Samsung Convergence Software Academy

Lecture 3. ARM CPU Internal II

Prof. Hoh Peter In & Prof. Taeweon Suh

Korea University

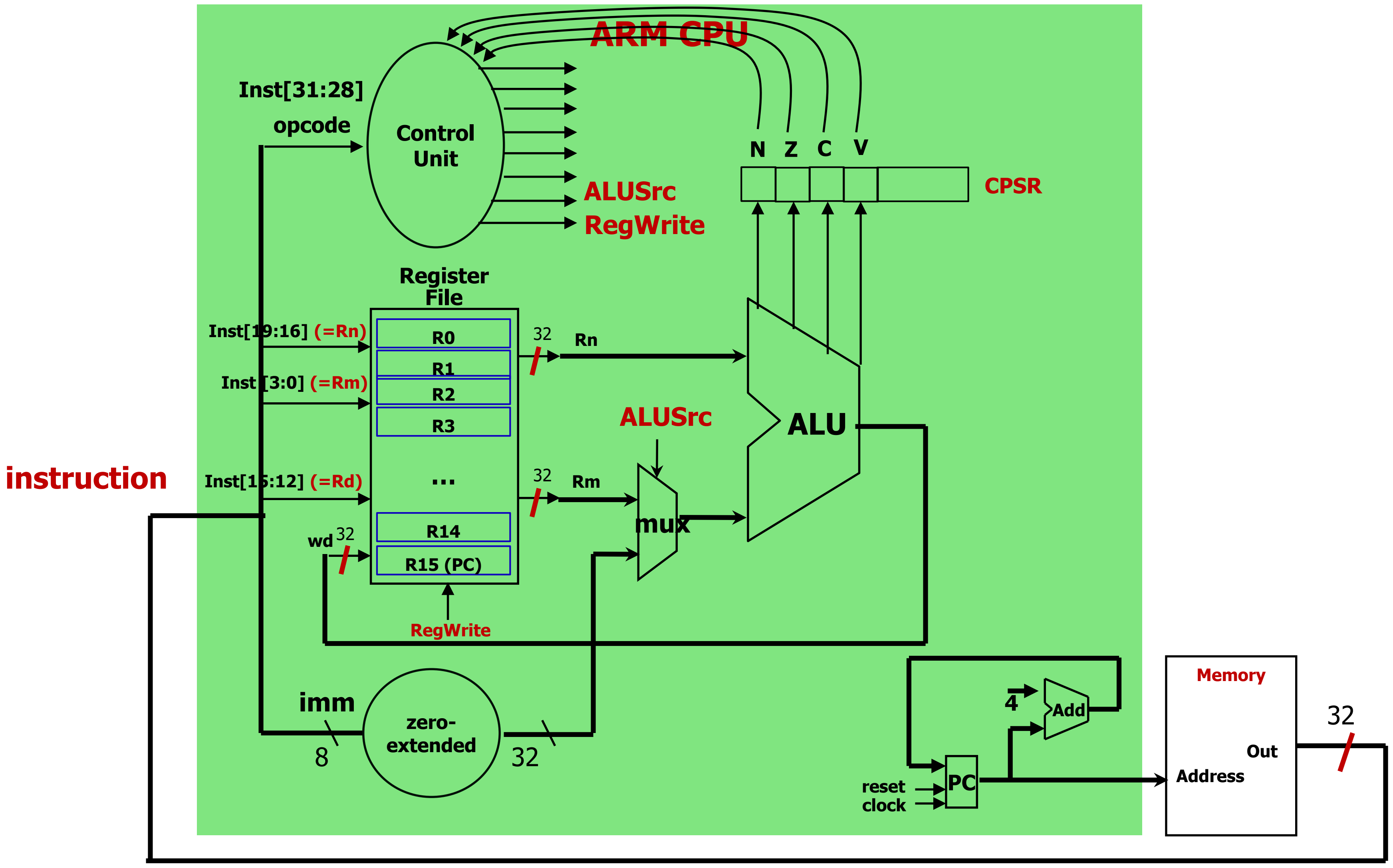
Conditional Execution

- **Most ARM instructions** can be **conditionally executed**
  - It means that they have their normal effect only if the N (Negative), Z (Zero), C (Carry) and V (Overflow) flags in the CPSR satisfy a condition specified in the instruction
    - If the flags do not satisfy this condition, the instruction acts as a NOP (No Operation)
    - In other words, the instruction has no effect and advances to the next instruction

Example Code

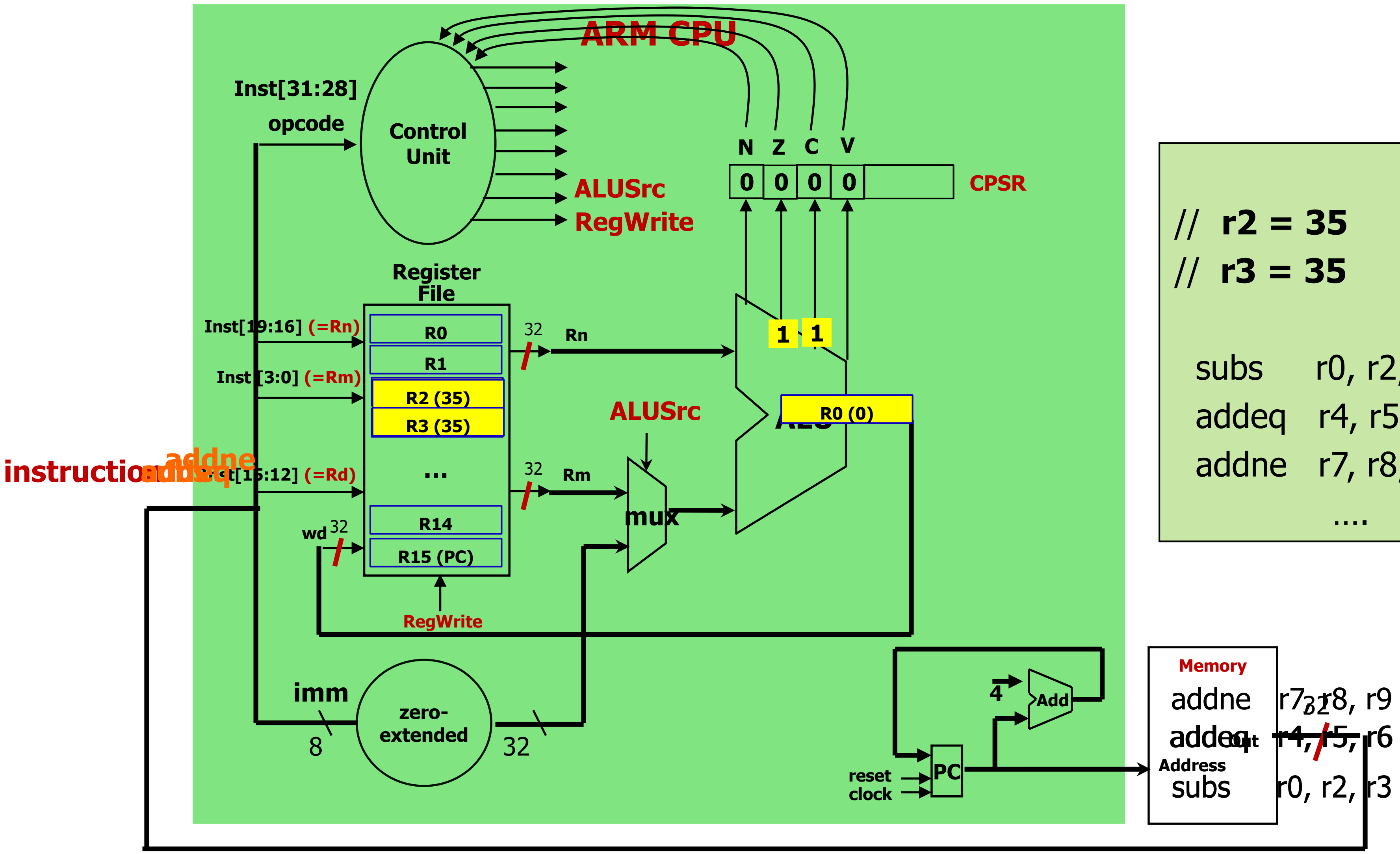
```
// Assume that r2 = 35 and r3 = 35

subs     r1, r2, r3    //  r1 =  r2 – r3 and set N, Z, C, V in CPSR
addeq    r4, r5, r6    //  r4 = r5 + r6 if condition is met
addne    r7, r8, r9     //  r7 = r8 | r9
         ….
```

