Four
 - E. None



Code Scheduling to Avoid Stalls

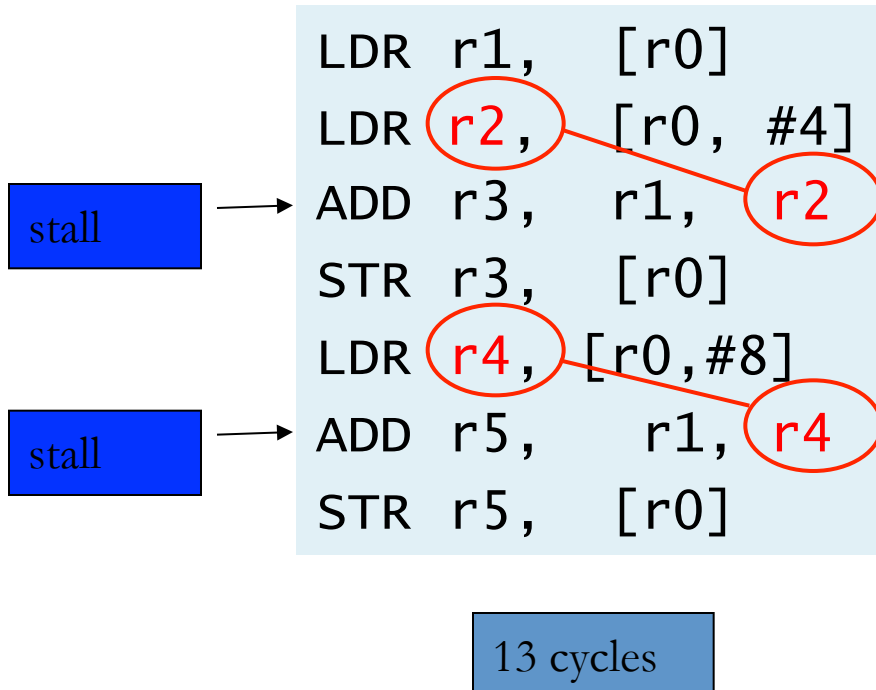
In how many cycles would the code execute?
(assume a 5 stage pipeline)

- A. Seven (no. of instructions)
- B. Nine
- C. Eleven
- D. Thirteen
- E. Fifteen



Code Scheduling to Avoid Stalls

In how many cycles would the code execute?
(assume a 5 stage pipeline)

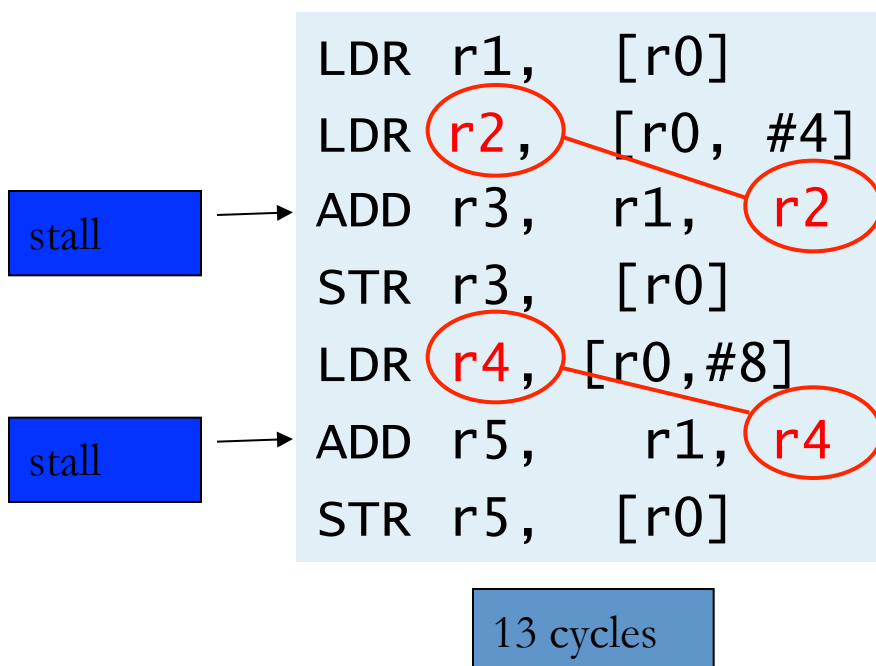


- A. Seven (no. of instructions)
- B. Nine
- C. Eleven
- D. Thirteen (5 for the first, and one for each subsequent instruction)
- E. Fifteen

Performance Metric: CPI

Important performance metric: (Avg) Cycles per instruction (CPI).

