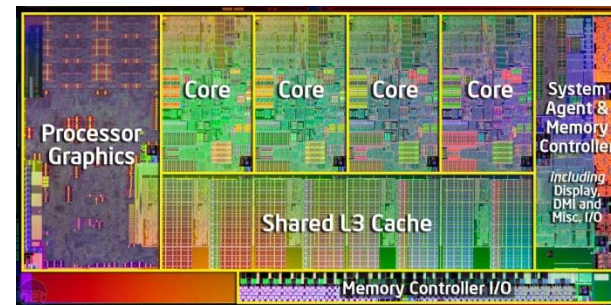




COSE222, COMP212 Computer Architecture

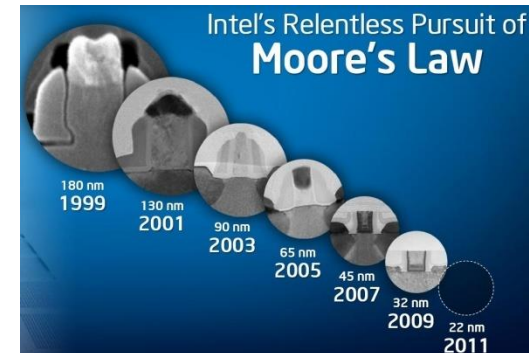
Lecture 5. MIPS Processor Design

Pipelined MIPS #1

Prof. Taeweon Suh
Computer Science & Engineering
Korea University

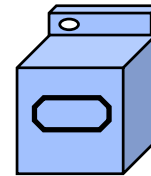
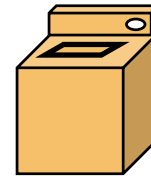
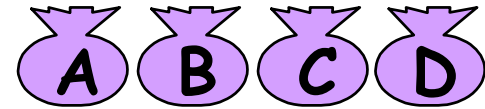
Processor Performance

- 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 , 4th Gen. Core i7 (Ivy Bridge, Haswell): 22nm technology
 - 5th Gen. Core i7 (Broadwell): 14nm technology
 - Can we increase the processor performance with a different microarchitecture?
 - **Yes! Pipelining**