Conditional Execution Logic



Korea Univ

Conditional Execution Example



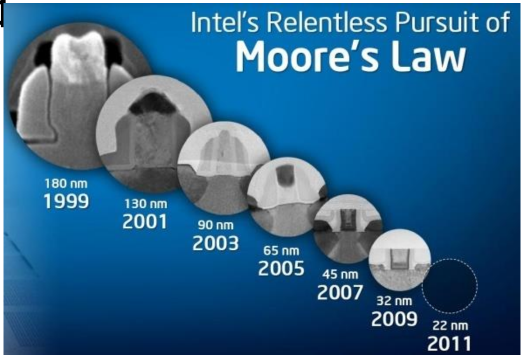
```
// r2 = 35
// r3 = 35

subs    r0, r2, r3
addeq   r4, r5, r6
addne   r7, r8, r9
....
```

Korea Univ

Processor Performance

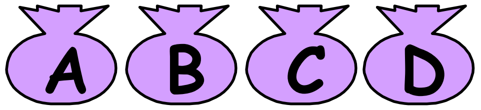
- Performance of single-cycle processor is limited by the **long** critical path delay
  - The critical path limits the operating clock frequency

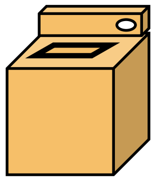
- Can we do better?
  - New semiconductor technology will reduce the critical path delay by manufacturing with small-sized transistors
    - Core 2 Duo: 65nm technology
    - 1st Gen. Core i7 (Nehalem): 45nm technology
    - 2nd Gen. Core i7 (Sandy Bridge): 32nm technology
    - 3rd Gen. Core i7 (Ivy Bridge): 22nm technology
  - Can we increase the processor performance with a different microarchitecture?
    - **Yes! Pipelining**

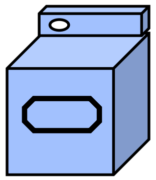


Revisiting Performance

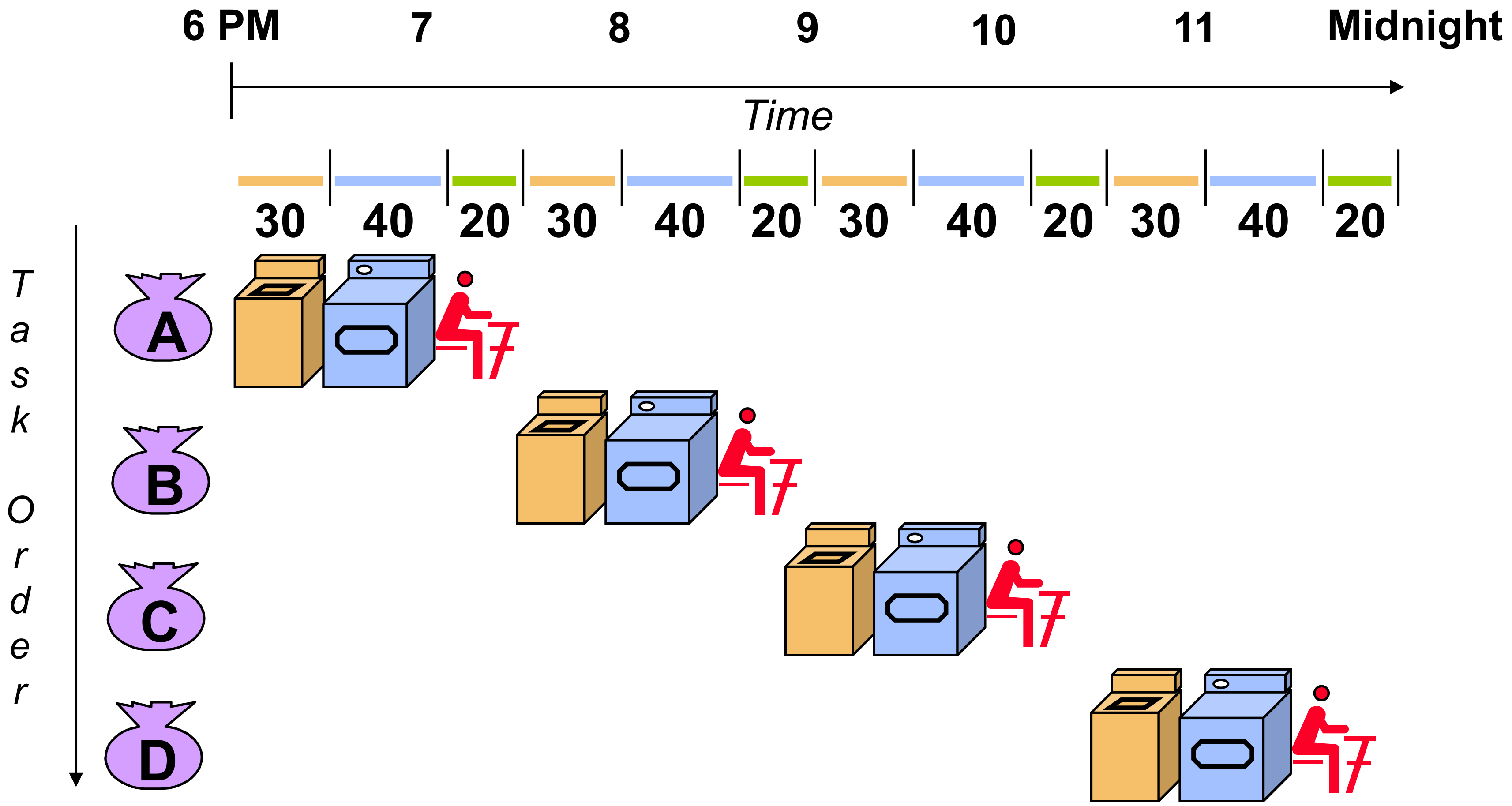
- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - **Washer** takes 30 minutes
  - **Dryer** takes 40 minutes
  - **Folder** takes 20 minutes





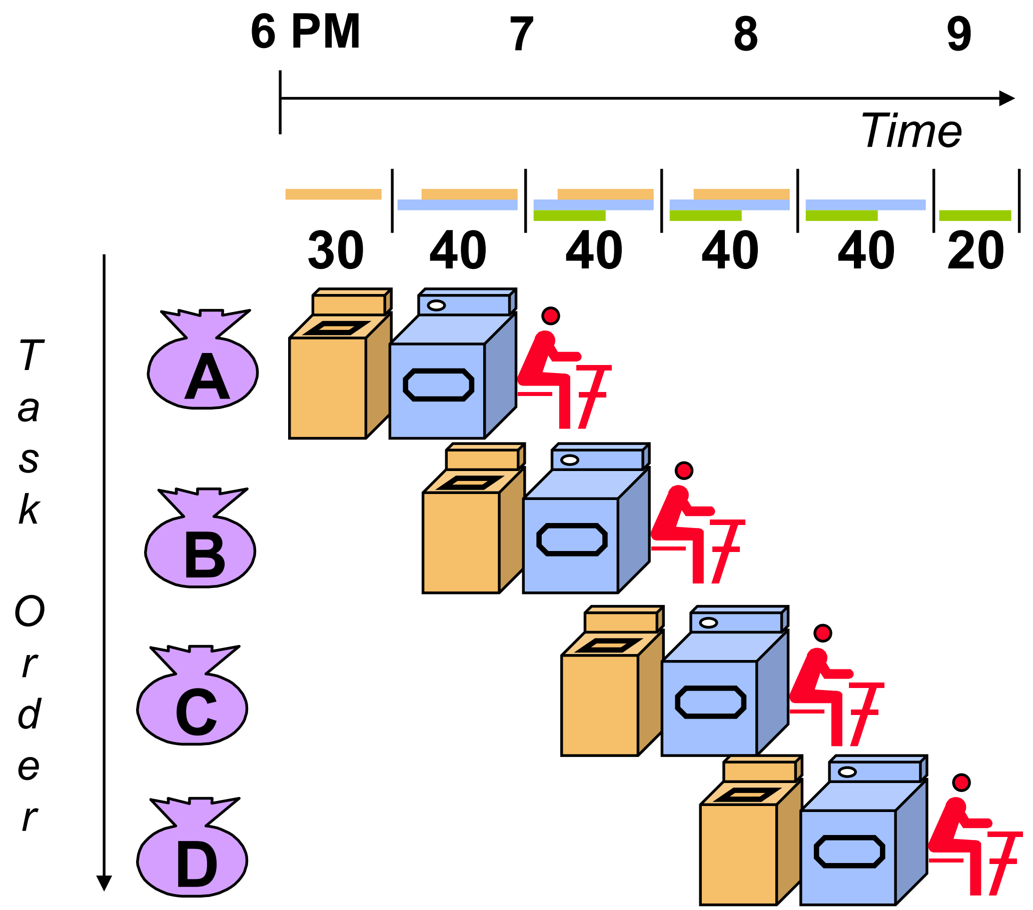


Sequential Laundry



- Response time: 90 mins
- Throughput: 0.67 tasks / hr (= 90mins/task, 6 hours for 4 loads)

Pipelining Lessons



- Pipelining doesn't help latency (response time) of a single task
- Pipelining helps throughput of entire workload
- Multiple tasks operating simultaneously
- Unbalanced lengths of pipeline stages reduce speedup
- Potential speedup = # of pipeline stages

