ECE 2300
Digital Logic & Computer Organization

Fall 2016

Advanced Topics

Cornell University

Parallelism: Making our Processor *Fast*

- Processor architects improve performance through hardware that exploits the different types of *parallelism* within computer programs

- Instruction-Level Parallelism (ILP)
  - Parallelism within a sequential program
- Thread-level parallelism (TLP)
  - Parallelism among different threads
- Data-level parallelism (DLP)
  - Parallelism among the data within a sequential program

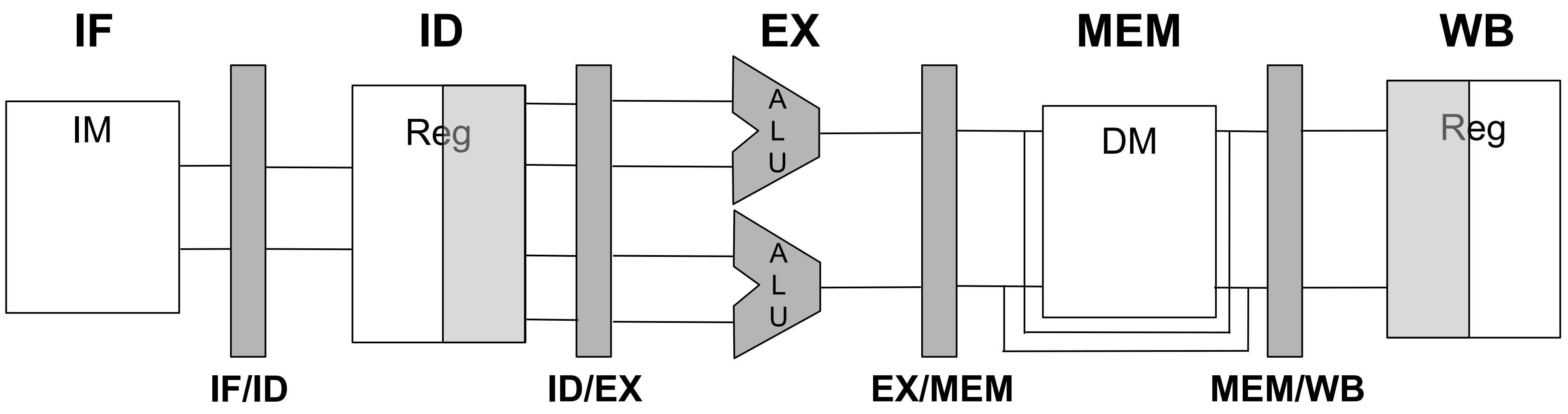
Instruction-Level Parallelism (ILP)

- Refers to the parallelism found within a sequential program
- Consider the ILP in this program segment

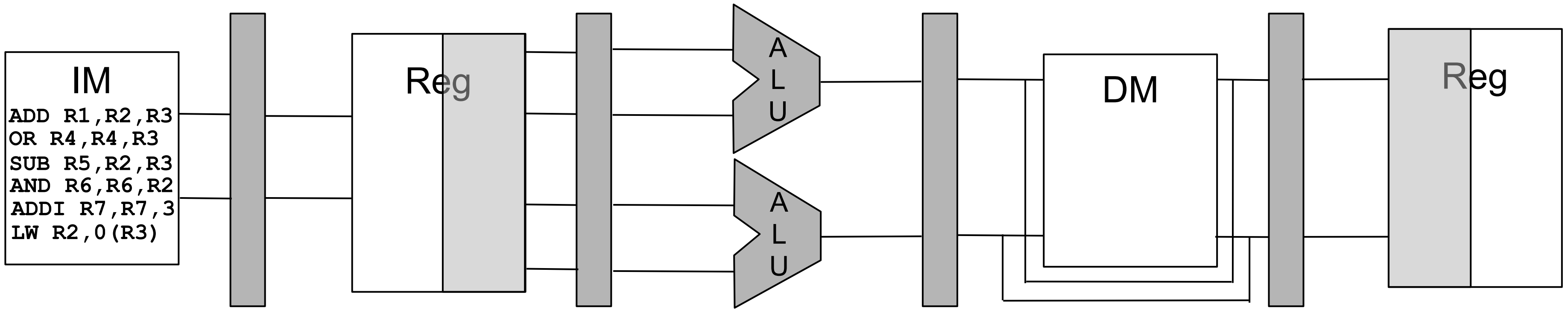
```
ADD R1,R2,R3
OR R4,R4,R3
SUB R5,R2,R3
AND R6,R6,R2
ADDI R7,R7,3
LW R2,0(R3)
```