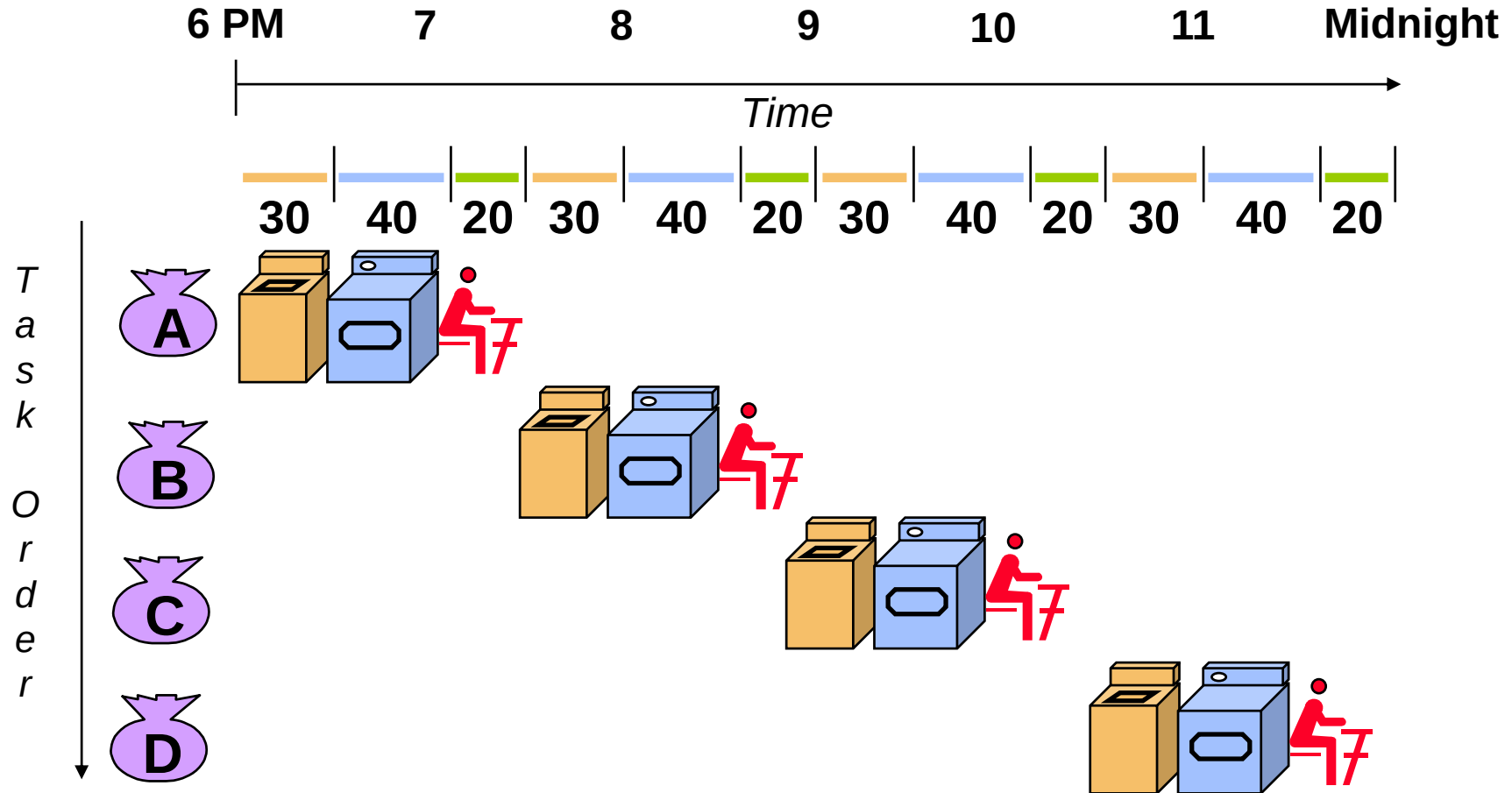


Revisiting Performance

- 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

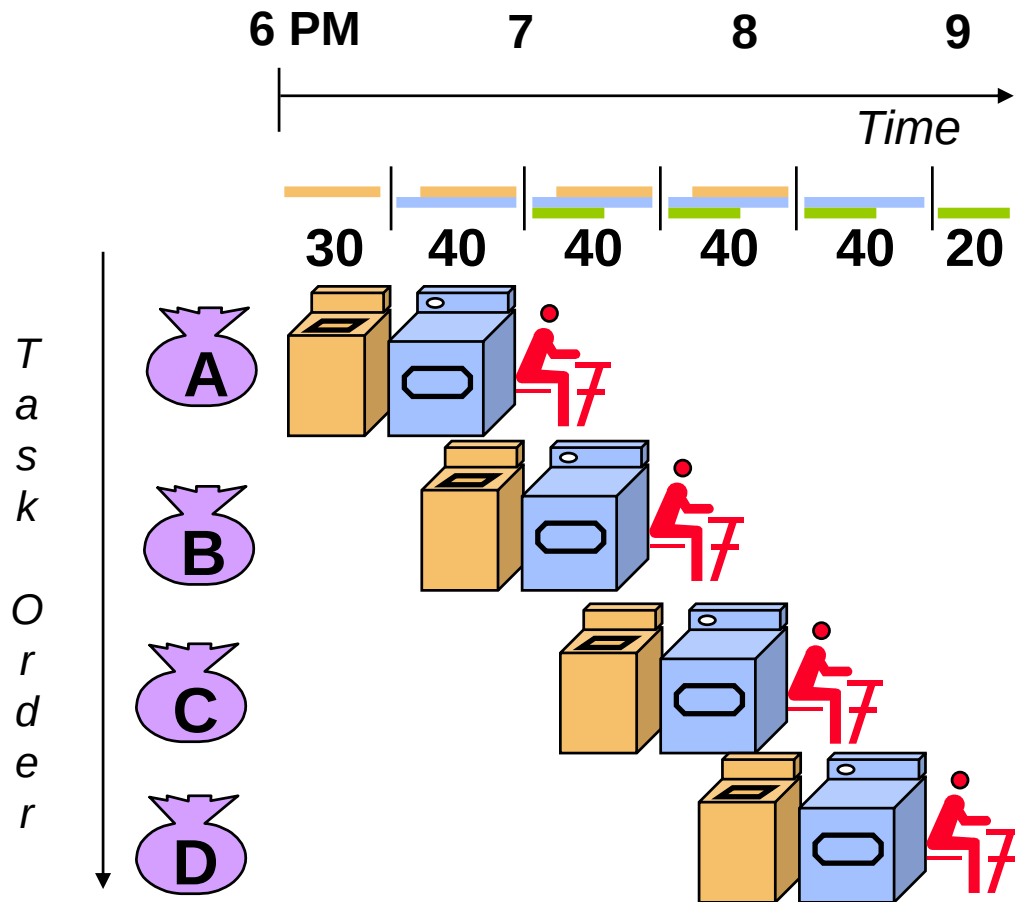


Sequential Laundry



- 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

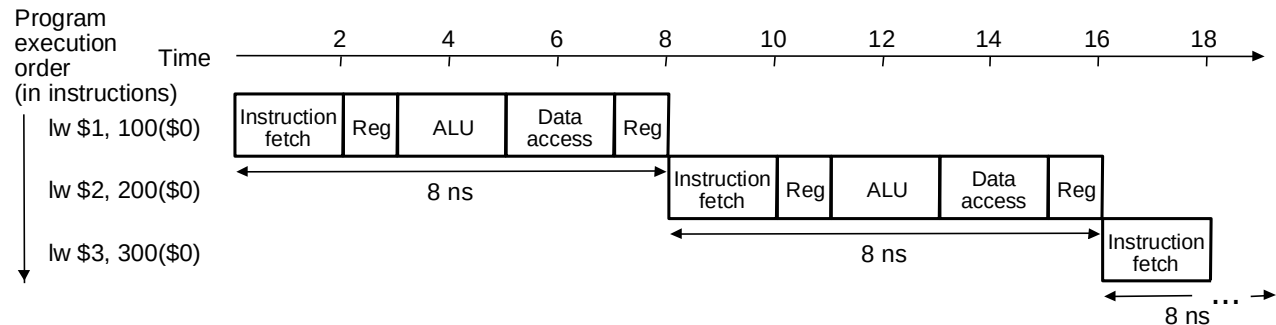
- Response time: **90 mins**
- Throughput: **1.14 tasks / hr (= 52.5 mins/task, 3.5 hours for 4 loads)**

Pipelining

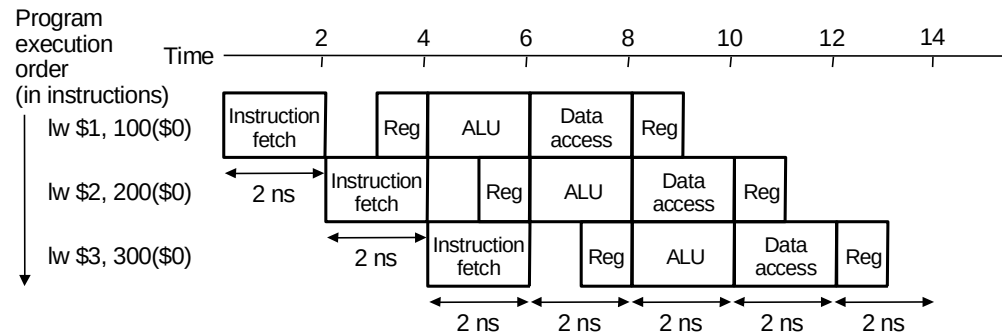
- Improve performance by increasing instruction **throughput**