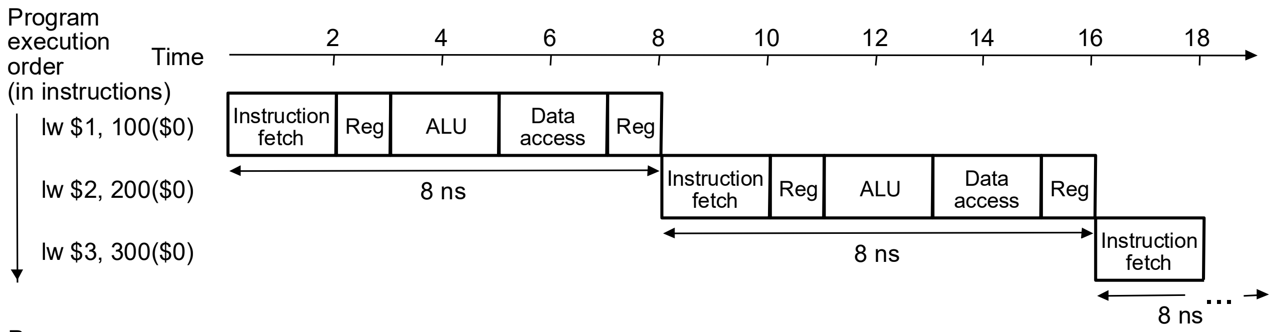
- Response time: 90 mins
- Throughput: 1.14 tasks / hr (= 52.5 mins/task, 3.5 hours for 4 loads)

Pipelining

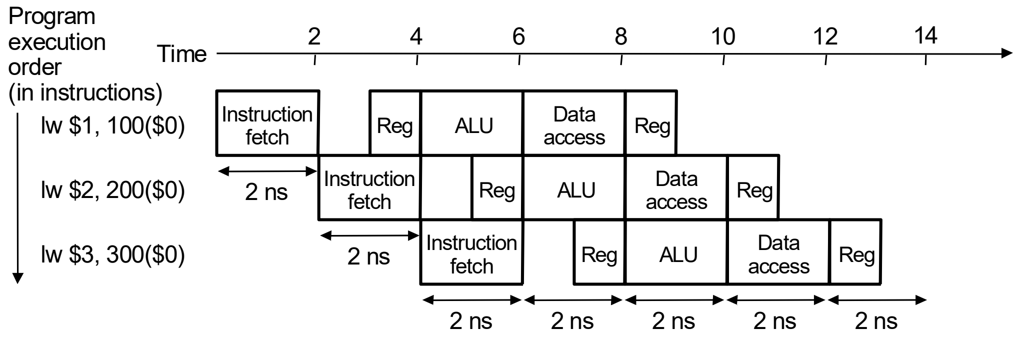
- Improve performance by increasing instruction **throughput**

Instruction Fetch	Register File Access (Read)	ALU Operation	Data Access	Register Access (Write)
2ns	1ns	2ns	2ns	1ns

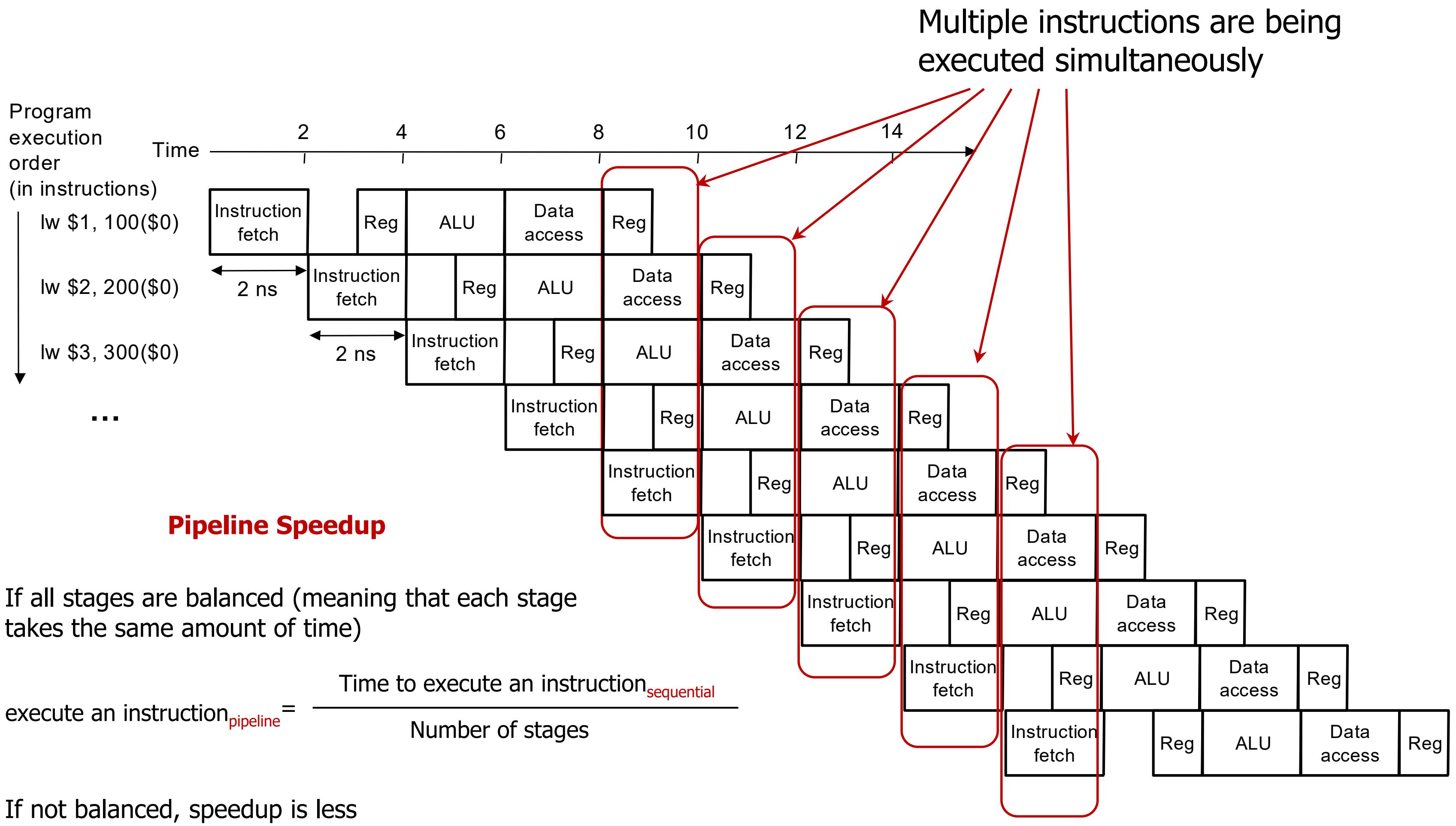
Sequential Execution



Pipelined Execution



Pipelining (Cont.)