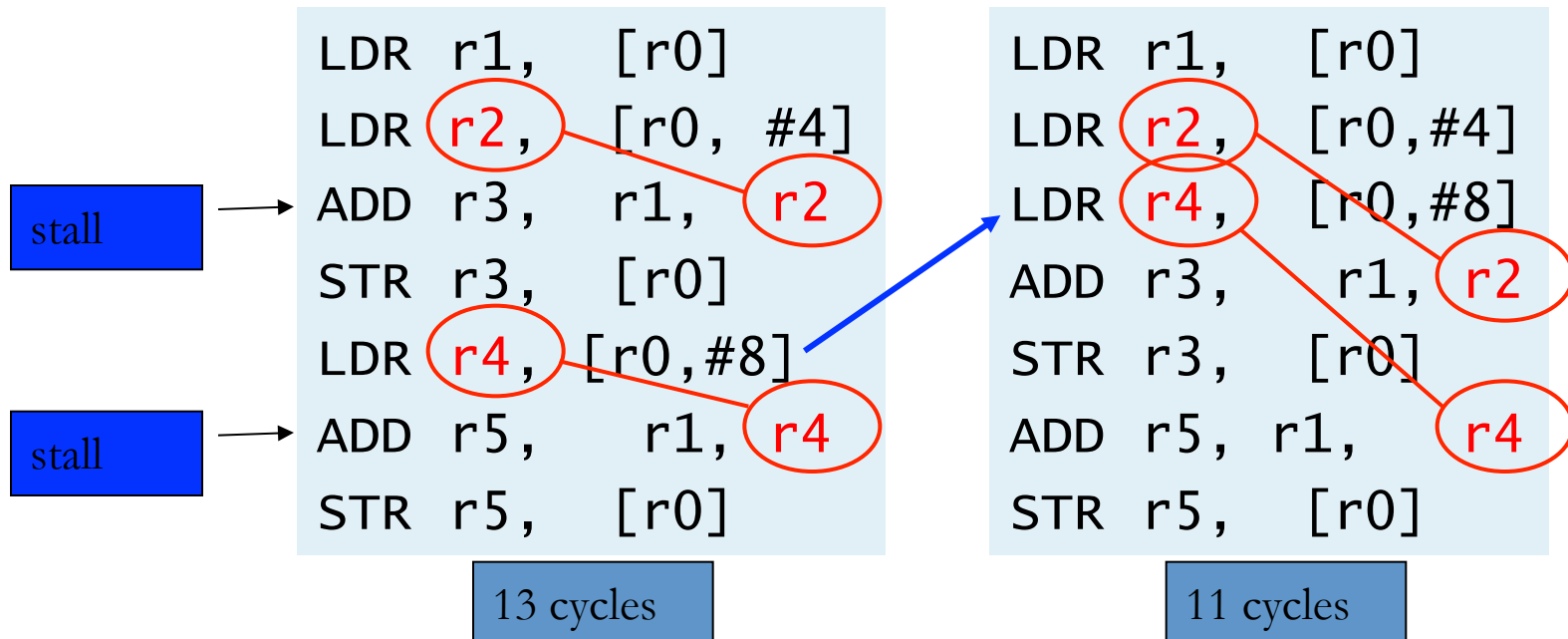
What is the CPI for the code below:



- A. 13
- B. 7
- C. $7/13 \sim 0.54$
- D. $13/7 \sim 1.86$

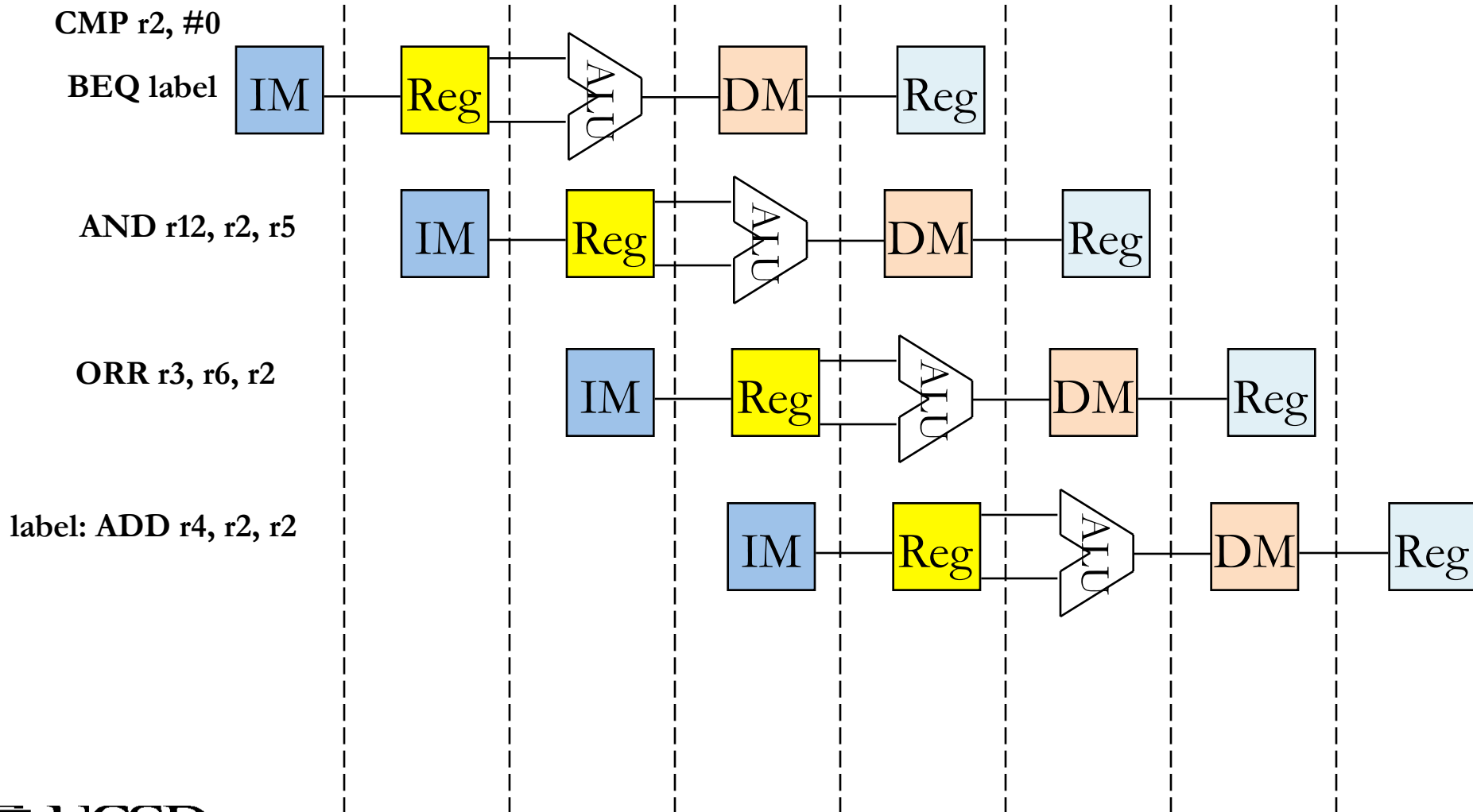
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction



Control Hazards

Disruption of the pipeline on encountering a Branch Instruction



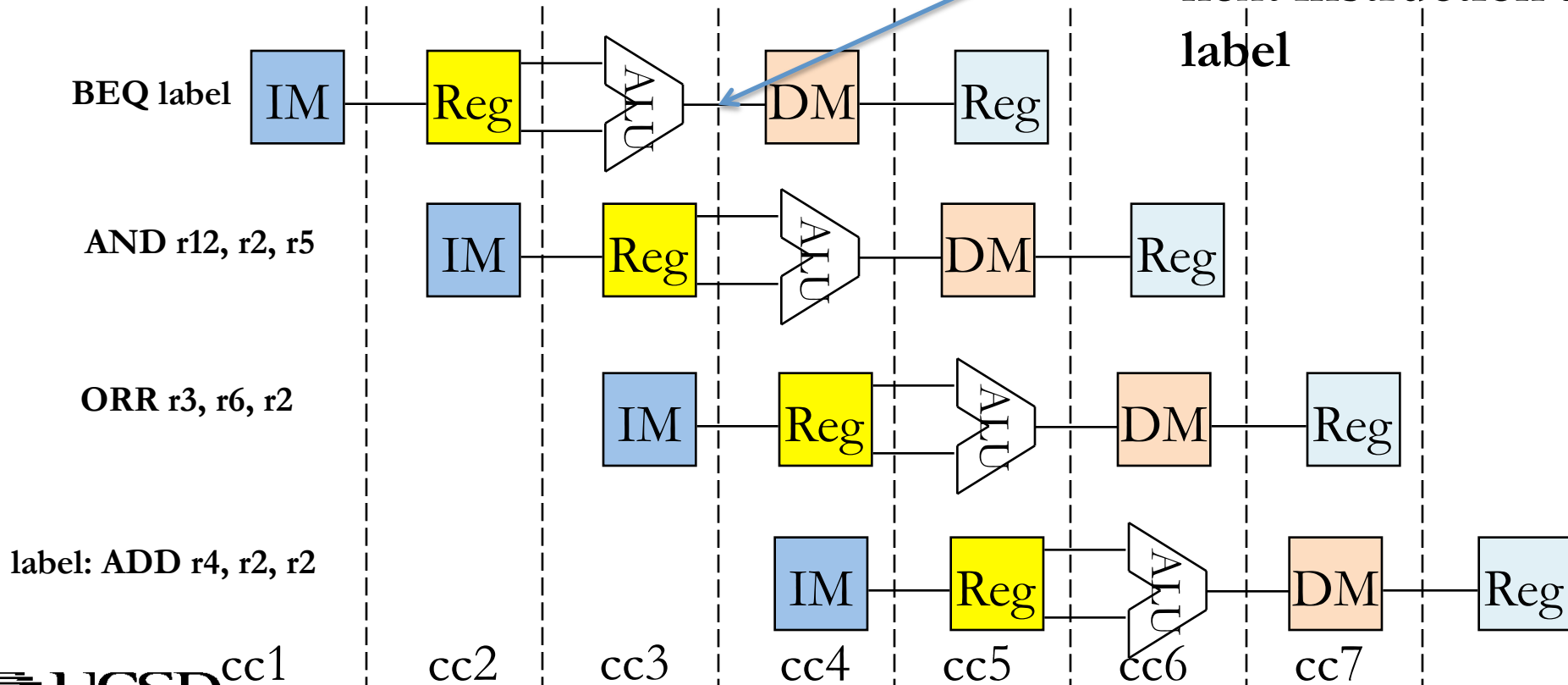
Control Hazards

In the third clock cycle (CC 3), we know we have encountered a branch.
In CC3, what is happening in the pipeline?

- A. ADD is being fetched because B was encountered
- B. ORR is being Fetched
- C. AND is being Fetched

It's a branch!