- *Superscalar pipelines* exploit ILP by duplicating the pipeline hardware

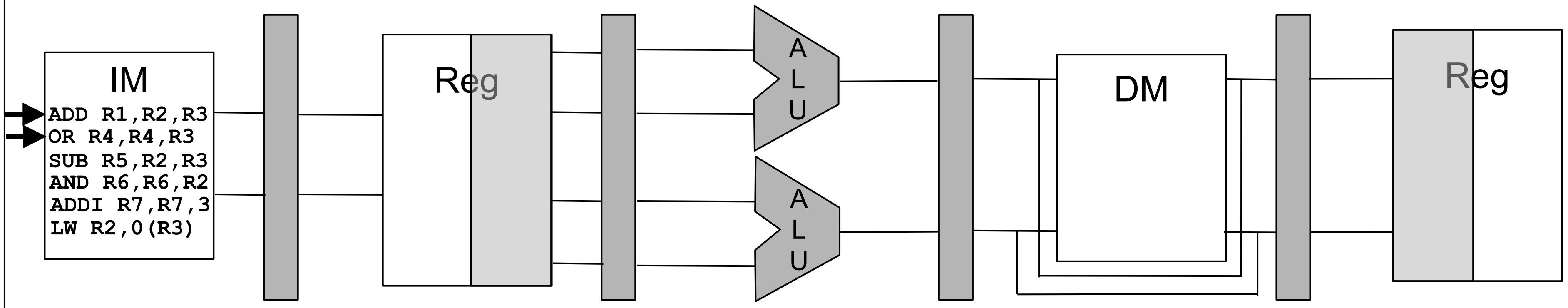
Two-Way Superscalar Pipeline



Instruction Sequence on 2W SS

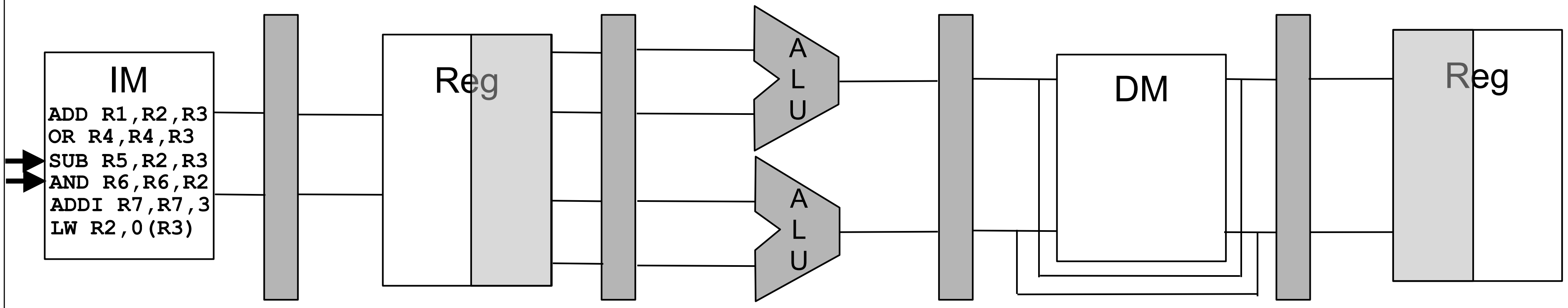


Instruction Sequence on 2W SS



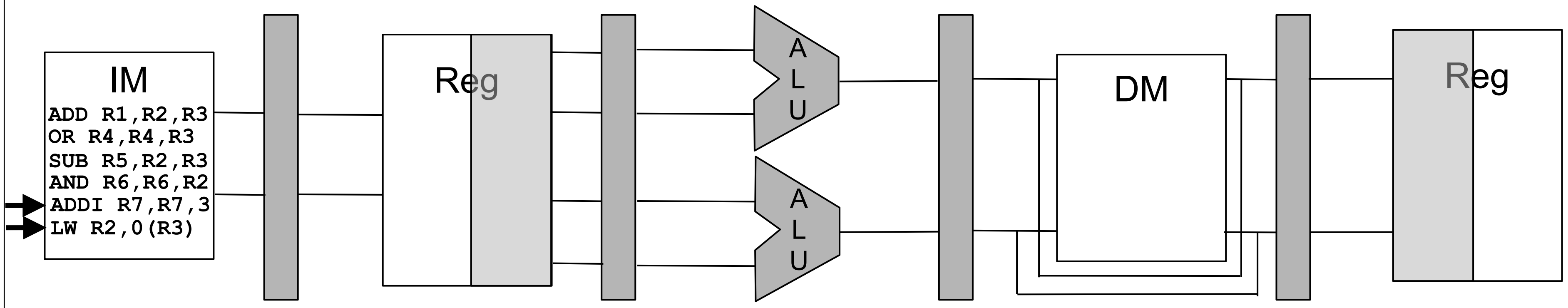
ADD R1,R2,R3
OR R4,R4,R3

Instruction Sequence on 2W SS



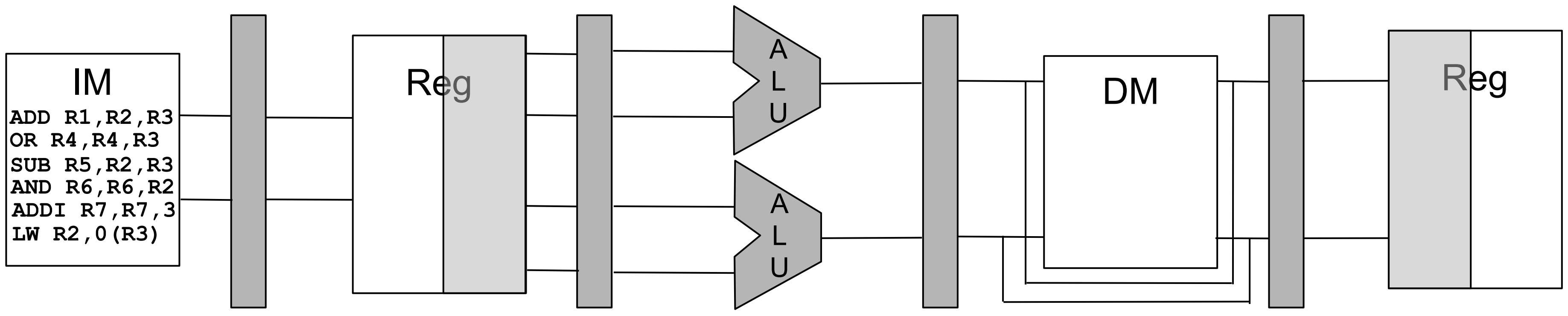
```
SUB R5,R2,R3     ADD R1,R2,R3
AND R6,R6,R2     OR R4,R4,R3
```

Instruction Sequence on 2W SS



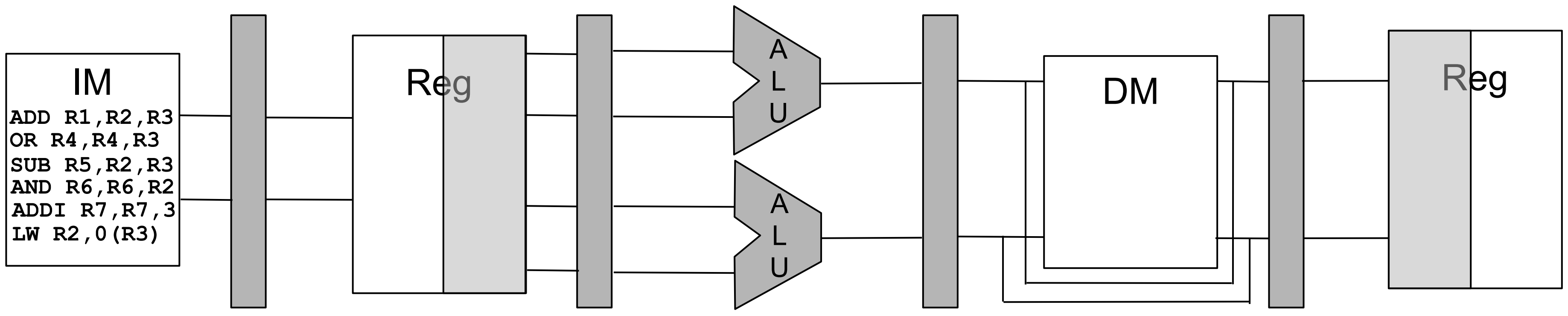
```
ADDI R7,R7,3     SUB R5,R2,R3     ADD R1,R2,R3
LW R2,0(R3)      AND R6,R6,R2     OR R4,R4,R3
```

Instruction Sequence on 2W SS



```
ADDI R7,R7,3     SUB R5,R2,R3     ADD R1,R2,R3
LW R2,0(R3)      AND R6,R6,R2     OR R4,R4,R3
```

Instruction Sequence on 2W SS



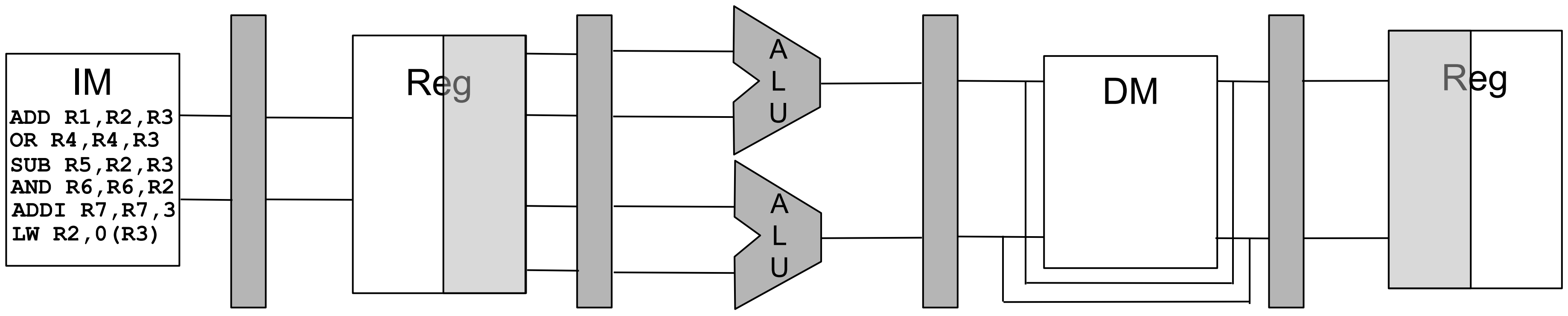
```
ADD R1,R2,R3
OR R4,R4,R3
SUB R5,R2,R3
AND R6,R6,R2
ADDI R7,R7,3
LW R2,0(R3)
```

```
ADDI R7,R7,3
LW R2,0(R3)
```

```
SUB R5,R2,R3
AND R6,R6,R2
```

```
ADD R1,R2,R3
OR R4,R4,R3
```