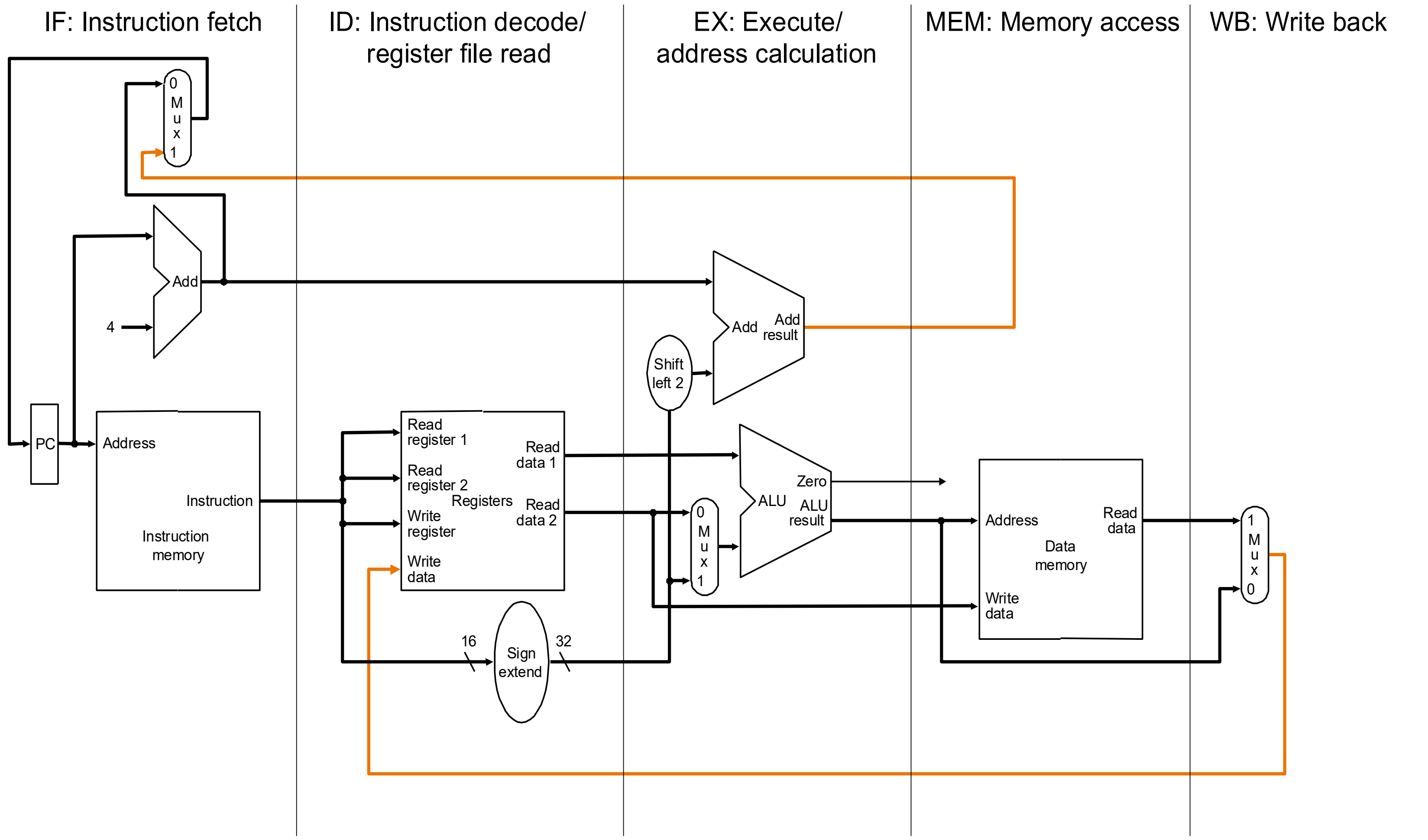
Pipeline Speedup

- If all stages are balanced (meaning that each stage takes the same amount of time)

$$\text{Time to execute an instruction}_{\text{pipeline}} = \frac{\text{Time to execute an instruction}_{\text{sequential}}}{\text{Number of stages}}$$

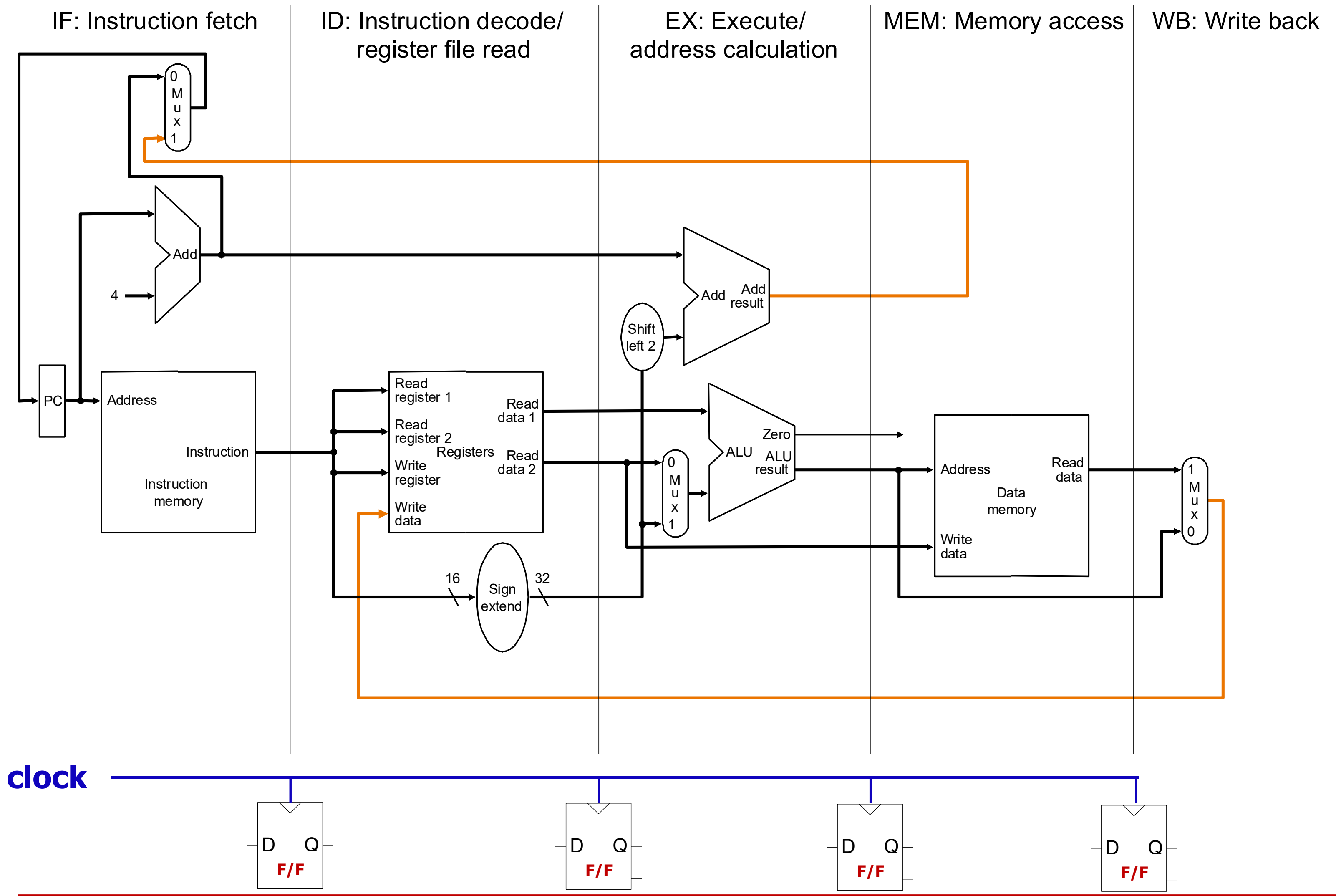
- If not balanced, speedup is less
- Speedup comes from increased **throughput** (the **latency** of instruction does not decrease)

Basic Idea

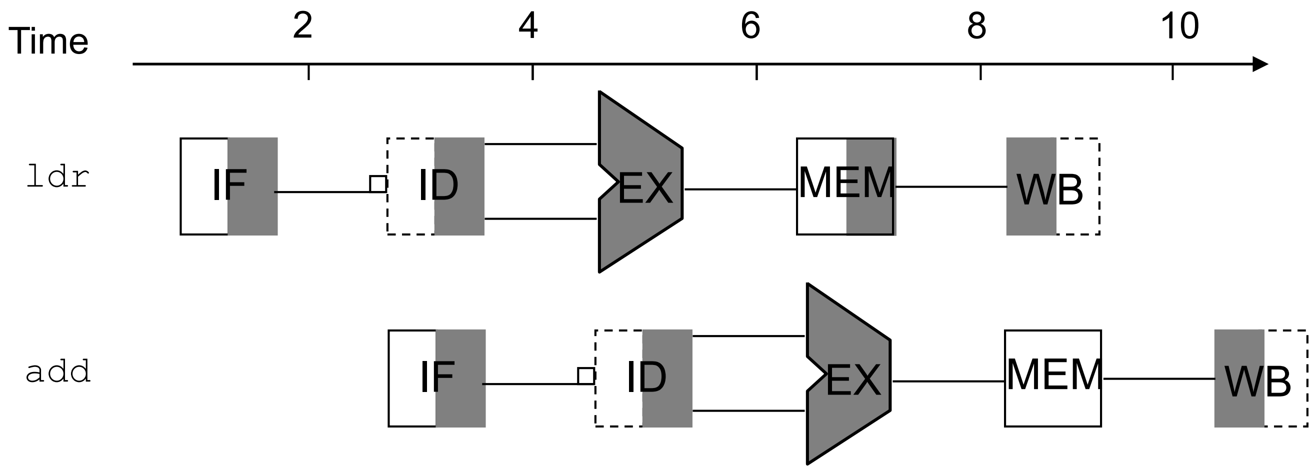


- What should be done to implement pipelining (split into stages)?