**Need to fetch the
next instruction at
label**



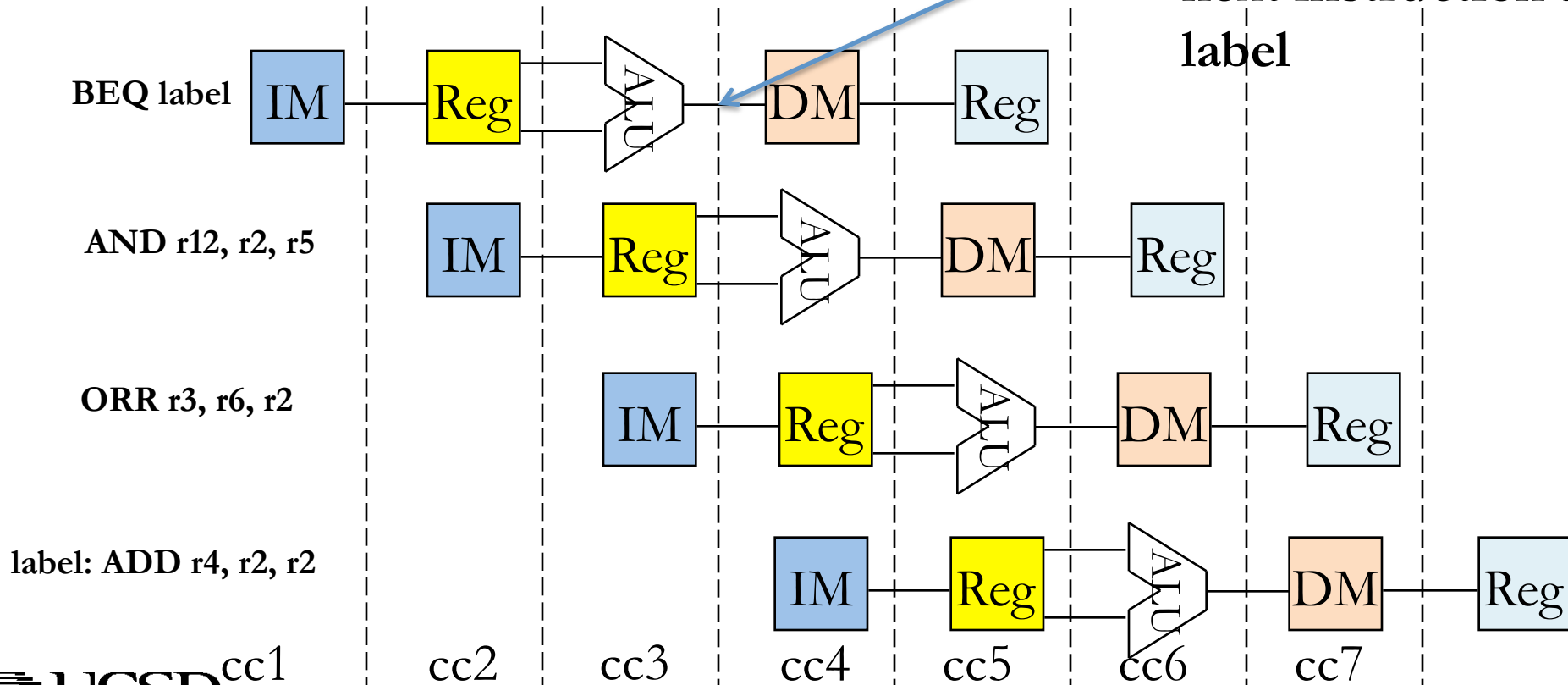
Control Hazards

In the third clock cycle (CC 3), we know we have encountered a branch.
In CC3, what is happening in the pipeline?

- A. ADD is being fetched because B was encountered
- B. ORR is being Fetched, AND decoded**
- C. AND is being Fetched

It's a branch!

**Need to fetch the
next instruction at
label**



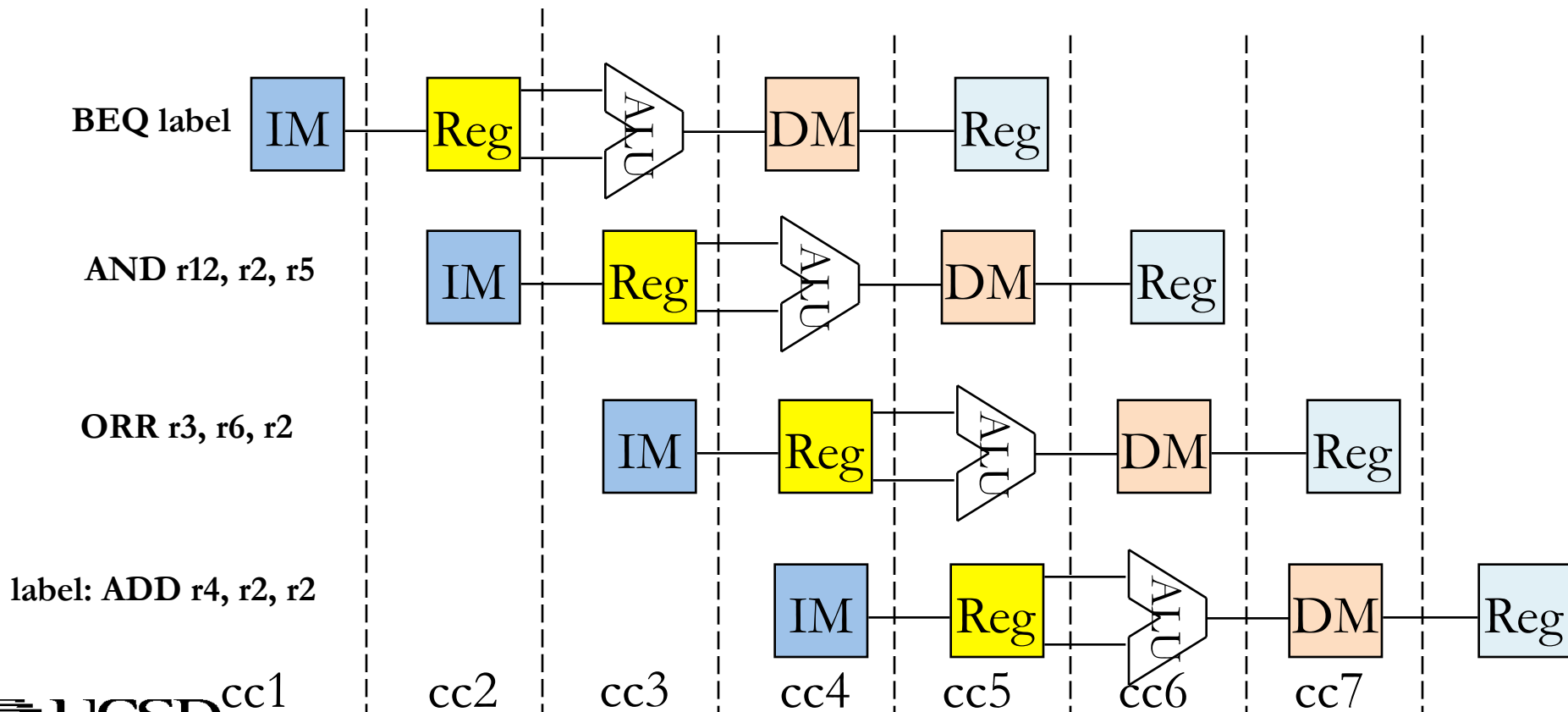
Control Hazards

Q: Which instruction should have been fetched after the branch?