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

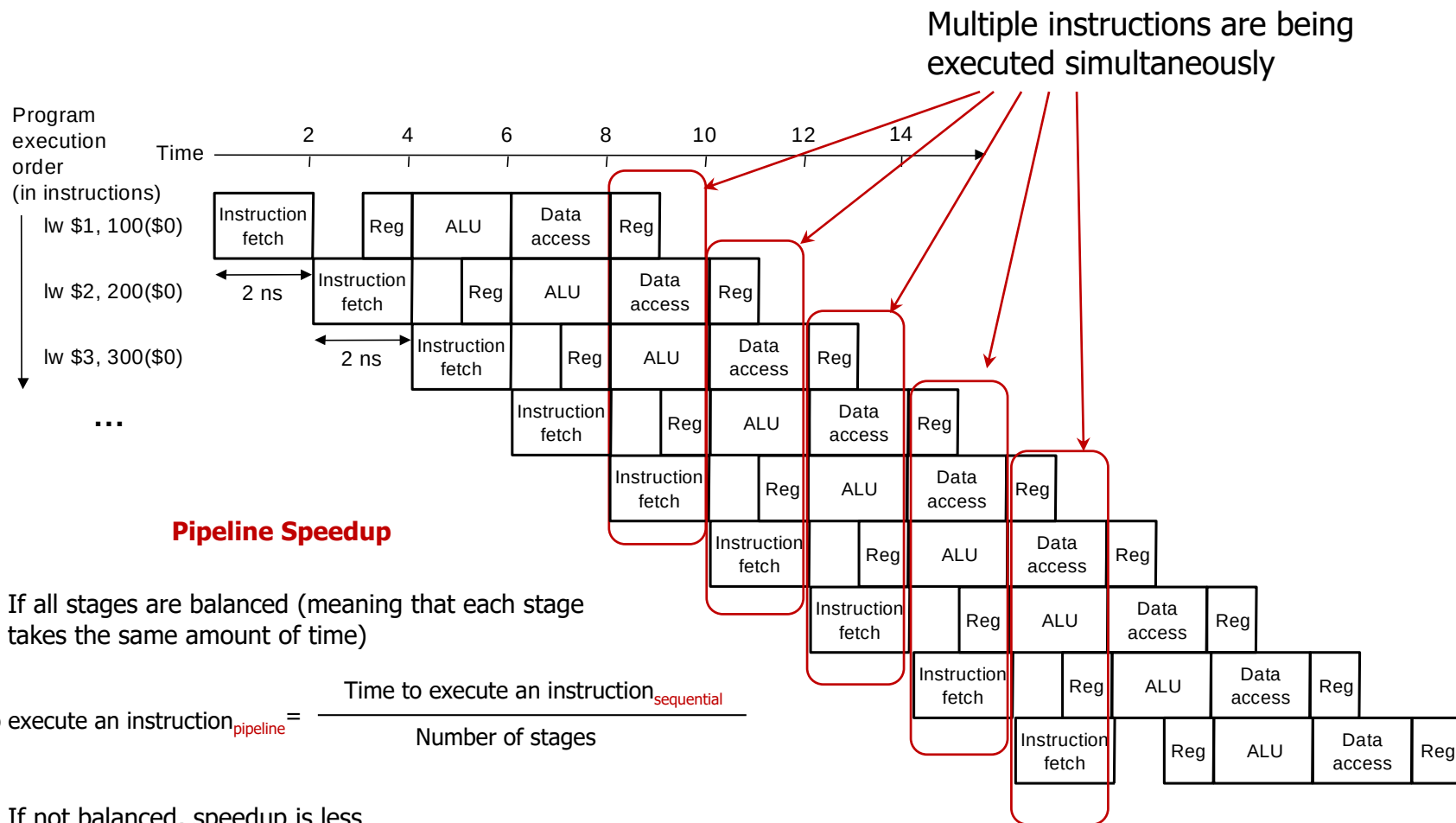
Sequential Execution



Pipelined Execution



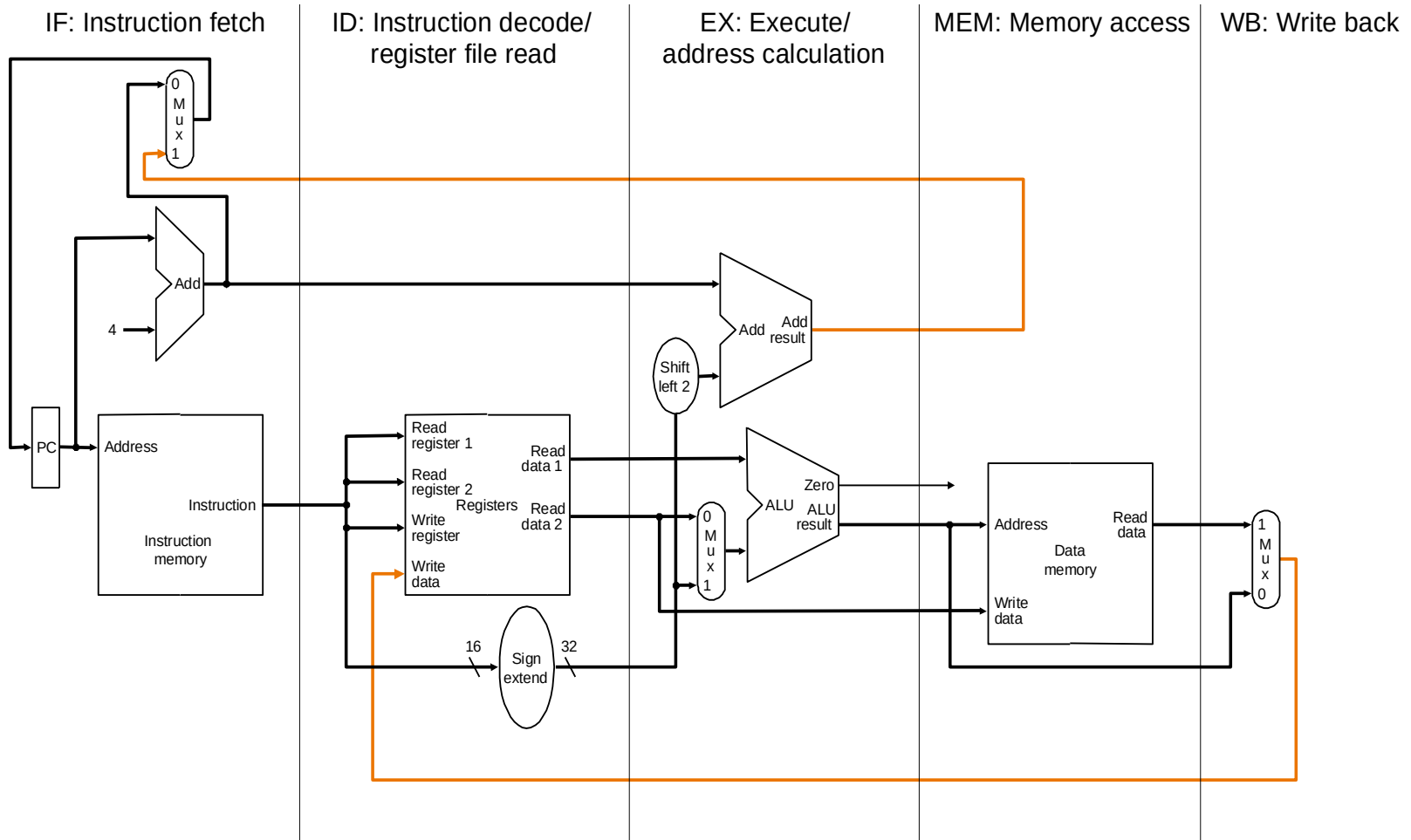
Pipelining (Cont.)