Basic Idea


IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
0 M u x 1
Add
4
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add
Add result
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
1 M u x 0
clock
D Q F/F
D Q F/F
D Q F/F
D Q F/F

Graphically Representing Pipelines



- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

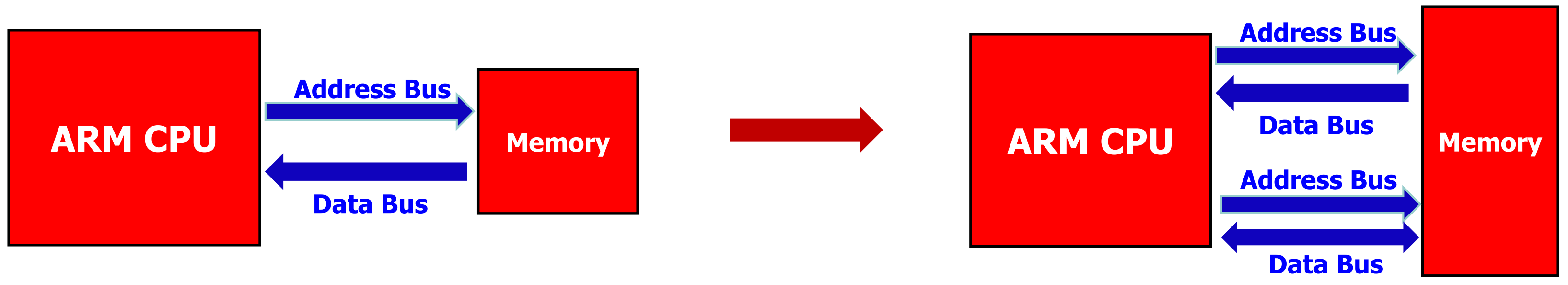
Hazards

- It would be happy if we split the datapath into stages and the CPU works just fine
  - But, things are not that simple as you may expect
  - There are **hazards**!

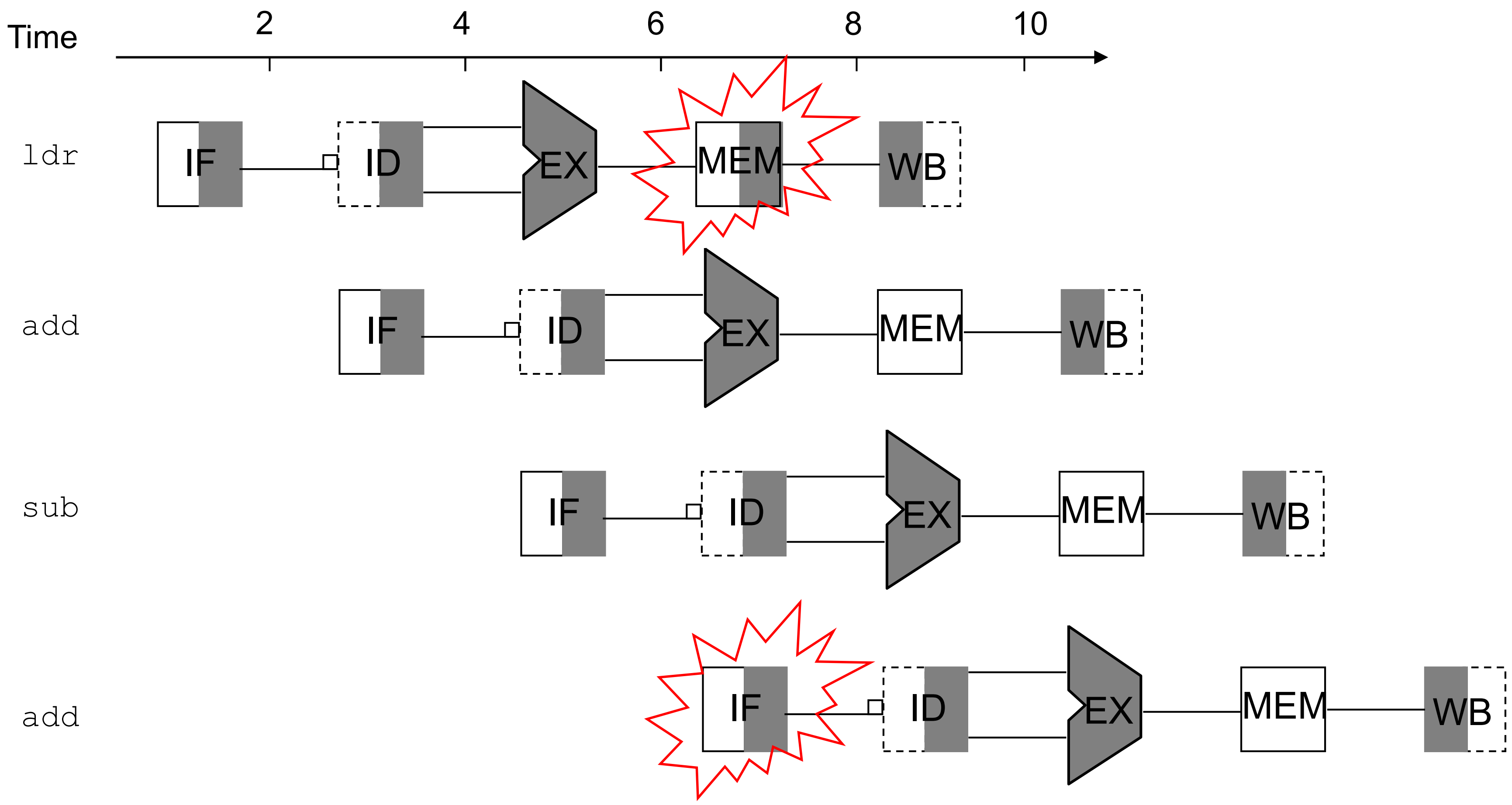
- Hazard is a situation that prevents starting the next instruction in the next cycle
  - **Structure hazards**
    - Conflict over the use of a resource at the same time
  - **Data hazard**
    - Data is not ready for the subsequent dependent instruction
  - **Control hazard**
    - Fetching the next instruction depends on the previous branch outcome

Structure Hazards

- Structural hazard is a conflict over the use of a resource at the same time
- Suppose the MIPS CPU with a single memory
  - Load/store requires data access in MEM stage
  - Instruction fetch requires instruction access from the same memory
    - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require either separate ports to memory or separate memories for instruction and data



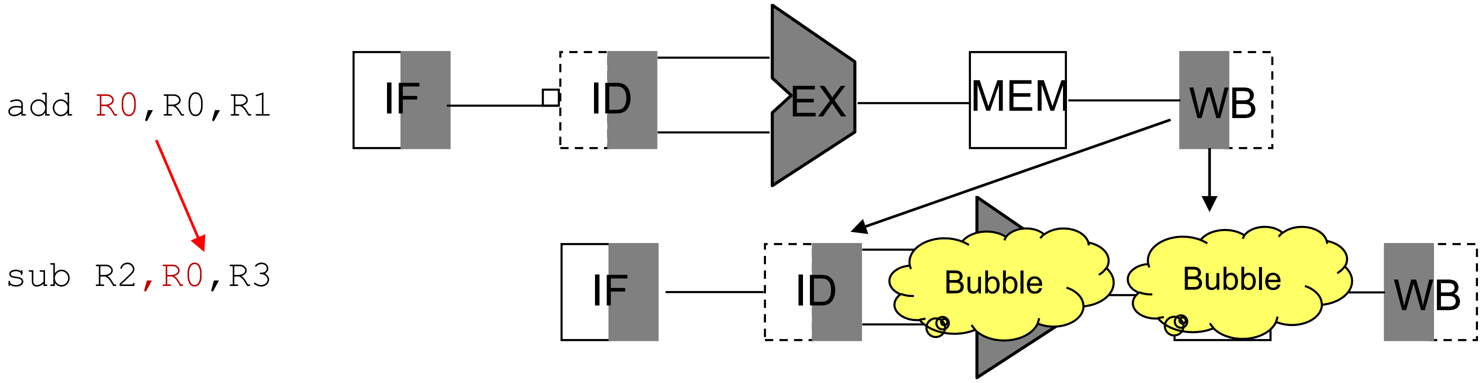
Structure Hazards (Cont.)



Either provide separate ports to access memory or provide instruction memory and data memory separately

Data Hazards

- Data is not ready for the subsequent dependent instruction

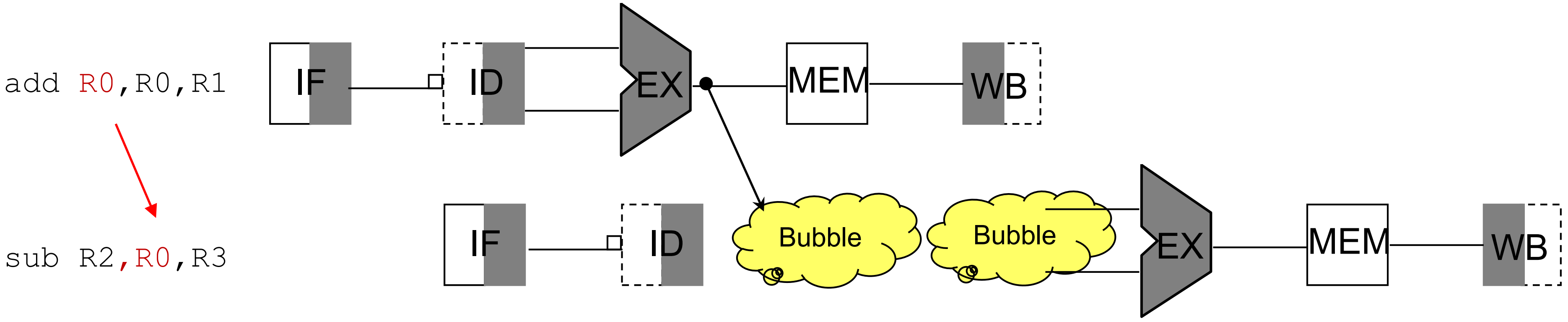


- To solve the data hazard problem, the pipeline needs to be stalled (typically referred to as "bubble")
  - Then, the performance is penalized
- A better solution?
  - **Forwarding** (or Bypassing)