A. AND instruction (the pipeline is not disrupted)

B. ADD instruction (the pipeline is disrupted)

C. Depends



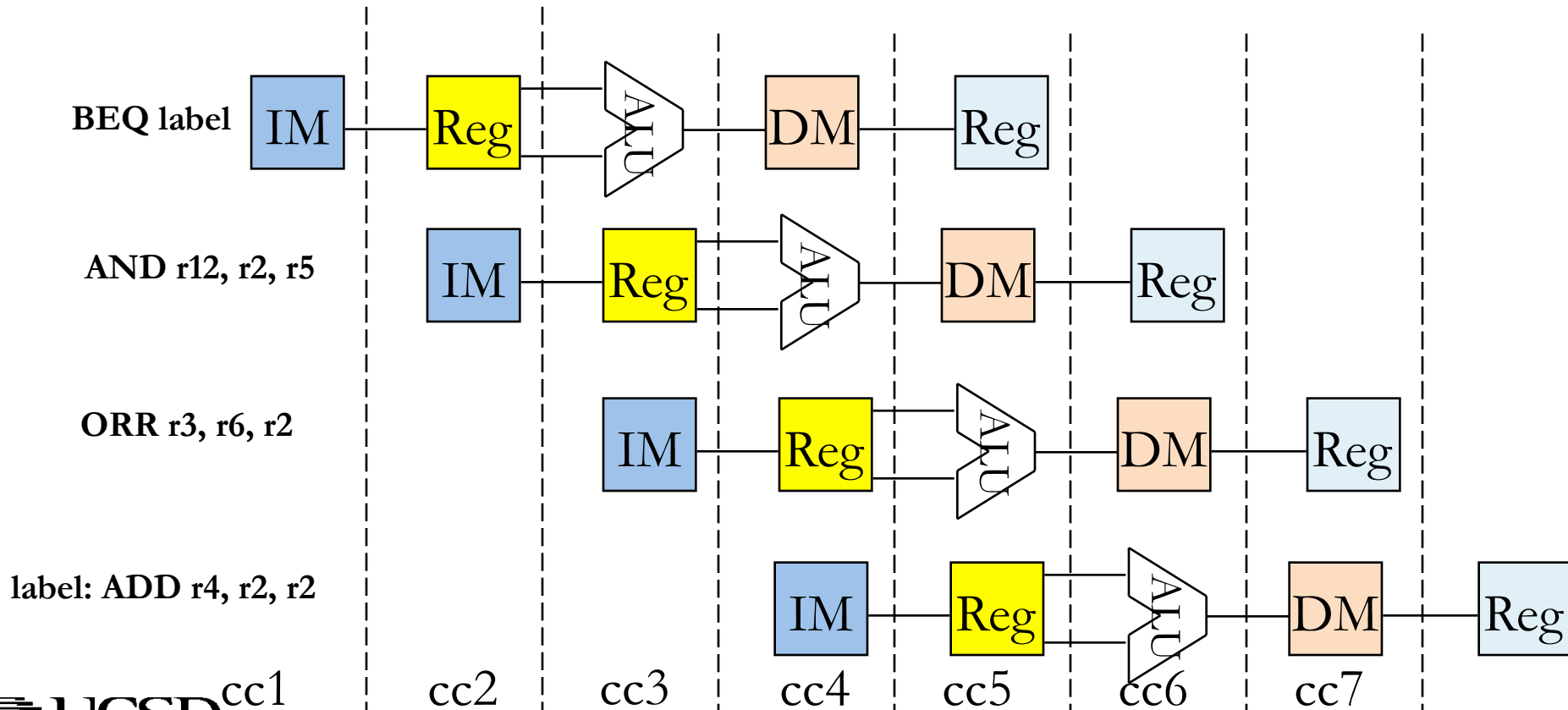
Control Hazards

Q: Which instruction should have been fetched after the branch?

A. AND instruction (the pipeline is not disrupted)

B. ADD instruction (the pipeline is disrupted)

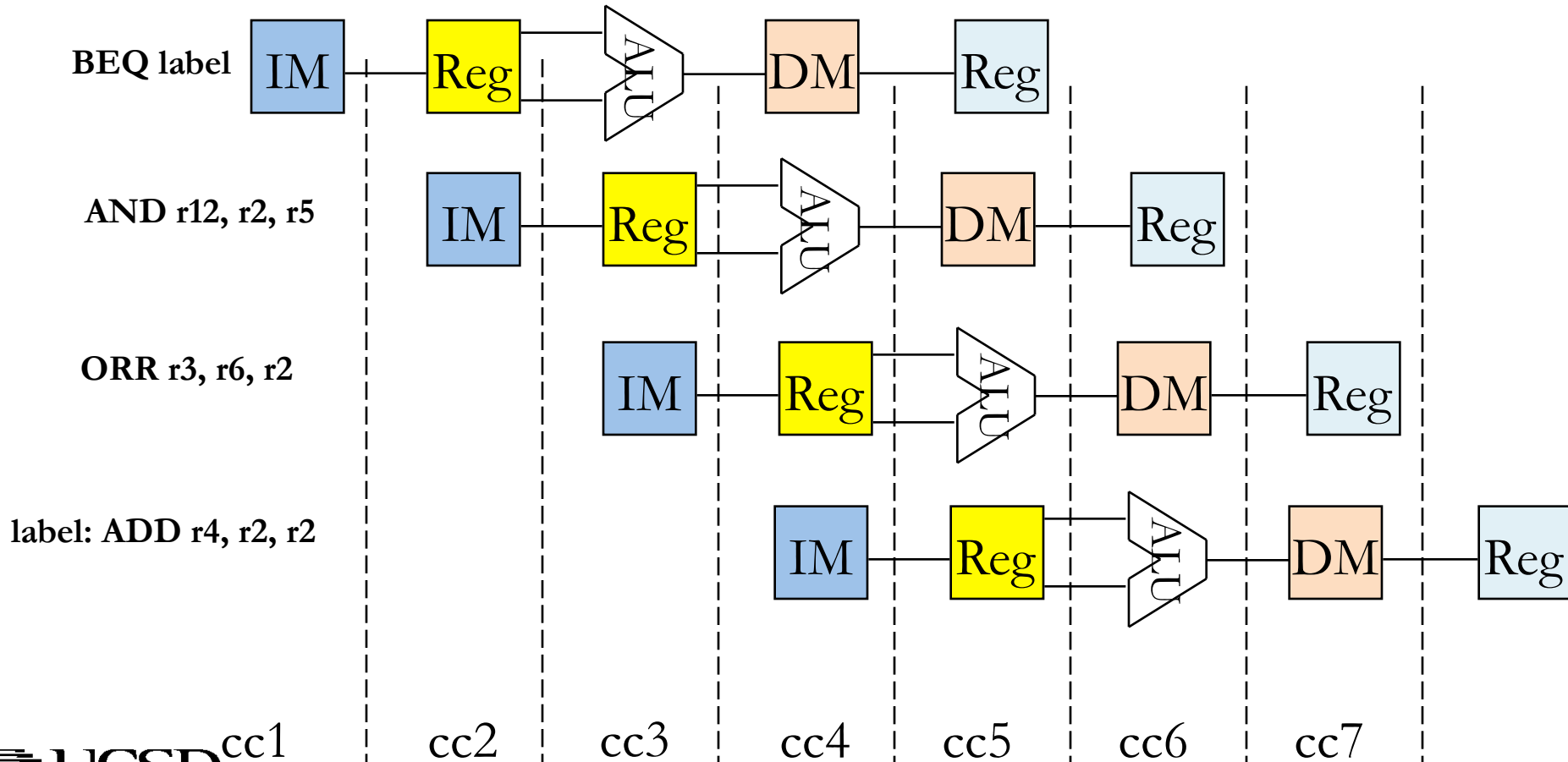
C. Depends on the result of the branch ($Z=1$ or not)