Pipelining and ISA Design

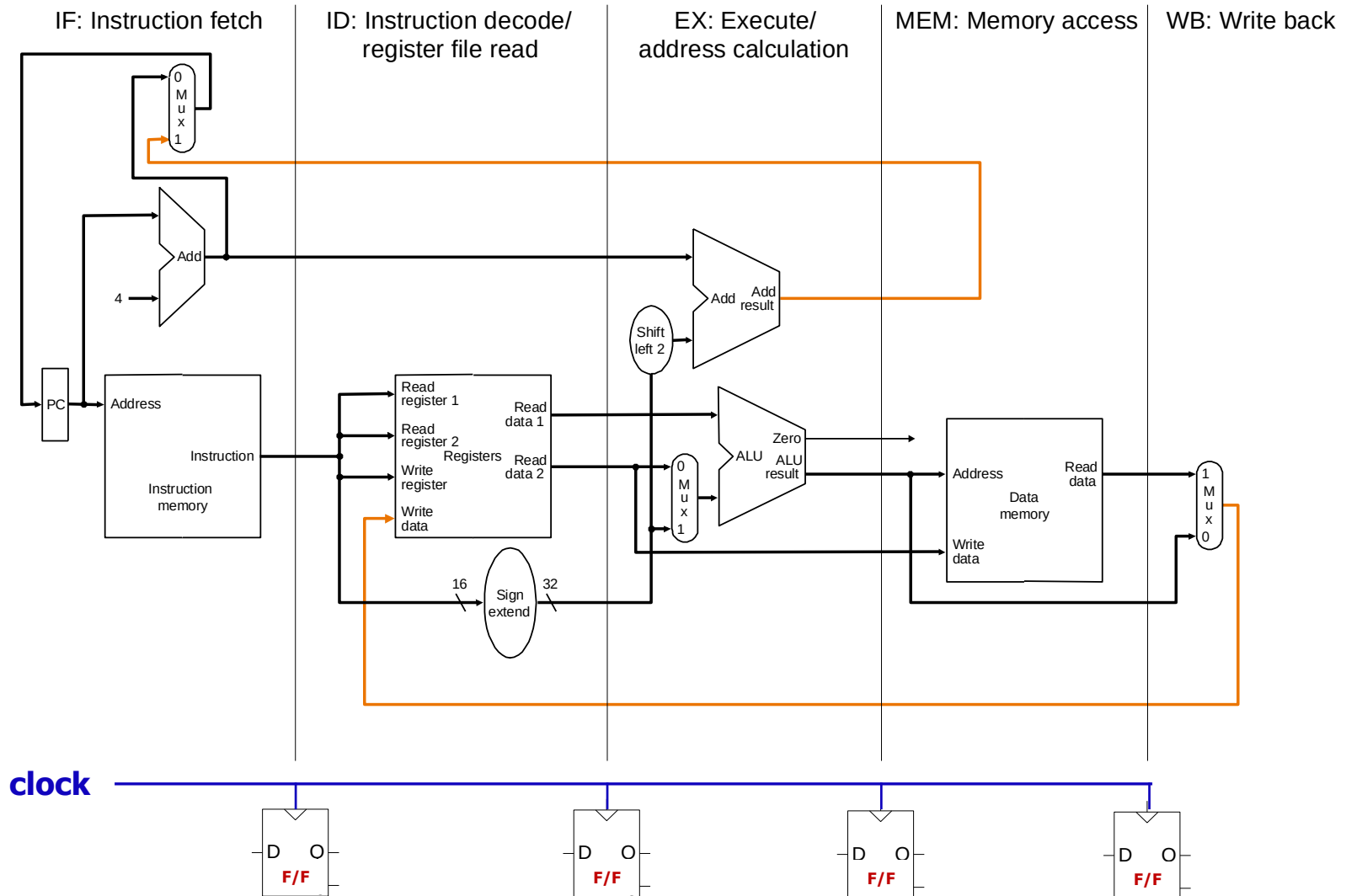
- MIPS ISA is designed with pipelining in mind
 - All instructions are 32-bits (4 bytes)
 - Compared with x86 (CISC): 1- to 17-byte instructions
 - Regular instruction formats
 - Can decode and read registers in one cycle
 - Load/store addressing
 - Calculate address in 3rd stage
 - Access memory in 4th stage
 - Alignment of memory operands in memory
 - Memory access takes only one cycle
 - For example, 32-bit data (word) is aligned at word address
 - 0x...0, 0x...4, 0x...8, 0x...C

Basic Idea

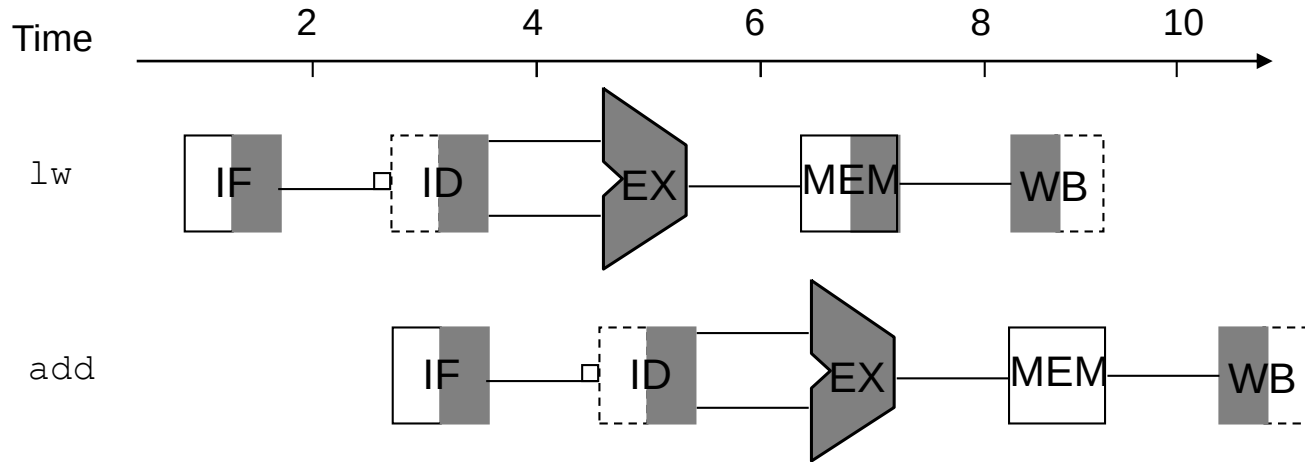


- What should be done to implement pipelining (split into stages)?