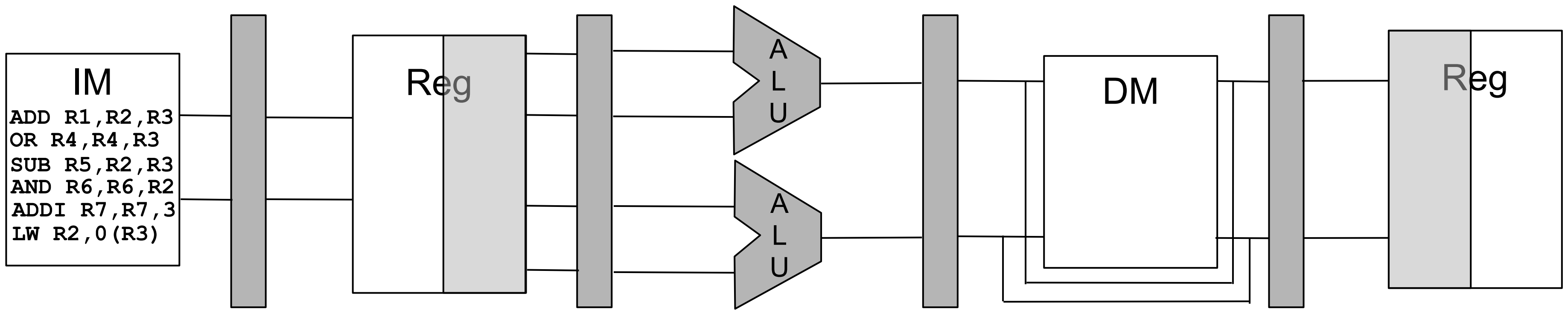
Instruction Sequence on 2W SS



```
ADD R1,R2,R3
OR R4,R4,R3
SUB R5,R2,R3
AND R6,R6,R2
ADDI R7,R7,3
LW R2,0(R3)
```

```
ADDI R7,R7,3     SUB R5,R2,R3
LW R2,0(R3)      AND R6,R6,R2
```

Instruction Sequence on 2W SS



```
ADDI R7,R7,3
LW R2,0(R3)
```

Limitations Due to Data Dependences

- Consider this program sequence

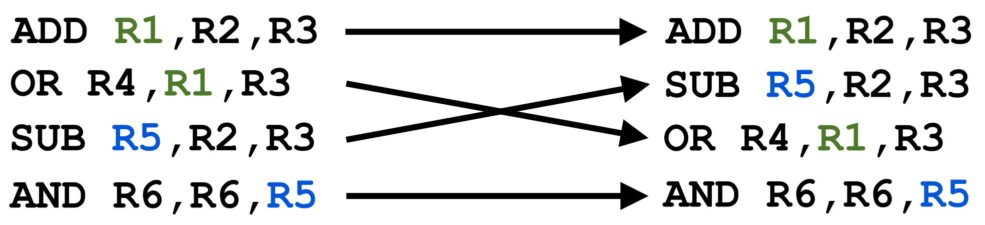
```
ADD R1,R2,R3
OR R4,R1,R3
SUB R5,R2,R3
AND R6,R6,R5
```

- The ADD and the OR, and the SUB and AND, cannot execute at the same time
  – Hardware would detect these dependences and issue the instructions (send them to EX) one by one

- Addressed by hardware that performs *out-of-order execution*

Out-of-Order Execution

- Processor hardware executes instructions out of the original program order



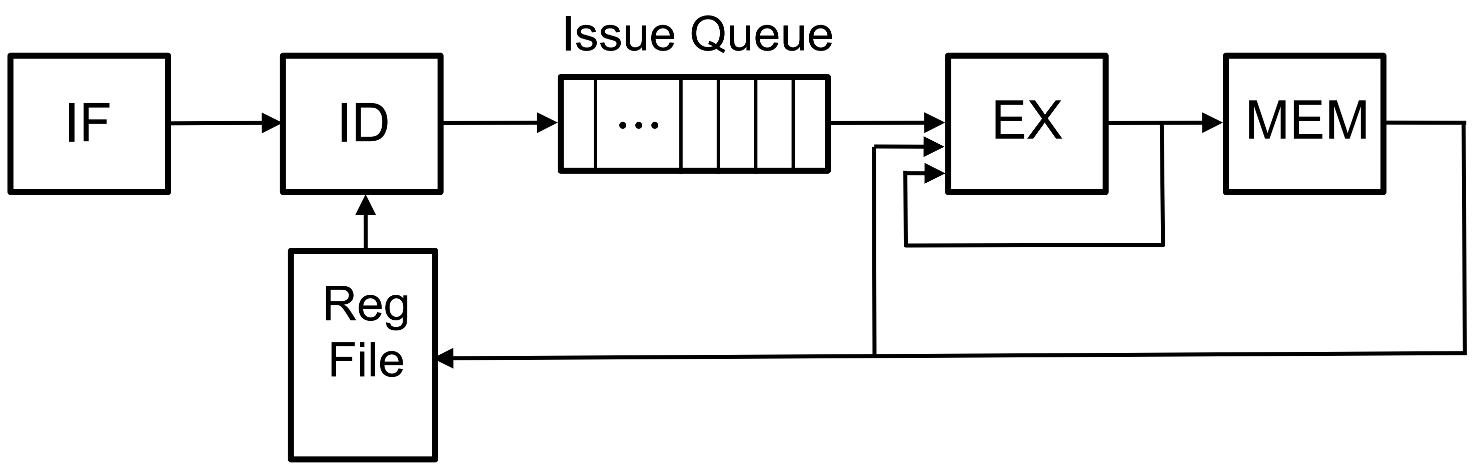
```
ADD R1,R2,R3
OR R4,R1,R3
SUB R5,R2,R3
AND R6,R6,R5
```

```
ADD R1,R2,R3
SUB R5,R2,R3
OR R4,R1,R3
AND R6,R6,R5
```

- Key component is an *issue queue* that tracks the availability of source operands