Control Hazards

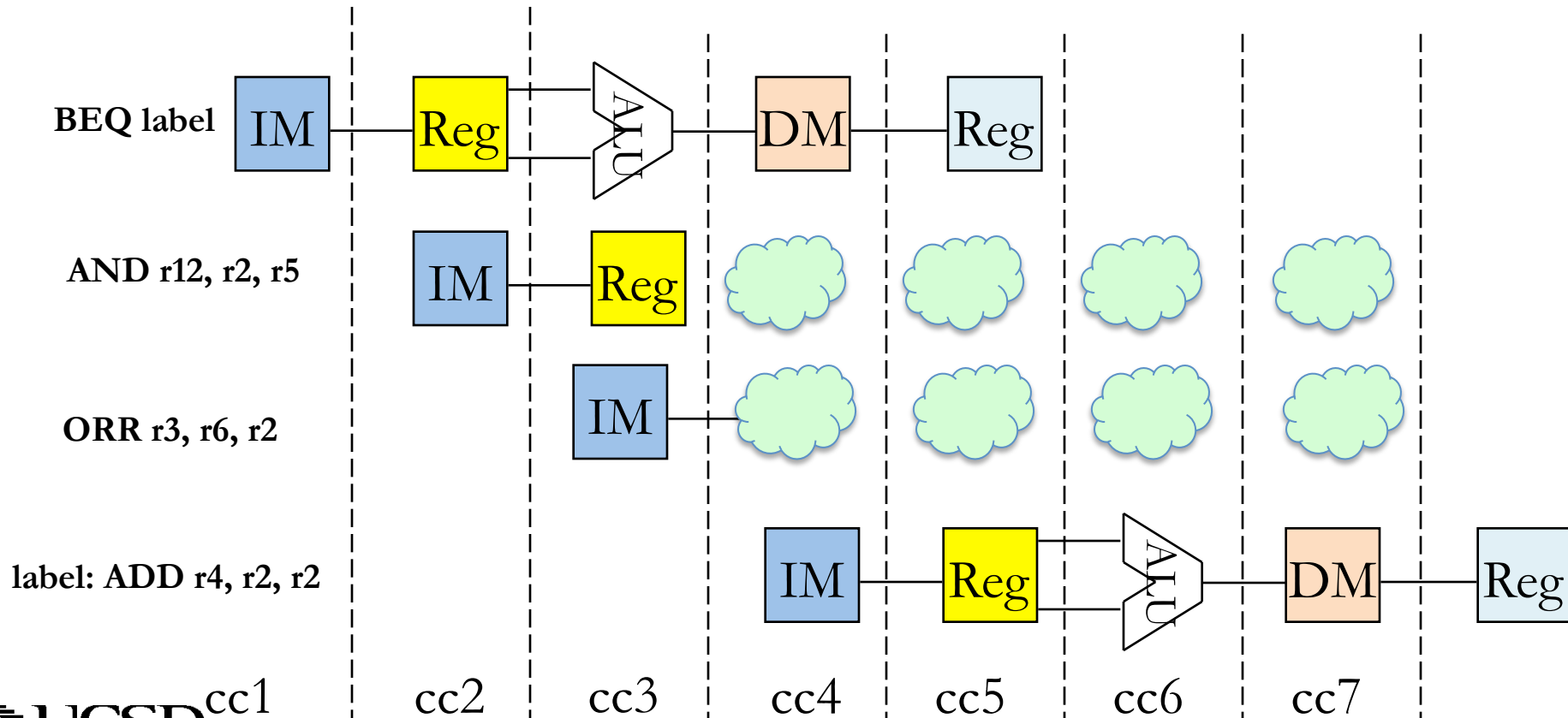
If the Z flag is one, the branch EQ condition is true

- So, AND and ORR incorrectly entered the pipeline
- This is a **Control Hazard**