Basic Idea



Graphically Representing Pipelines



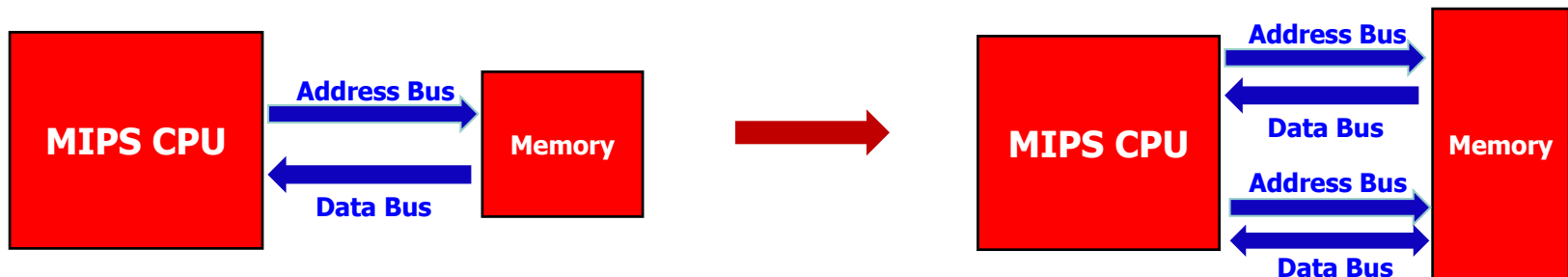
- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

Hazards

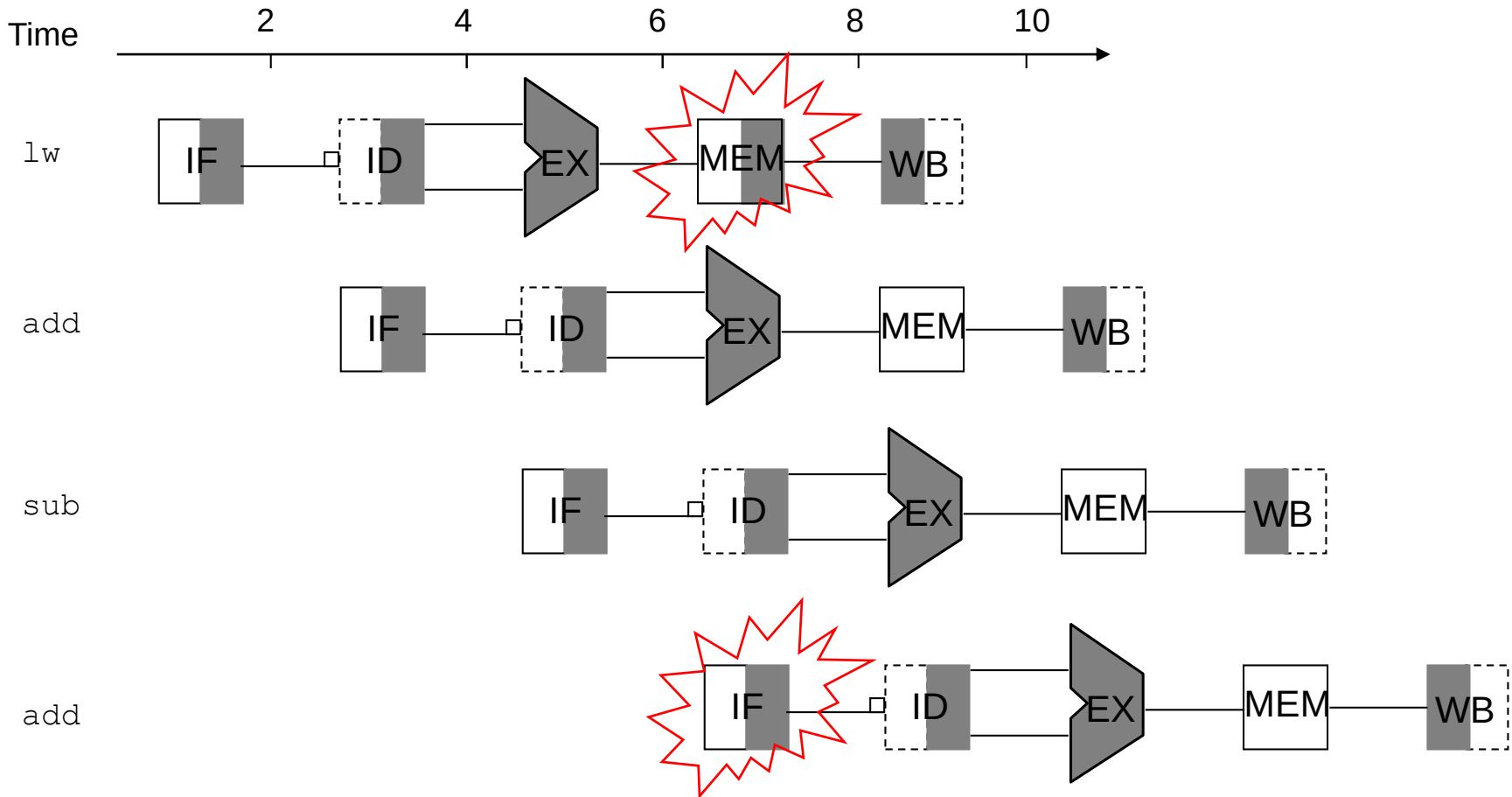
- It would be happy if we split the datapath into stages and the CPU works just fine
 - But, things are not that simple as you may expect
 - There are **hazards**!
- Hazard is a situation that prevents starting the next instruction in the next cycle
 - **Structure hazards**
 - Conflict over the use of 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 1 connection (1 port) to memory
 - Load/store requires data access in MEM stage
 - Instruction fetch requires instruction access from the same memory
 - Instruction fetch would have to be **stalled** for that cycle
 - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require either separate ports to memory or separate memories for instruction and data



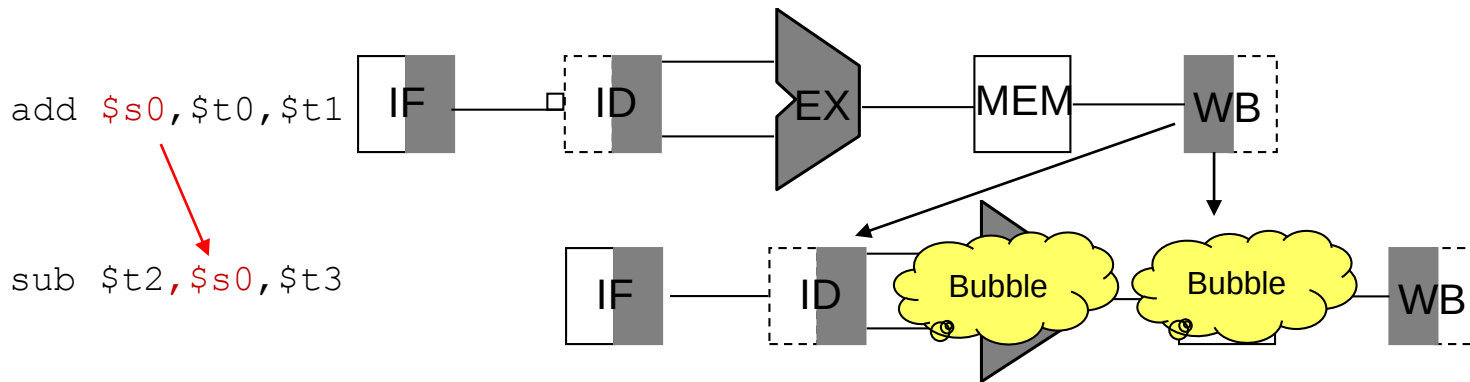
Structure Hazards (Cont.)