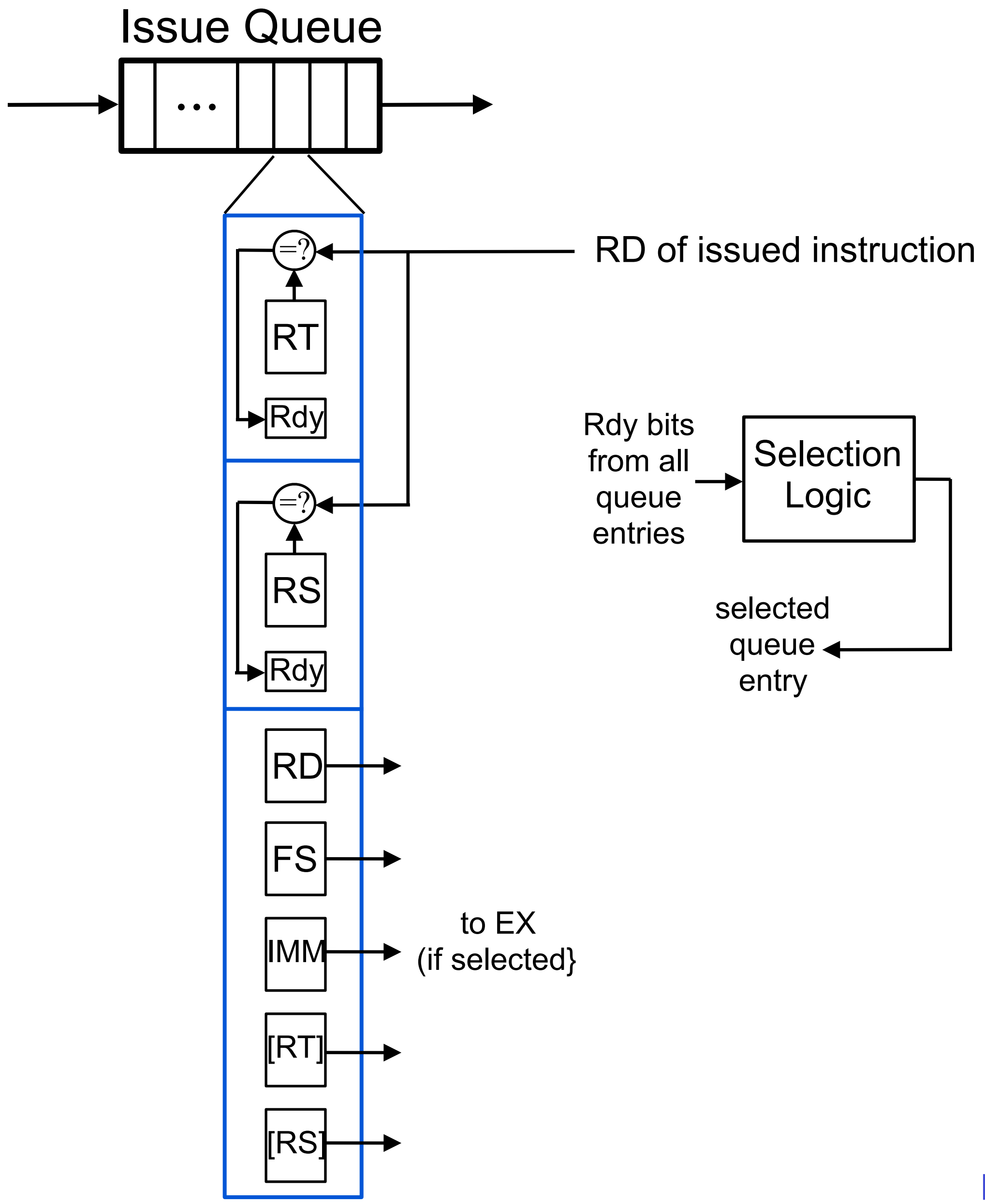
Issue Queue

- Hardware structure where instructions wait until source operands can be bypassed

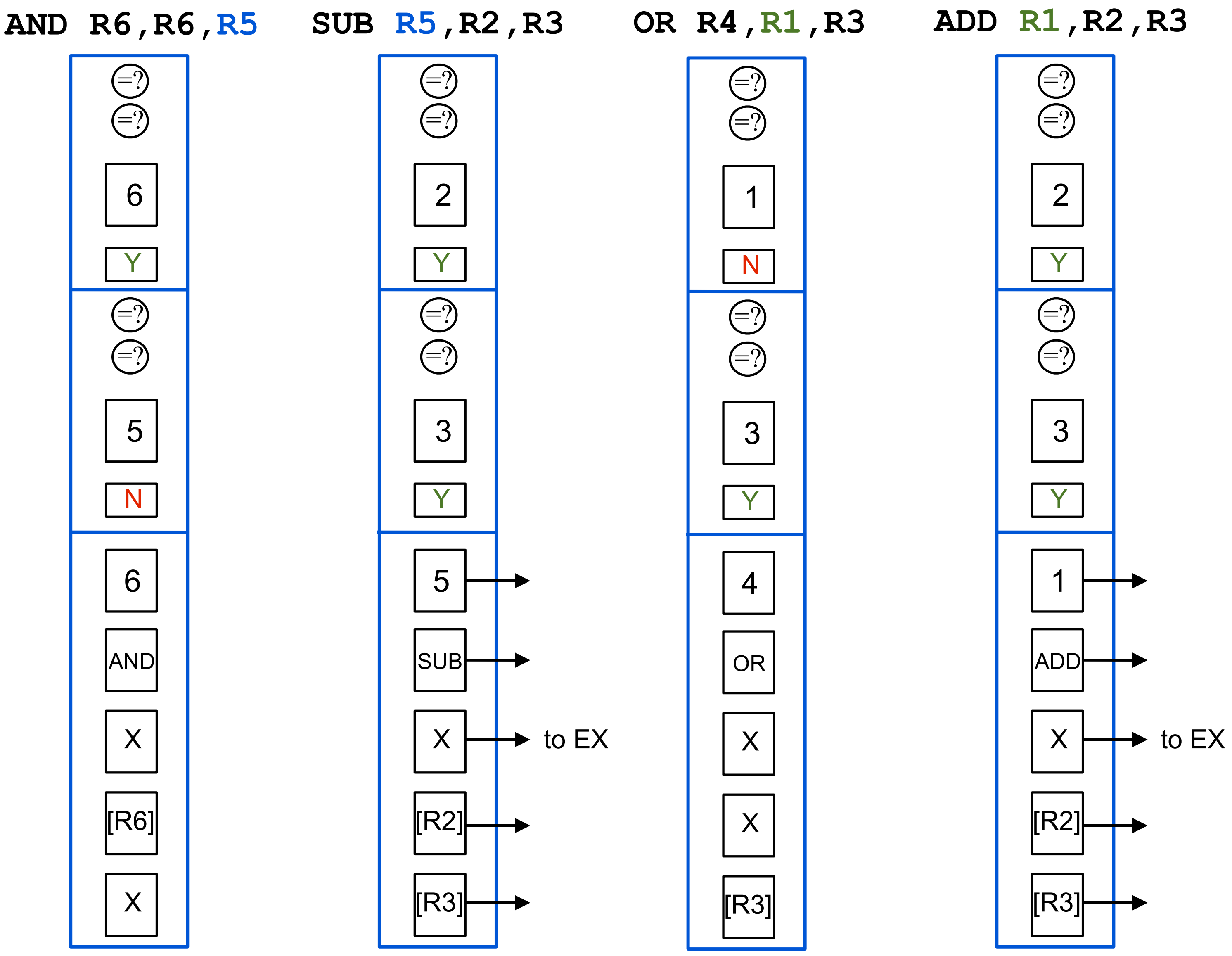


Issue Queue Structure (Single Issue)

```
ADD R1,R2,R3
OR R4,R1,R3
SUB R5,R2,R3
AND R6,R6,R5
```