Forwarding

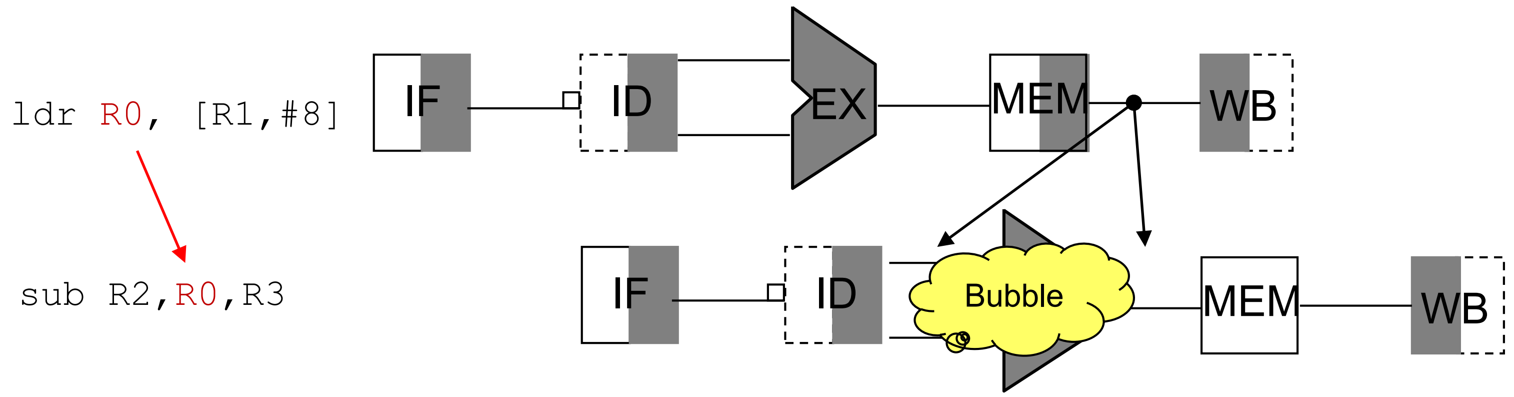
```
add R0,R0,R1
```

```
sub R2,R0,R3
```



Data Hazard - `Load-Use` Case

- Can't always avoid stalls by forwarding
  - Can't forward backward in time!
- Hardware interlock is needed for the pipeline stall

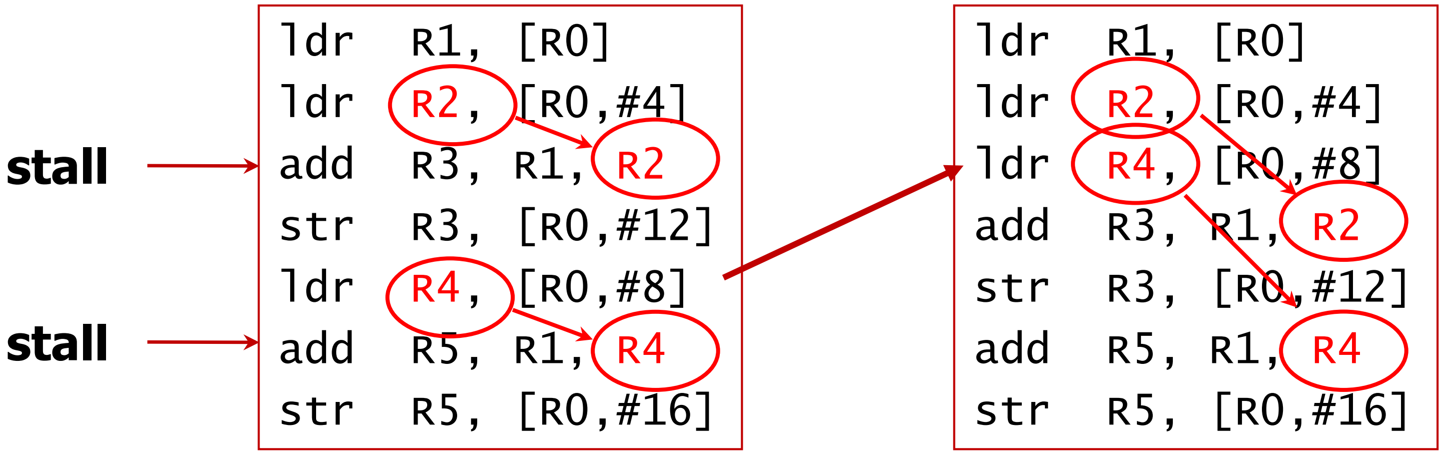


- This bubble can be hidden by proper instruction scheduling

Code Scheduling to Avoid Stalls

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

```
A = B + E; // B is loaded to R1, E is loaded to R2
C = B + F; // F is loaded to R4
```



```
        ldr   R1, [R0]
        ldr   R2, [R0,#4]
stall → add   R3, R1, R2
        str   R3, [R0,#12]
        ldr   R4, [R0,#8]
stall → add   R5, R1, R4
        str   R5, [R0,#16]
```

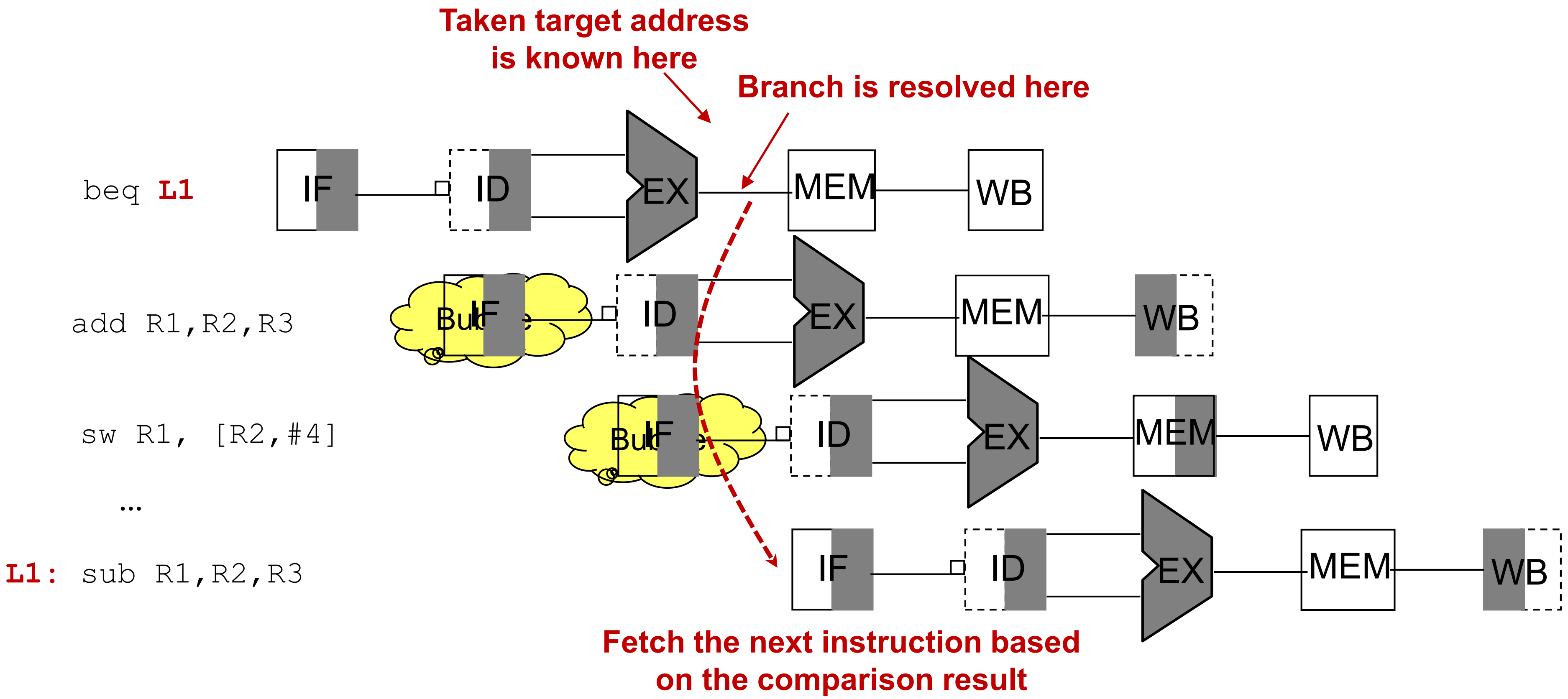
13 cycles

```
ldr   R1, [R0]
ldr   R2, [R0,#4]
ldr   R4, [R0,#8]
add   R3, R1, R2
str   R3, [R0,#12]
add   R5, R1, R4
str   R5, [R0,#16]
```