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

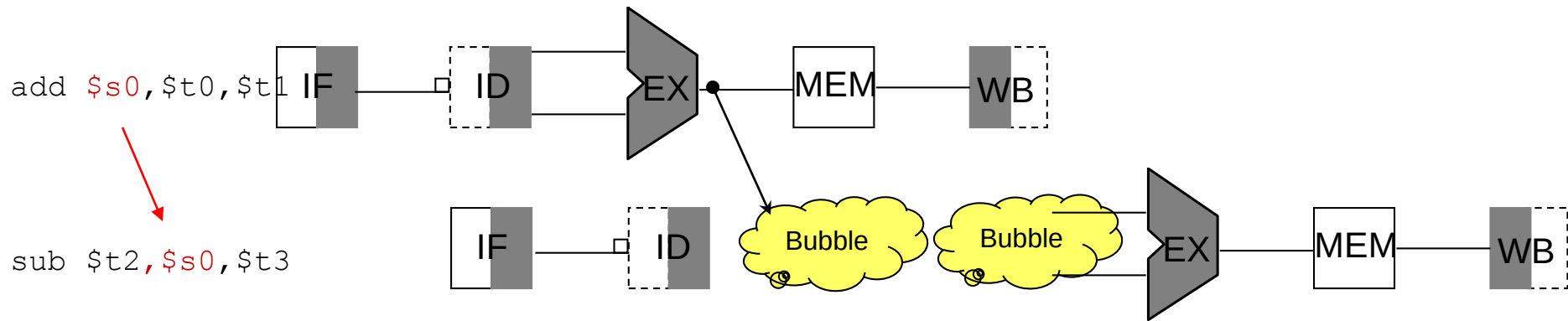
Data Hazards

- Data is not ready for the subsequent dependent instruction



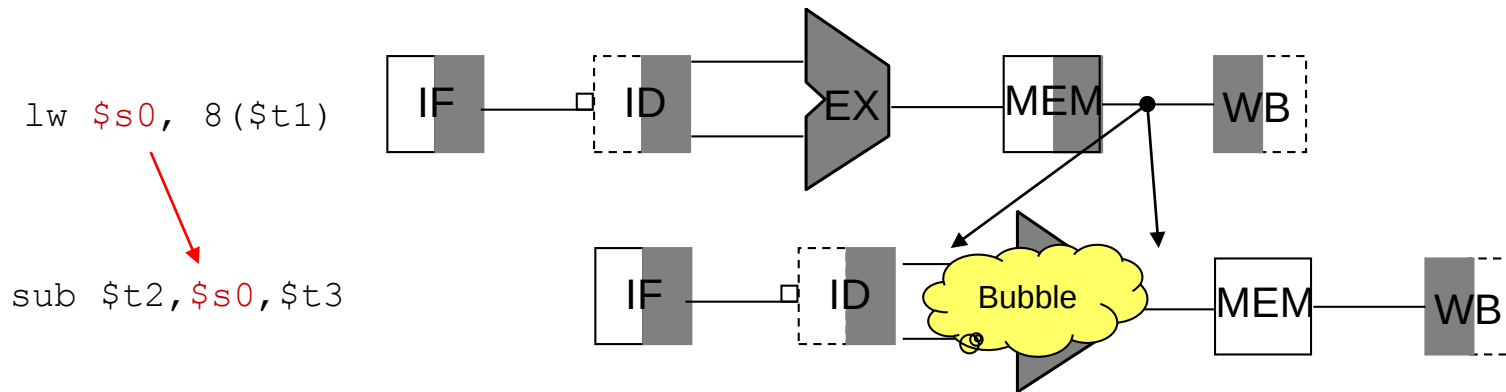
- To solve the data hazard problem, the pipeline needs to be stalled (typically referred to as "bubble")
 - Then, the performance is penalized
- A better solution?
 - **Forwarding** (or Bypassing)