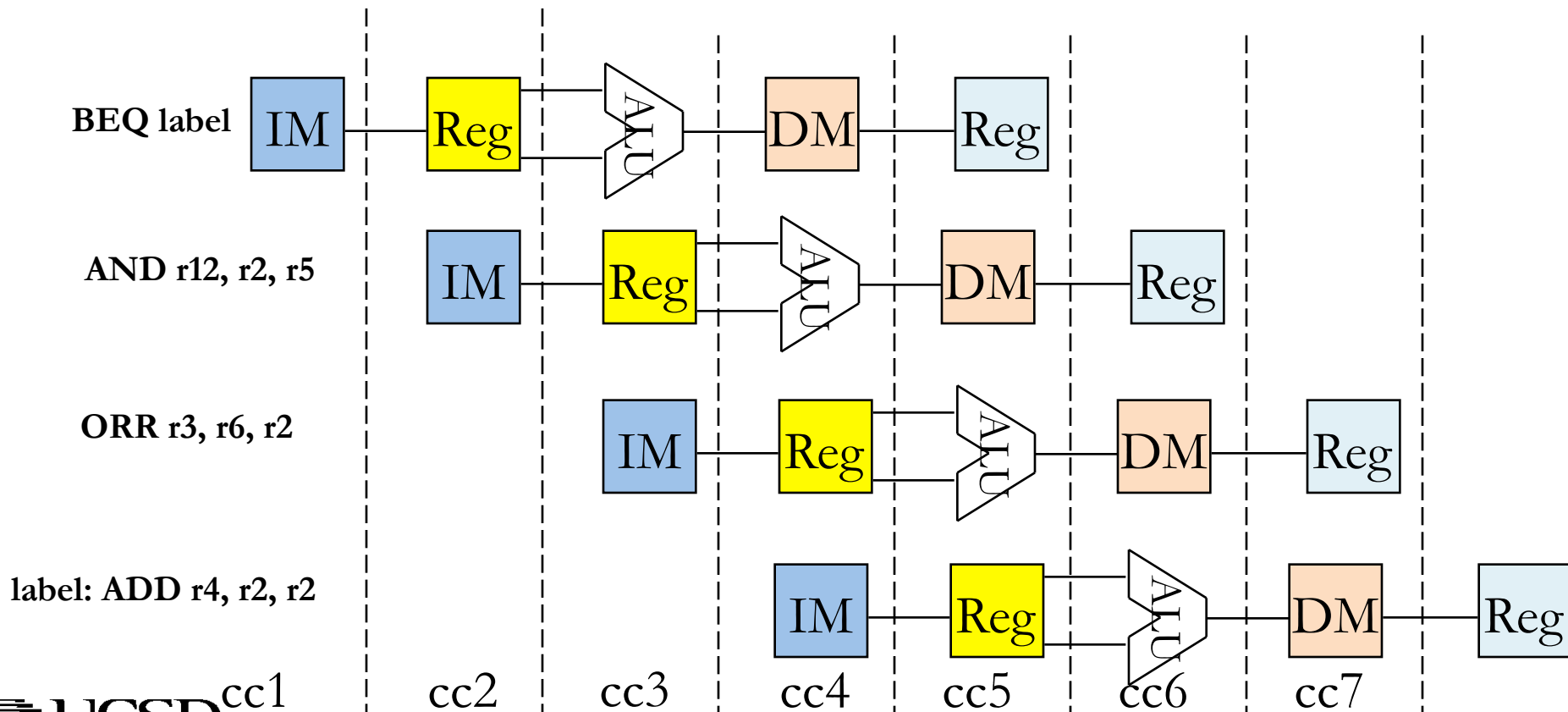
Dealing with Control Hazards

1. One way is to cancel the instructions that were incorrectly fetched in cc2 and cc3, sending bubbles through the pipeline
2. And fetching the right instruction in CC4.
3. This solution is essentially **Stalling** the Processor
4. Branches are 17% of instructions- Stalling is wasteful



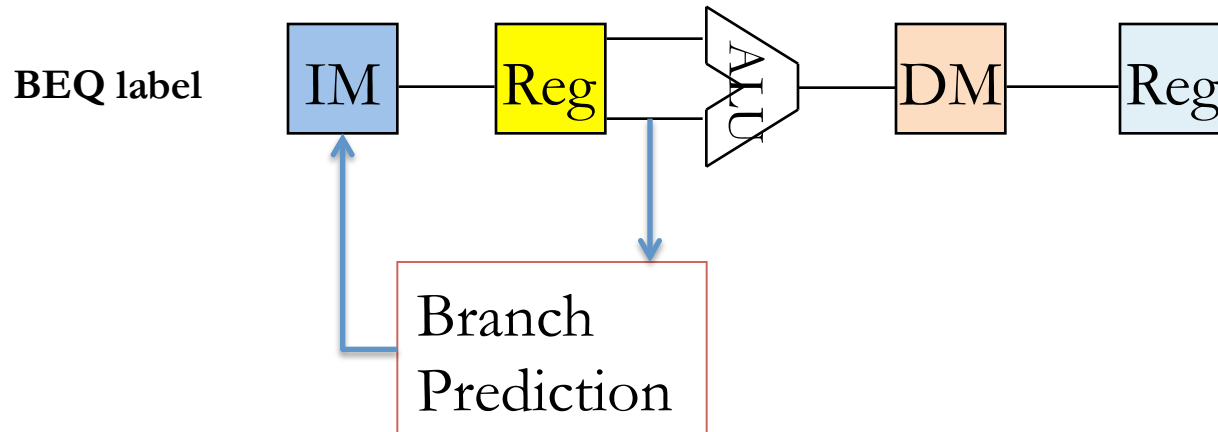
Solutions in Hardware: Choose a default

- Make a default assumption about the next instruction to be fetched and cancel that instruction if the assumption turns out wrong
- E.g. If Z flag was 0 the branch was harmless