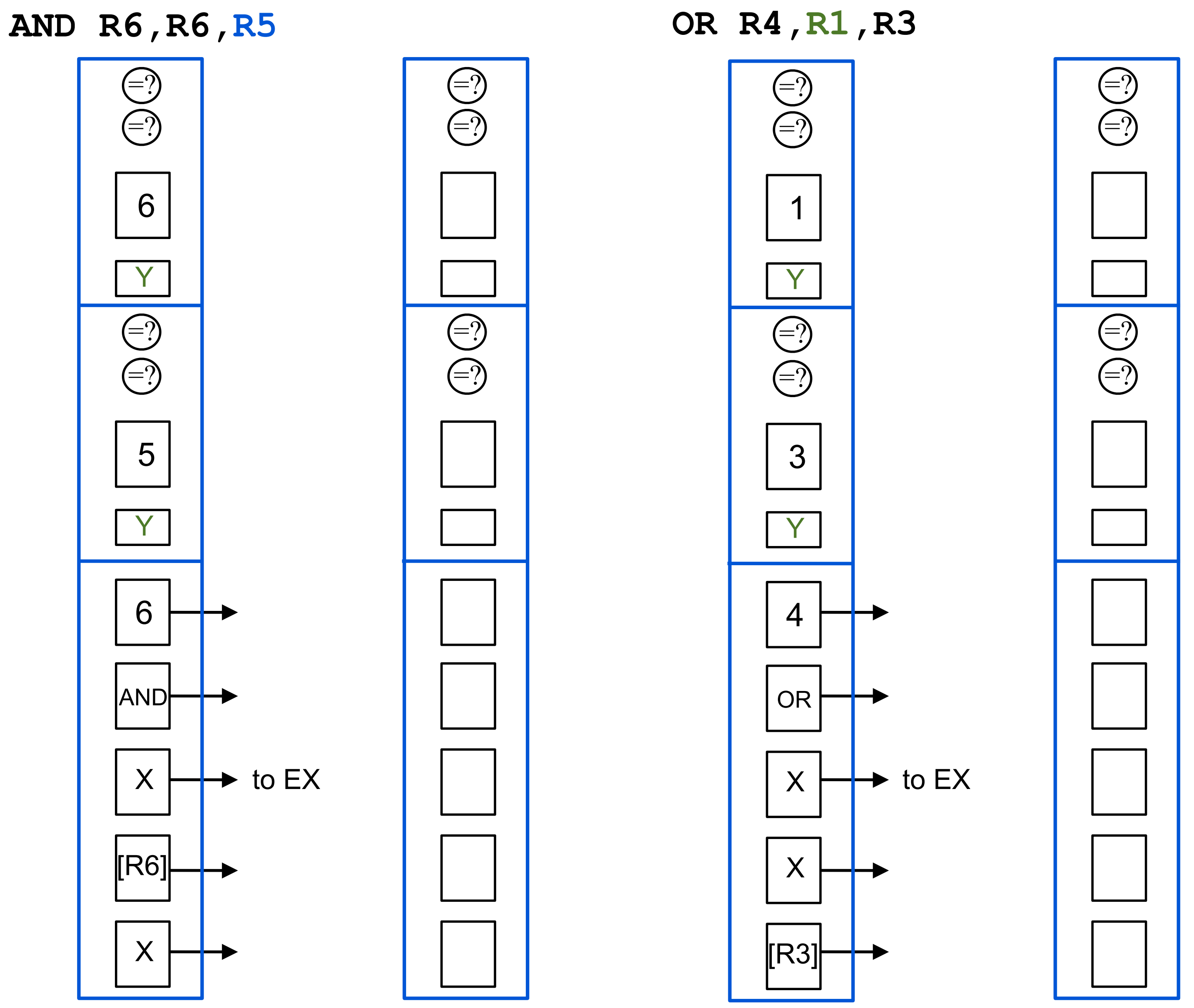


Example Dual Issue Queue Operation



Cycle 1

Example Dual Issue Queue Operation



Cycle 2

A More Troublesome Piece of Code

- Now consider this program sequence

```
Loop  ADD R1,R2,R3
      OR R4,R1,R3
      SUB R5,R4,R3
      AND R1,R6,R5
      BEQ R2,R1,Loop
```

- Superscalar pipeline would send instructions one by one through EX, MEM, and WB
  - 1 ALU, 1 memory port, and 1 RF port would sit idle, perhaps through 1000 loop iterations!
- But what if we had a *second* program to run?

Multiprogramming and Multithreading

- Multiprogramming is the sharing of the computer hardware among multiple active programs
- Programs may be *time multiplexed*, in which they take turns using the processor
- Or they can use the hardware at the same time!
  - Using multiple *cores* (next time)
  - Using the same core at the same time
    - For multiple simultaneously running programs, or multiple *threads* of the same program

Thread-Level Parallelism (TLP)

- Refers to the parallelism among different *threads*
  - A thread is a path of execution within a program
  - Each uses its own registers but they share memory

- Consider two threads that we want to run

Thread 1

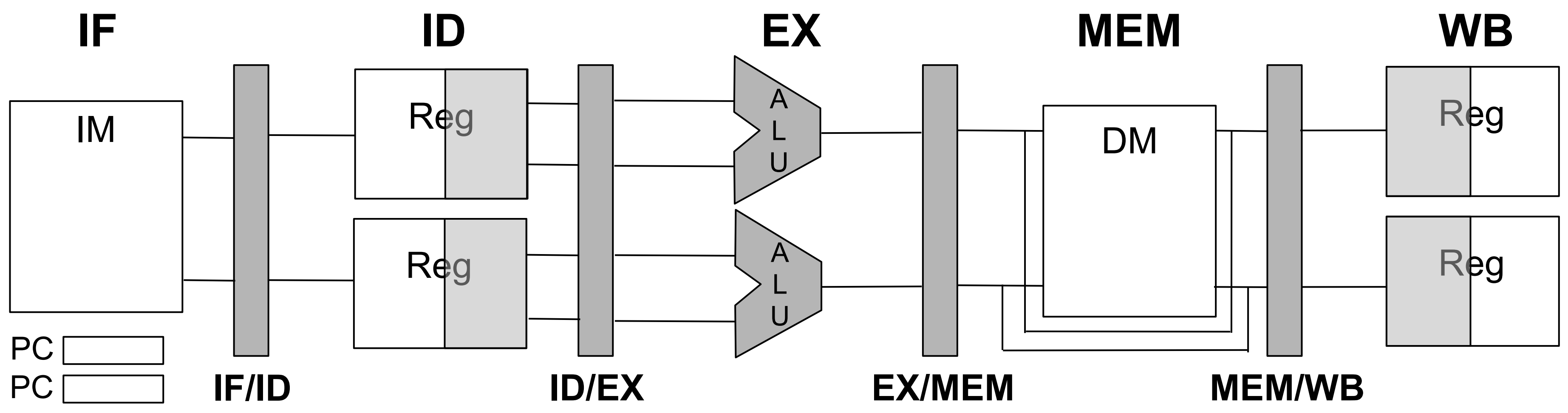
```
Loop   ADD R1,R2,R3
       OR R4,R1,R3
       SUB R5,R4,R3
       AND R1,R6,R5
       BEQ R2,R1,Loop
```

Thread 2

```
LW R7,0(R1)
ADD R4,R7,R2
SUBI R5,R4,1
SW R5,R1,0
ADDI R1,R1,1
```

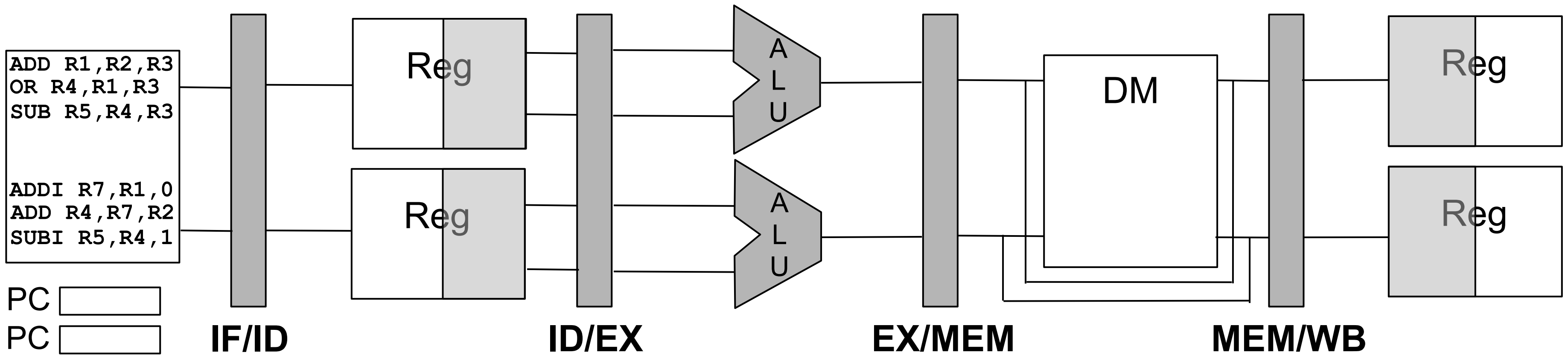
- We can run them on separate cores (later), or create a superscalar pipeline that can run them both at the same time