Forwarding



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

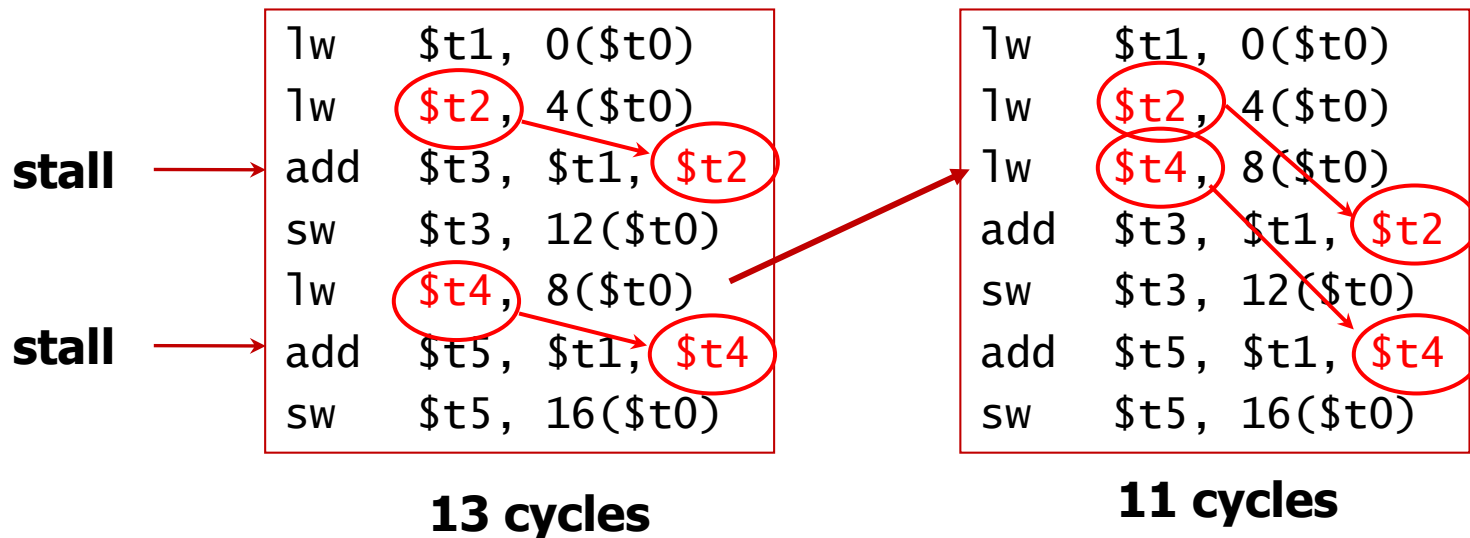


- This bubble can be hidden by proper instruction scheduling

Code Scheduling to Avoid Stalls

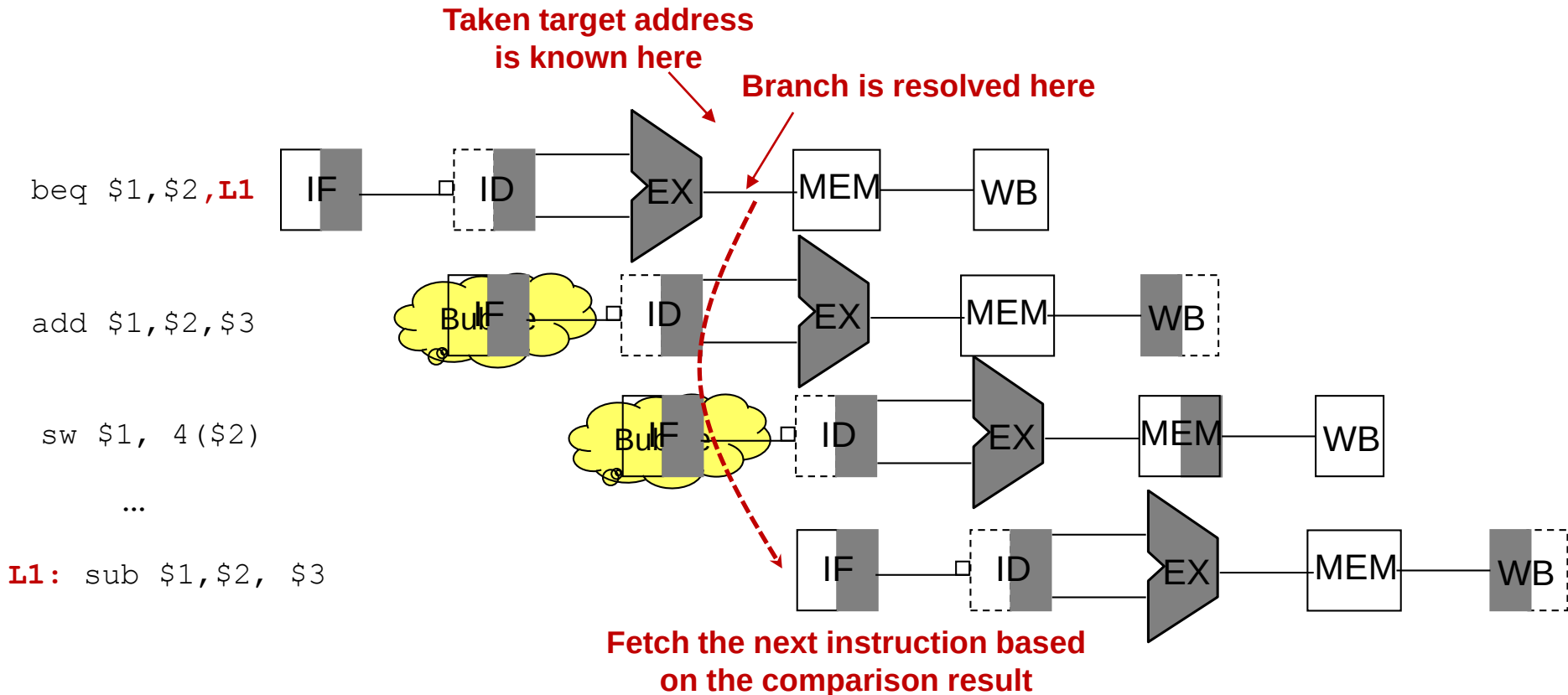
- Reorder code to avoid use of load result in the next instruction

`A = B + E; // B is loaded to $t1, E is loaded to $t2`
`C = B + F; // F is loaded to $t4`