11 cycles

Control Hazard

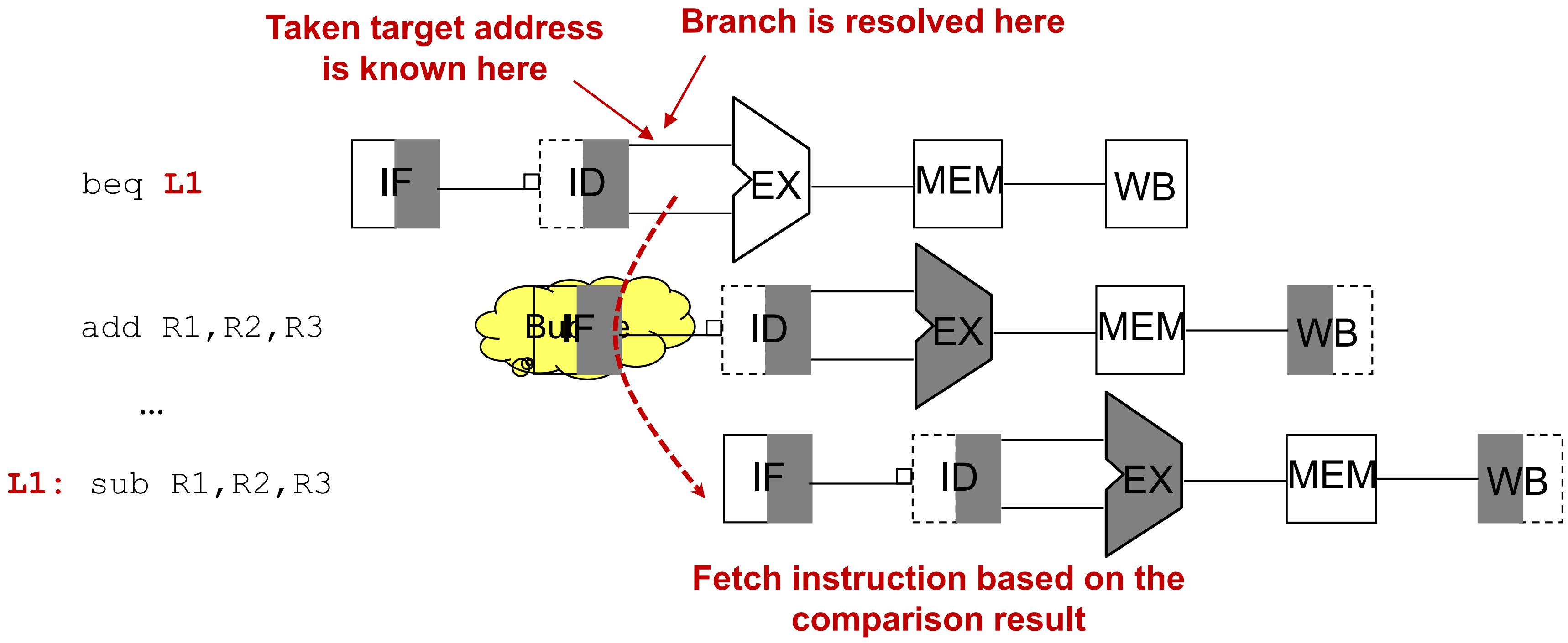
- Branch determines the flow of instructions
- Fetching the next instruction depends on the branch outcome
  - Pipeline can't always fetch correct instruction
  - Branch instruction is still working on ID stage when fetching the next instruction


Taken target address is known here
Branch is resolved here
beq L1
IF
ID
EX
MEM
WB
add R1,R2,R3
Bubble
sw R1, [R2,#4]
…
L1: sub R1,R2,R3
Fetch the next instruction based on the comparison result

Korea Univ

Reducing Control Hazard

- To reduce 2 bubbles to 1 bubble, add hardware in ID stage to compare registers (and generate branch condition)
  - But, it requires additional forwarding and hazard detection logic – Why?



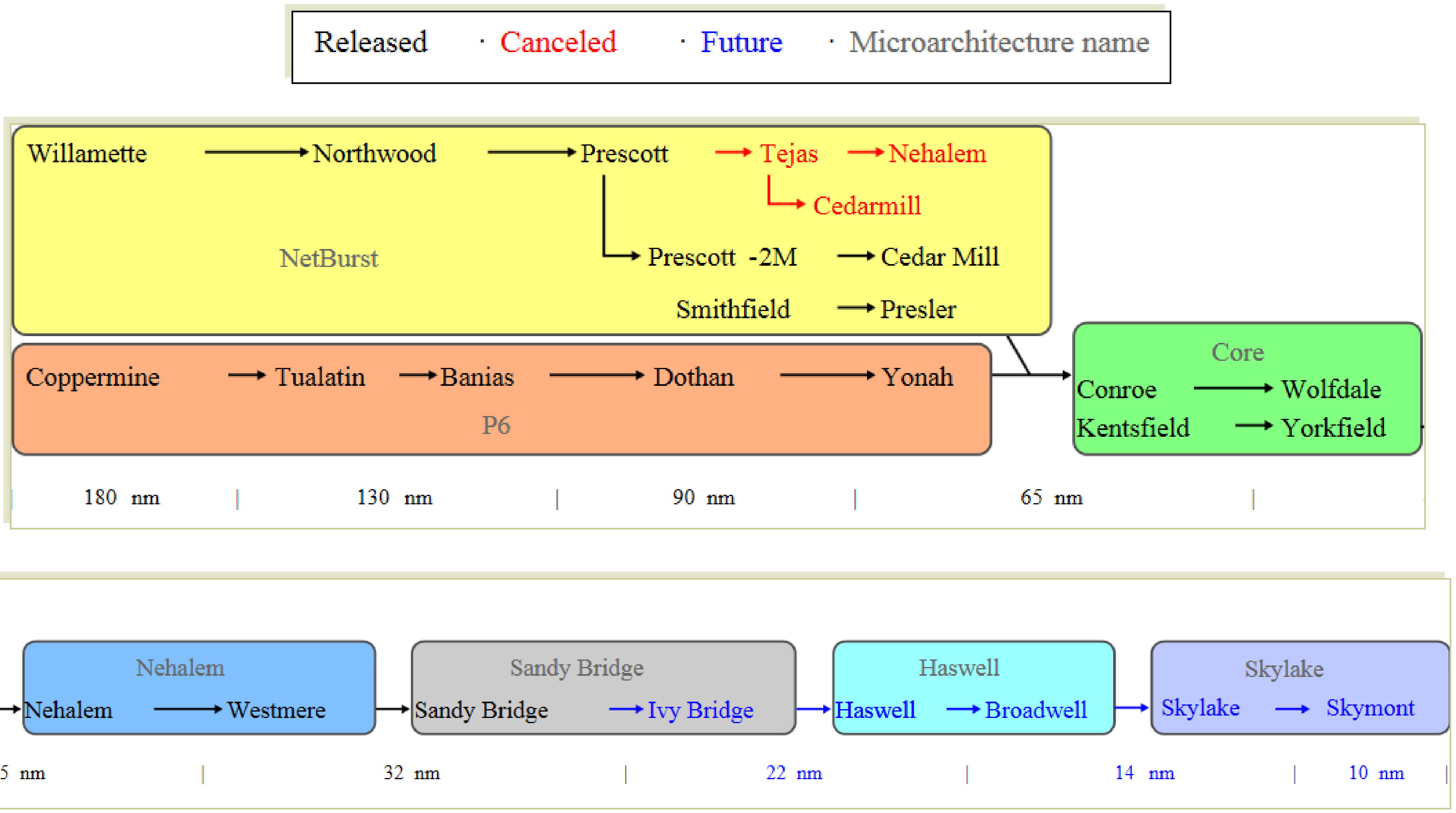
Taken target address is known here
Branch is resolved here
beq L1
IF
ID
EX
MEM
WB
add R1,R2,R3
Bubble
…
L1: sub R1,R2,R3
Fetch instruction based on the comparison result

Pipeline Summary

- Pipelining improves performance by increasing instruction **throughput**
  - Executes multiple instructions in parallel
- Pipelining is subject to hazards
  - Structure hazard
  - Data hazard
  - Control hazard
- ISA affects the complexity of the pipeline implementation