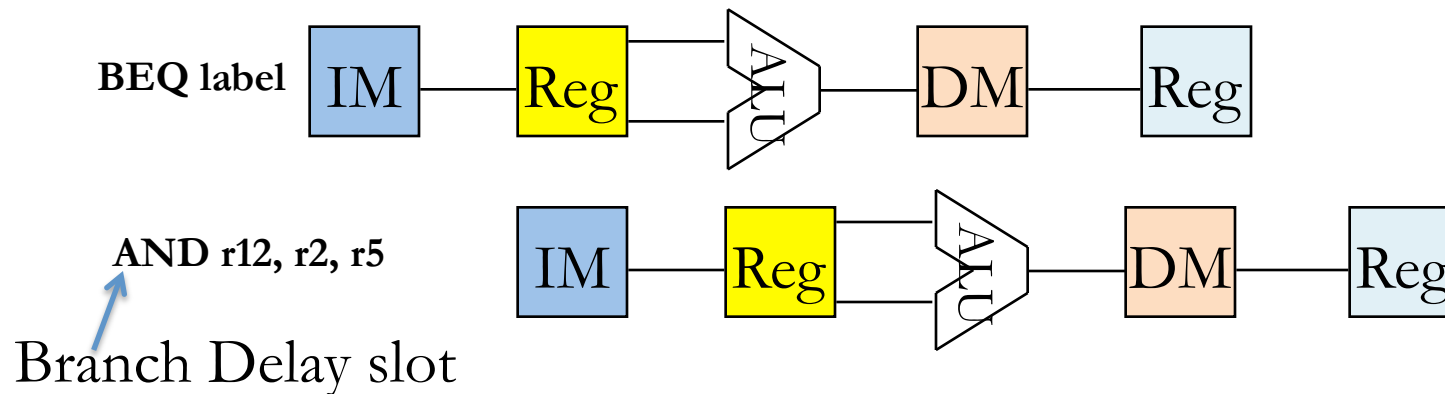
Solutions in H/W: Branch Prediction

- Build a circuit that predicts the outcome of branch earlier (in the decode stage)
 - Only stall if prediction is wrong



Solutions in S/W: Branch Delay slot

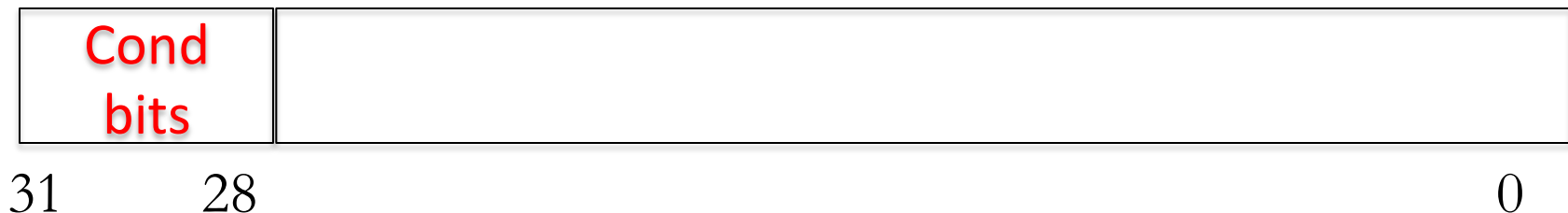
- The h/w always executes the instruction following the branch
- It is up to the compiler to put something useful into that slot (the slot is called Branch Delay slot)



Effect of Branches on pipelines

- Branches generally disrupt the pipeline
- We usually try to avoid them (if possible)
- A mechanism that allows this is by conditionally executing instructions
 - Also called predicated instructions
 - Bits reserved within an instruction to indicate if it is predicated

32 bit instruction



Predicated Instructions

- ❖ All instructions can be executed conditionally. Simply add {EQ,NE,LT,LE,GT,GE, etc.} to end
- ❖ We already used this to conditionally execute the Branch instruction

C source code

```
if (r0 == 0)
{
    r1 = r1 + 1;
}
else
{
    r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
CMP r0, #0
BNE else
ADD r1, r1, #1
B end
else
    ADD r2, r2, #1
end
...
```

conditional

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

- 3 instructions
- 3 words
- 3 cycles

Exercise

- Translate the following code to assembly

```
while (a!=b) {  
    if (a>b)  
        a = a-b;  
    else  
        b=b-a;  
}
```

Assume a: r0, b: r1

```

Loop  CMP r0, r1;    Statement 1
      BEQ end;       Statement 2
      BNEQ less;     Statement 3
      SUB r0,r0, r1; Statement 4
Less  SUBS r1, r1, r0; Statement 5
      B Loop;        Statement 6
End

```

```

C code:
while (r0!=r1) {
    if (r0>r1)
        r0 = r0-r1;
    else
        r1=r1-r0;
}

```

What is wrong with the following translation?

- A. There is a statement missing after statement 4
- B. Statement 4 is never executed
- C. Both A & B
- D. No errors

Loop	CMP r0, r1;	Statement 1
	BEQ end;	Statement 2
	BNEQ less;	Statement 3
	SUB r0,r0, r1;	Statement 4
Less	SUBS r1, r1, r0;	Statement 5
	B Loop;	Statement 6
End		

C code:

```
while (r0!=r1) {
    if (r0>r1)
        r0 = r0-r1;
    else
        r1=r1-r0;
}
```

What is wrong with the following translation?

- A. There is a statement missing after statement 4
- B. Statement 4 is never executed
- C. Both A & B**
- D. No errors

Rewrite the code to minimize the number of branches

```
Loop  CMP r0, r1;    Statement 1
      BEQ end;        Statement 2
      BLT less;       Statement 3
      SUB r0,r0, r1;  Statement 4
      B Loop
Less  SUB r1, r1, r0;  Statement 5
      B Loop;         Statement 6
End
```

```
Loop  CMP r0, r1;
      BEQ end;
      BNEQ less;
      SUB r0,r0, r1;
Less  SUBS r1, r1, r0;
      B Loop;
End
```

Rewrite the code to minimize the number of branches

```
Loop  CMP r0, r1;   Statement 1
      BEQ end;      Statement 2
      BLT less;     Statement 3
      SUB r0,r0, r1; Statement 4
      B Loop
Less  SUB r1, r1, r0; Statement 5
      B Loop;       Statement 6
End
```

```
Loop  CMP r0, r1
      SUBGT r0, r0, r1
      SUBLT r1, r1, r0
      BNEQ Loop
```