Two-Way *Multithreaded* Pipeline

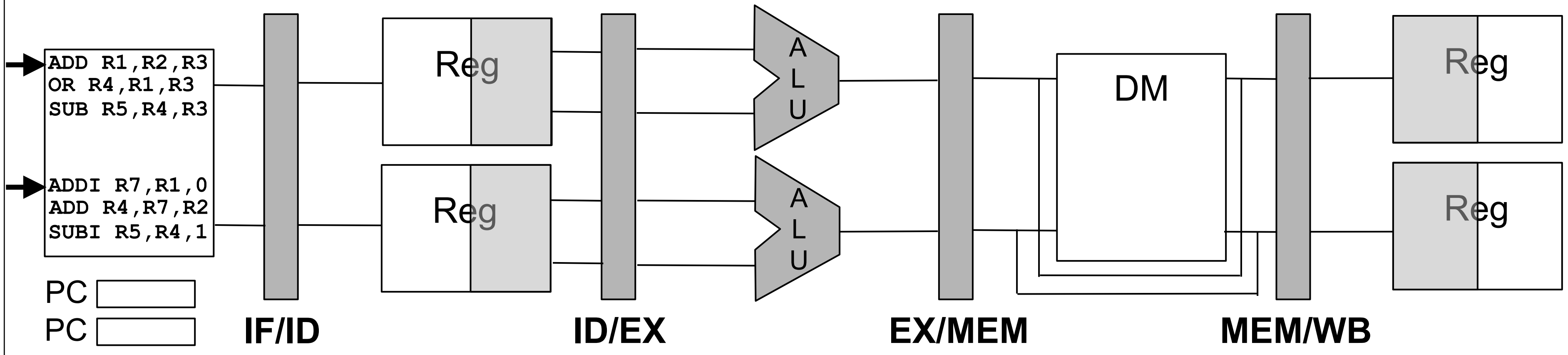


- Two threads share many of the pipeline resources
- Each thread has its own PC and registers
- This is one example of multithreading, called *Simultaneous Multithreading (SMT)*

Example Multithreaded Operation

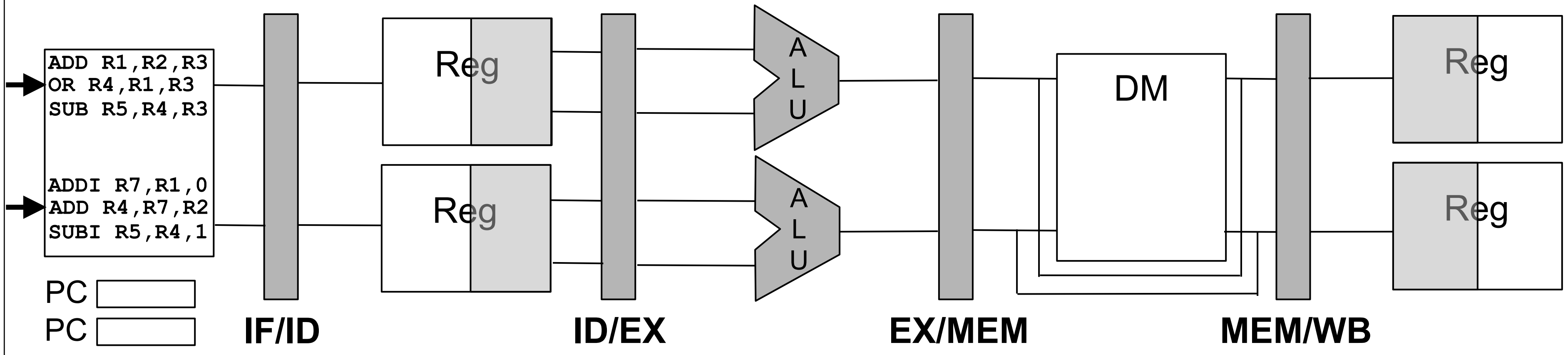


Example Multithreaded Operation



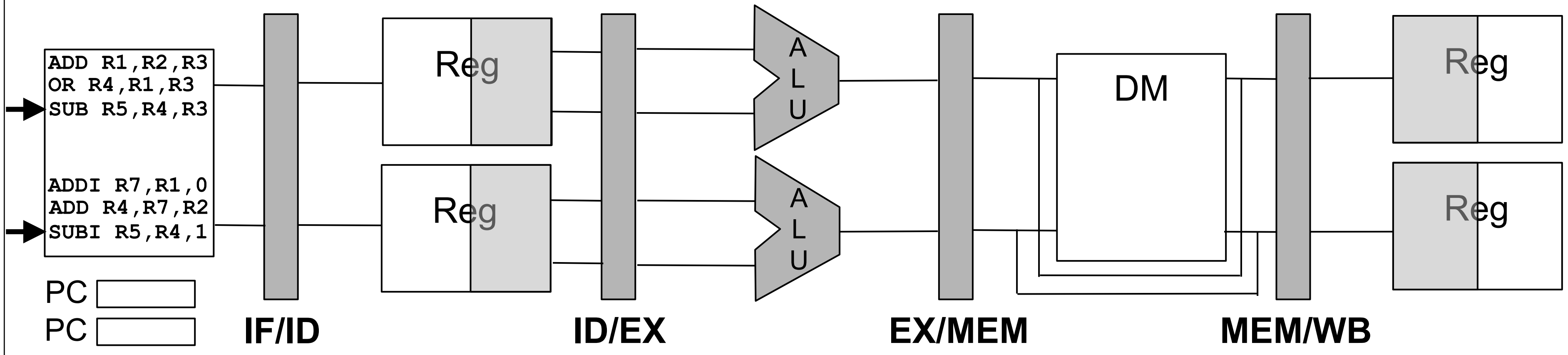
```
ADD R1,R2,R3
ADDI R7,R1,0
```

Example Multithreaded Operation



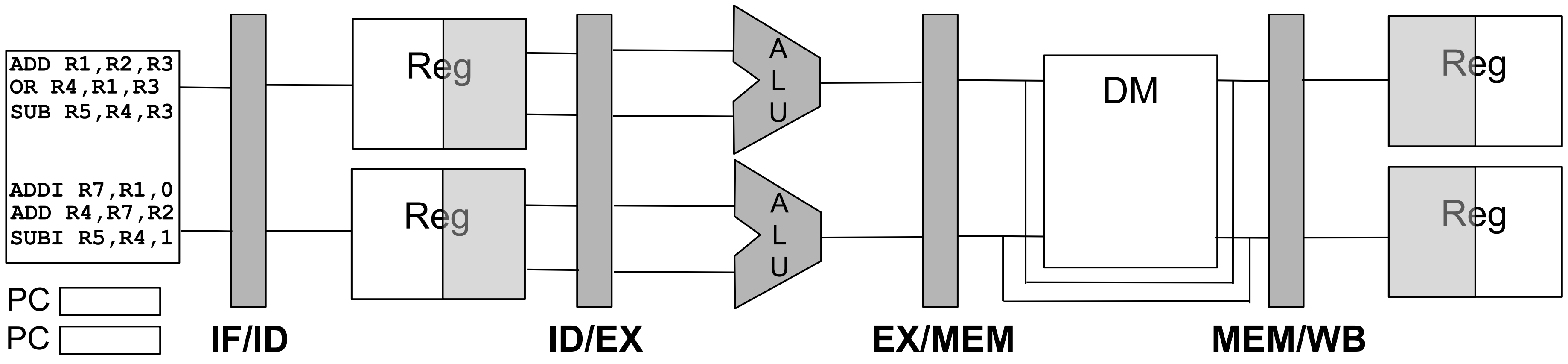
```
OR R4,R1,R3     ADD R1,R2,R3
ADD R4,R7,R2    ADDI R7,R1,0
```