Backup Slides

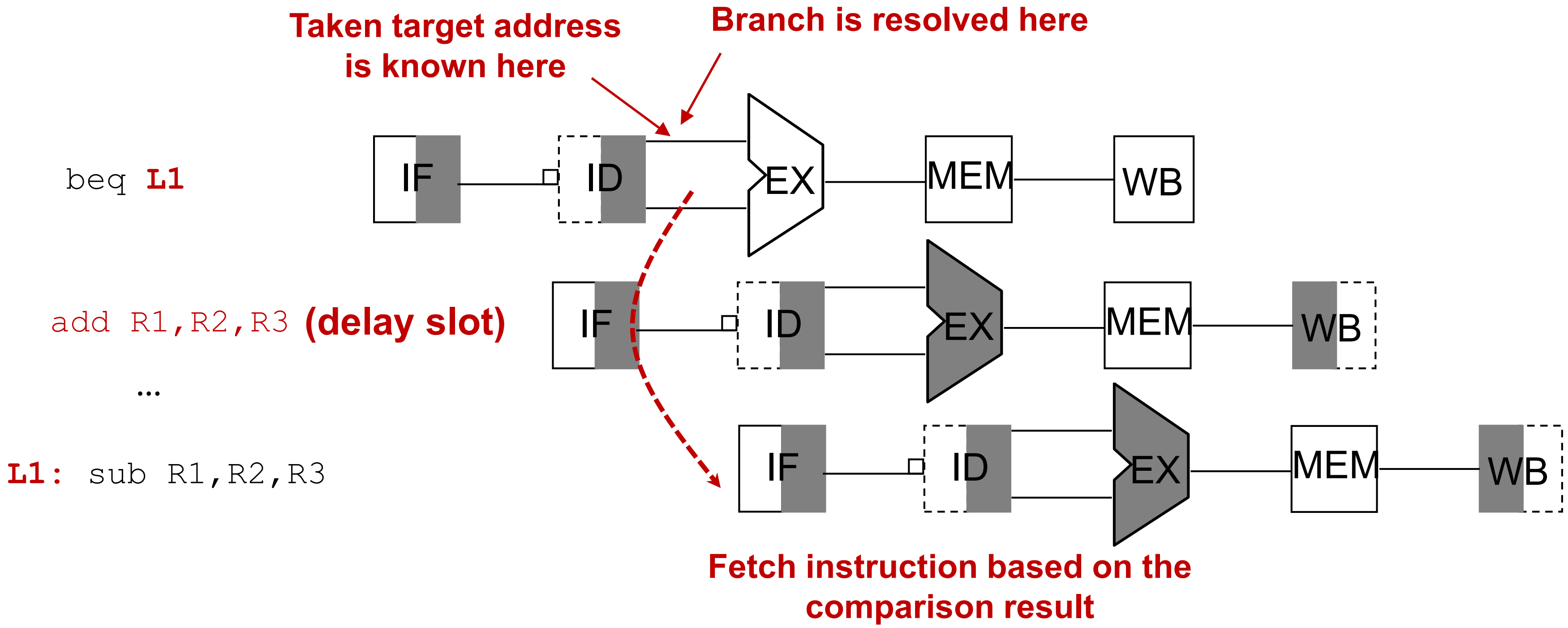
Past, Present, Future (Intel)



Source: http://en.wikipedia.org/wiki/Sandy_Bridge

Delayed Branch

- Many CPUs adopt a technique called the delayed branch to further reduce the stall
  - **Delayed branch** always executes the next sequential instruction
    - The branch takes place after that one instruction delay
  - **Delay slot** is the slot right after a delayed branch instruction



Taken target address is known here
Branch is resolved here
beq L1
add R1,R2,R3 (delay slot)
…
L1: sub R1,R2,R3
IF
ID
EX
MEM
WB
Fetch instruction based on the comparison result

Delay Slot (Cont.)

- Compiler needs to schedule a useful instruction in the delay slot, or fills it up with `nop` (no operation)

```
// R1 = a,  R2 = b,  R3 = c
// R4 = d,  R5 = f
a = b + c;
if (d == 0) {f = f + 1;}
f = f + 2;
```

$\longrightarrow$

```
    add R1,R2,R3
    cmp R4,#0
    bne L1
    nop  //delay slot
    add R5, R5, #1
L1: add R5, R5, #2
```

Can we do better?

```
    cmp R4, #0
    bne L1
    add R1,R2,R3 // delay slot
    addi R5, R5, #1
L1: addi R5, R5, #2
```

Fill the delay slot with a useful and valid instruction

Other suggestion using condition code in ARM?

Branch Prediction

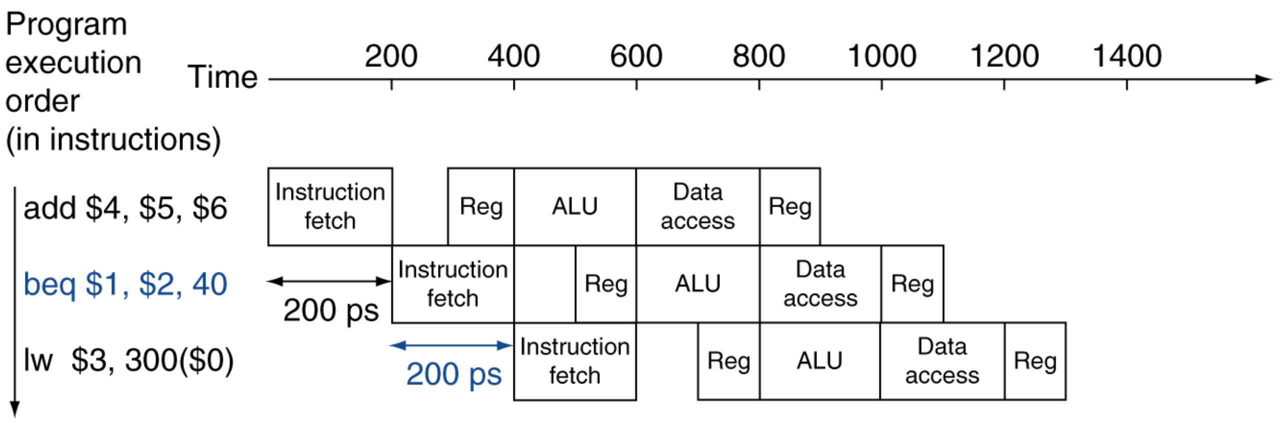
- Longer pipelines (for example, Core 2 Duo has 14 stages) can't readily determine branch outcome early
  - **Stall penalty** becomes unacceptable since branch instructions are used so frequently in the program

- **Solution: Branch Prediction**
  - Predict the branch outcome in hardware
  - Flush the instructions (that shouldn't have been executed) in the pipeline if the prediction turns out to be wrong
  - Modern processors use sophisticated branch predictors

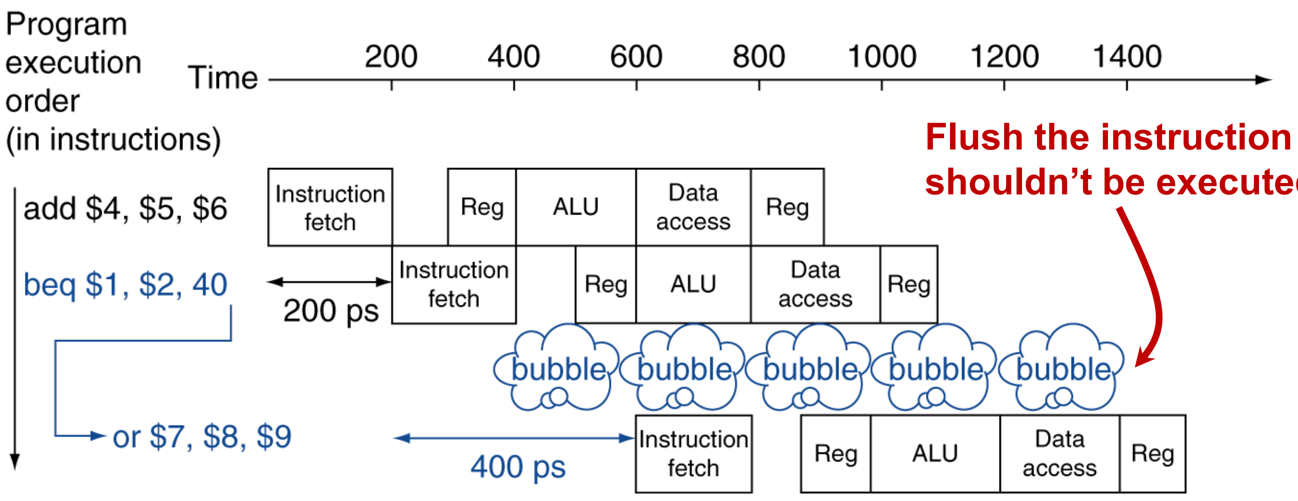
- Our MIPS implementation is like branches-not-taken prediction (with no delayed branch)
  - Fetch the next instruction after branch
  - If the prediction turns out to be wrong, flush out the instruction fetched

MIPS with Predict-Not-Taken

Prediction correct



Prediction incorrect



Flush the instruction that shouldn't be executed