Example Multithreaded Operation



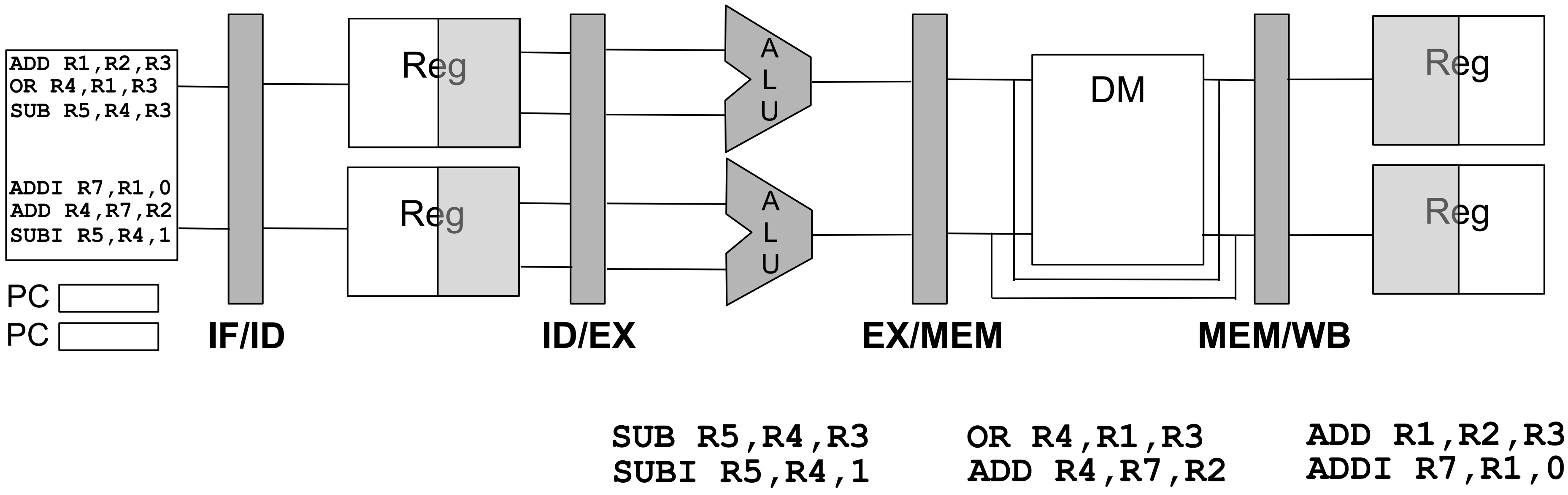
```
SUB R5,R4,R3      OR R4,R1,R3       ADD R1,R2,R3
SUBI R5,R4,1      ADD R4,R7,R2      ADDI R7,R1,0
```

Example Multithreaded Operation

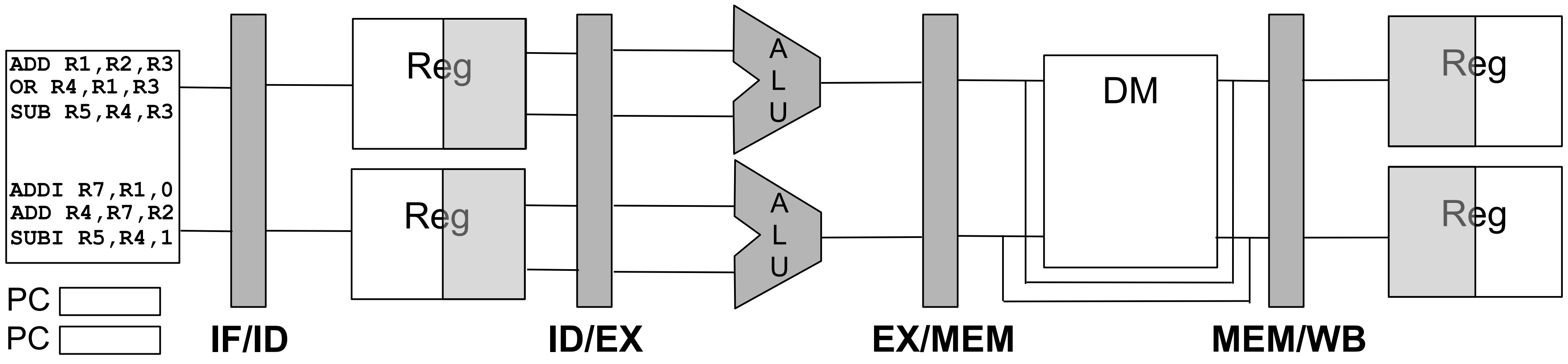


```
SUB R5,R4,R3     OR R4,R1,R3      ADD R1,R2,R3
SUBI R5,R4,1     ADD R4,R7,R2     ADDI R7,R1,0
```

Example Multithreaded Operation

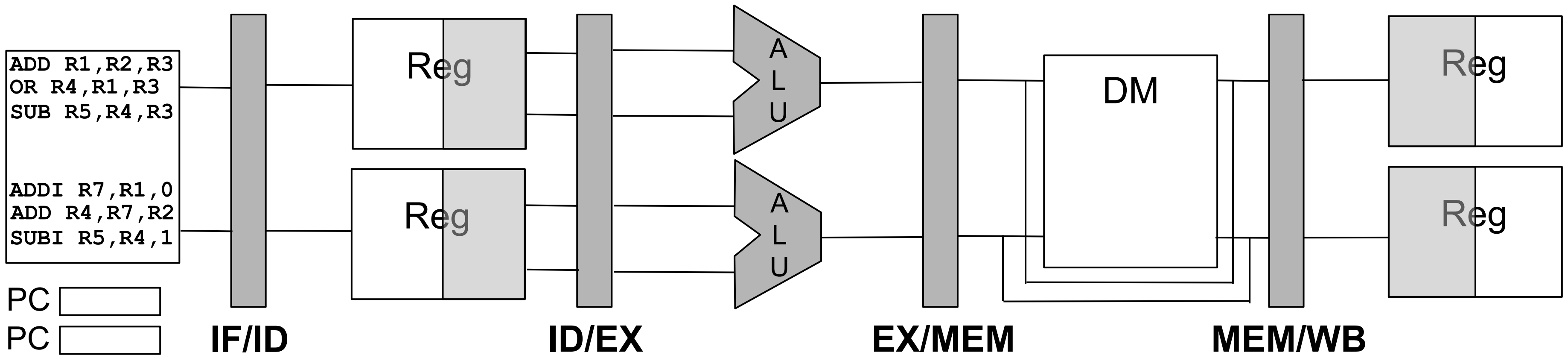


Example Multithreaded Operation



```
SUB R5,R4,R3      OR R4,R1,R3
SUBI R5,R4,1      ADD R4,R7,R2
```

Example Multithreaded Operation



```
SUB R5,R4,R3
SUBI R5,R4,1
```

Data-Level Parallelism

- Consider the following C code

```
char a[4], b[4], c[4];

for (i = 0; i < 4; i++)
  a[i] = b[i] + c[i];
```

- Arrays *a, b, and c* contain four 8-bit elements
  - a[0], a[1], a[2], a[3] for array *a*
- Same operation is done for each data element
- Can replace the 4 add operations in the loop above by 1 *SIMD add* instruction

SIMD Instructions

- *Single Instruction, Multiple Data*
- Special instructions for *vector* data (arrays)
- Identical operation is performed on each of the corresponding data elements
- Data elements are stored contiguously
  - 1 load (store) can read (write) all the elements at once
  - Register file is wide enough to hold all the elements in one register

Implementing the *for* Loop Using SIMD

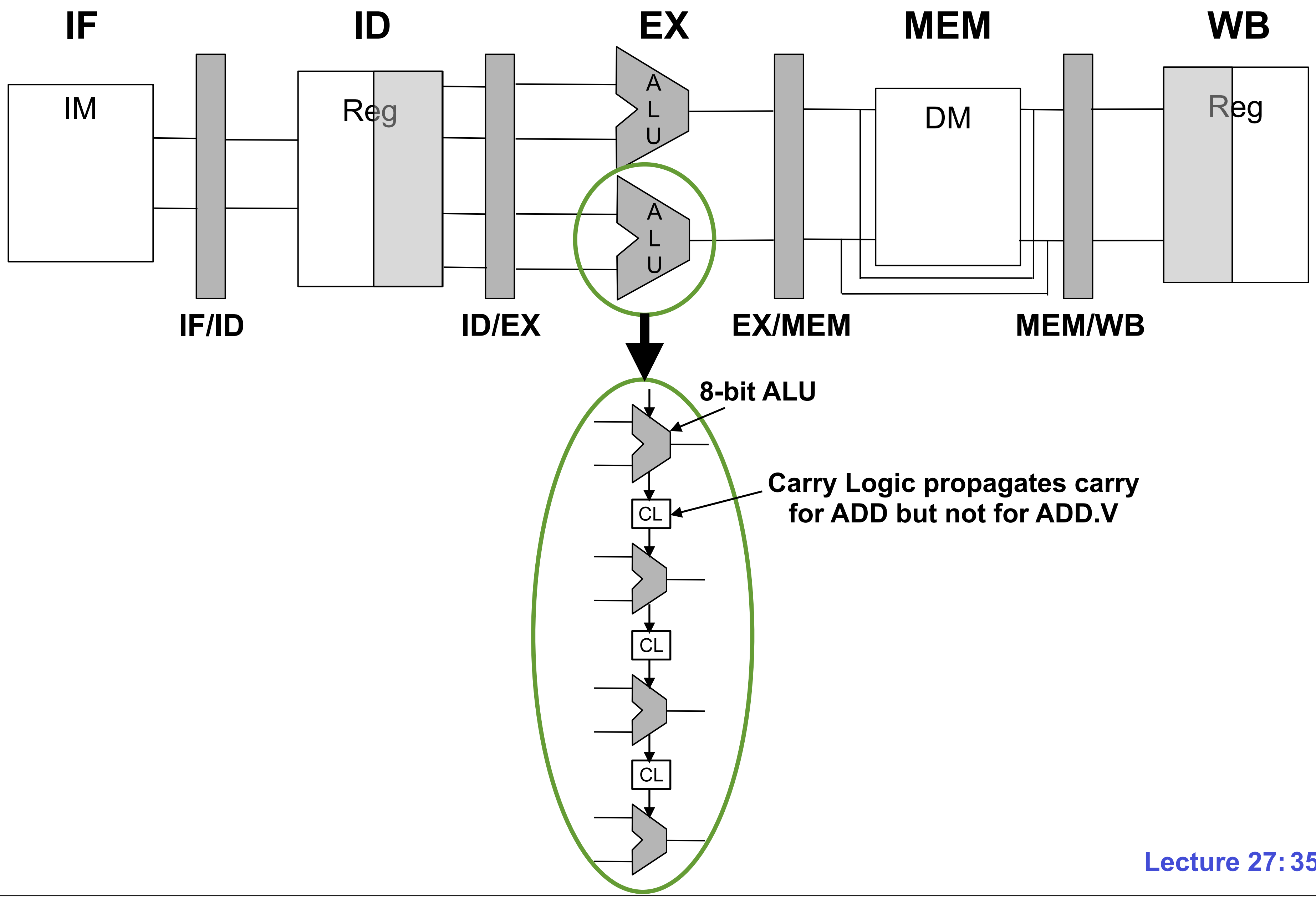
```
char a[4], b[4], c[4];

for (i = 0; i < 4; i++)
  a[i] = b[i] + c[i];
```

↓

```
; load b and c from memory
LW      R0, 0(R4)    ; R4 points to b
LW      R1, 0(R5)    ; R5 points to c
; vector add
ADD.V   R2, R0, R1   ; 1 inst does 4 adds!
; store result
SW      R2, 0(R3)    ; R3 points to a
```

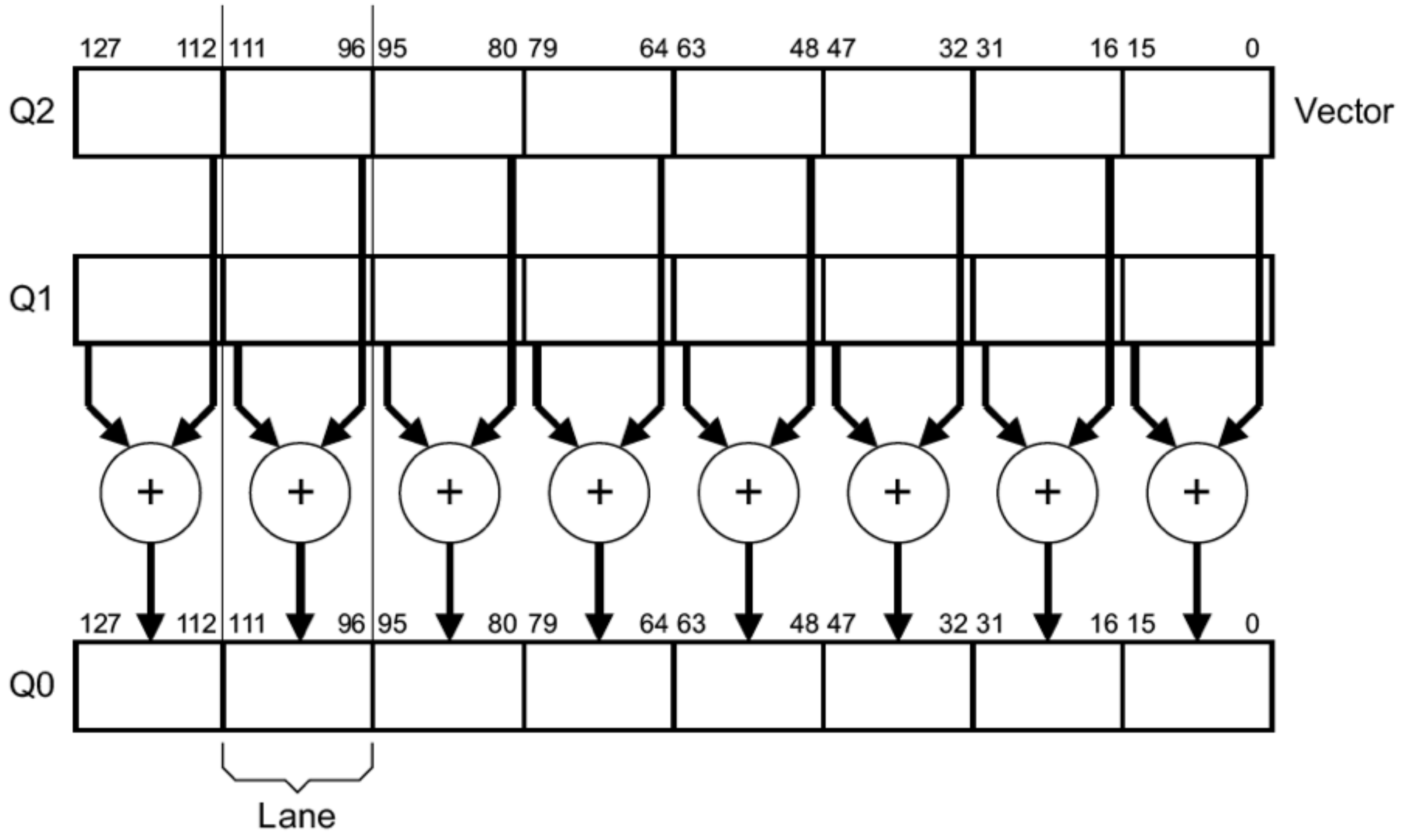
Assumes 32-bit rather than 16-bit registers,
and that a word is 32 bits

Adding Support for ADD.V to the ALU



Example ARM SIMD Instruction

```
VADD.I16 Q0, Q1, Q2
```



ILP, TLP, and DLP

- Many processors exploit all three
  - Best performance/watt achieved with each in moderation rather than one/two to the extreme
- ILP
  - Typically 2 to 6-way superscalar pipeline
  - Performance improvement tapers off with wider pipelines while power may increase significantly
- TLP
  - Support for 2 threads may require small amount of additional hardware over single threaded SS pipeline
  - May improve HW *efficiency* compared to SS alone
- DLP (via SIMD)
  - Many applications (graphics, video and audio processing, etc) make this worthwhile

Next Time

More Advanced Topics

Case Study