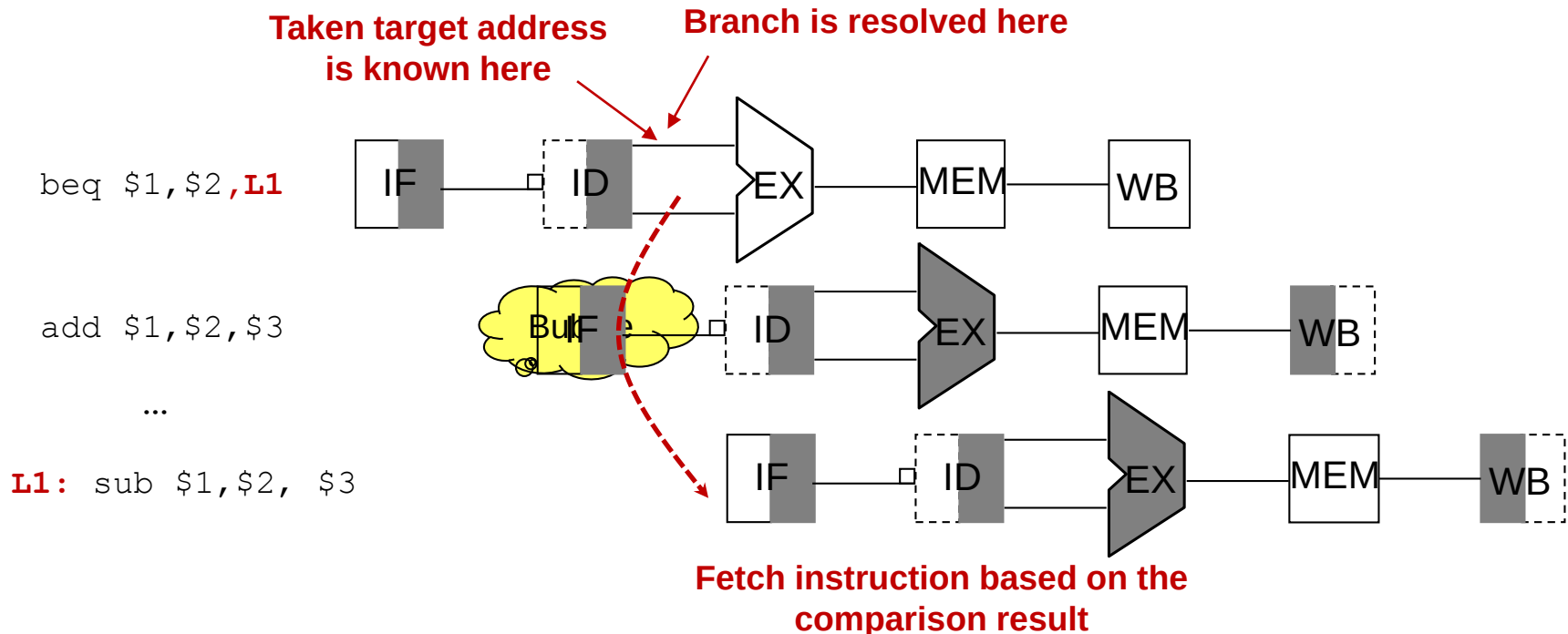
Control Hazard

- 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



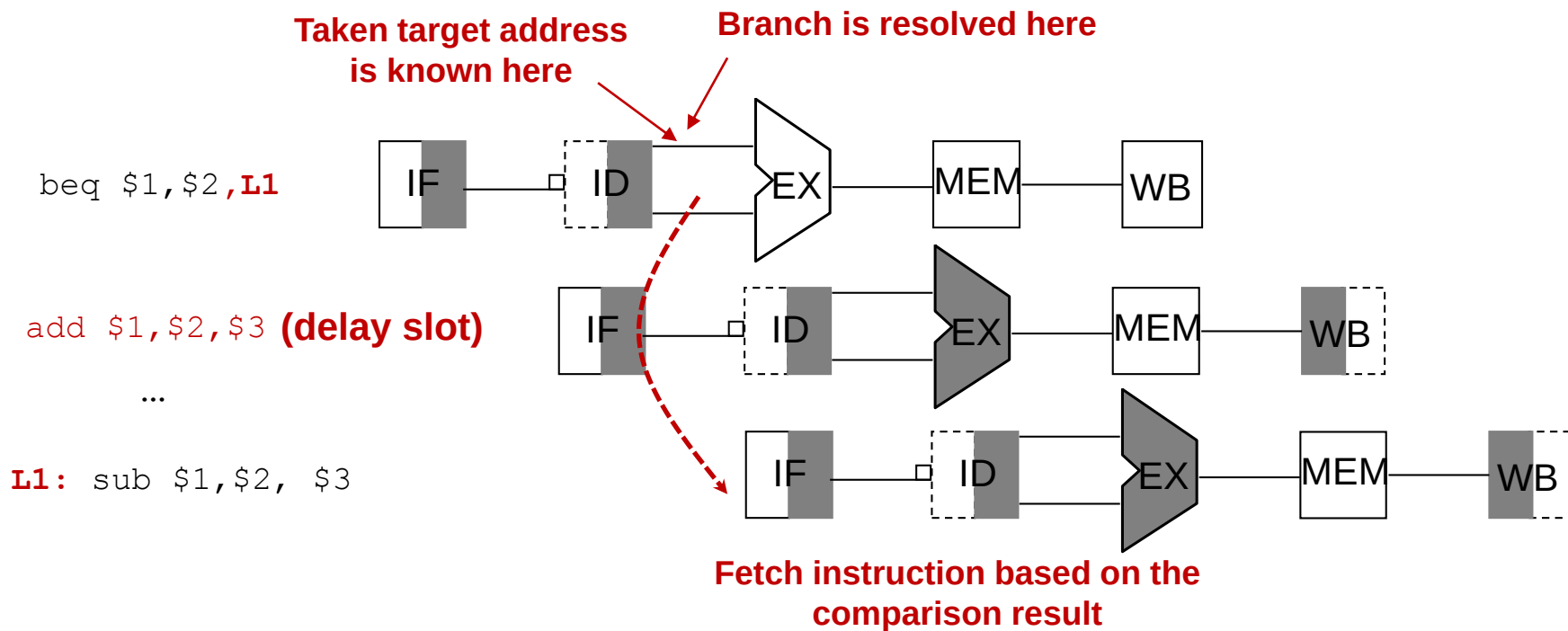
Reducing Control Hazard

- 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?



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 execution of the next instruction
 - Delay slot** is the slot right after delayed branch instruction



Delay Slot (Cont.)

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

```
// $s1 = a, $s2 = b, $s3 = c
// $t0 = d, $t1 = f
a = b + c;
if (d == 0) {f = f + 1;}
f = f + 2;
```



```
add $s1,$s2, $s3
bne $t0,$zero, L1
nop    //delay slot
addi $t1, $t1, 1
L1: addi $t1, $t1, 2
```

Can we do better?

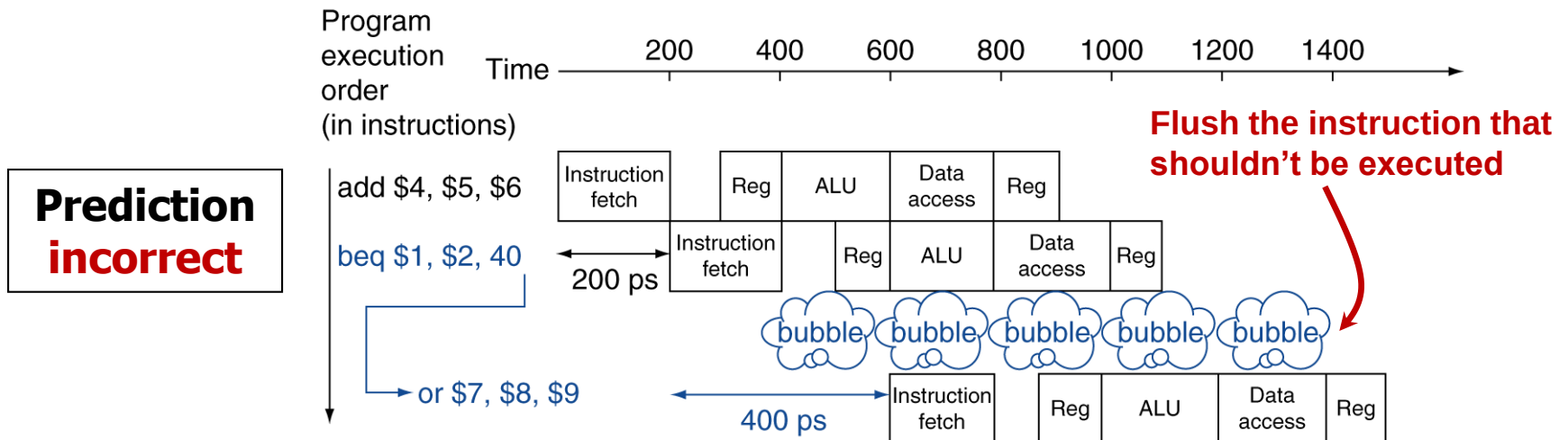
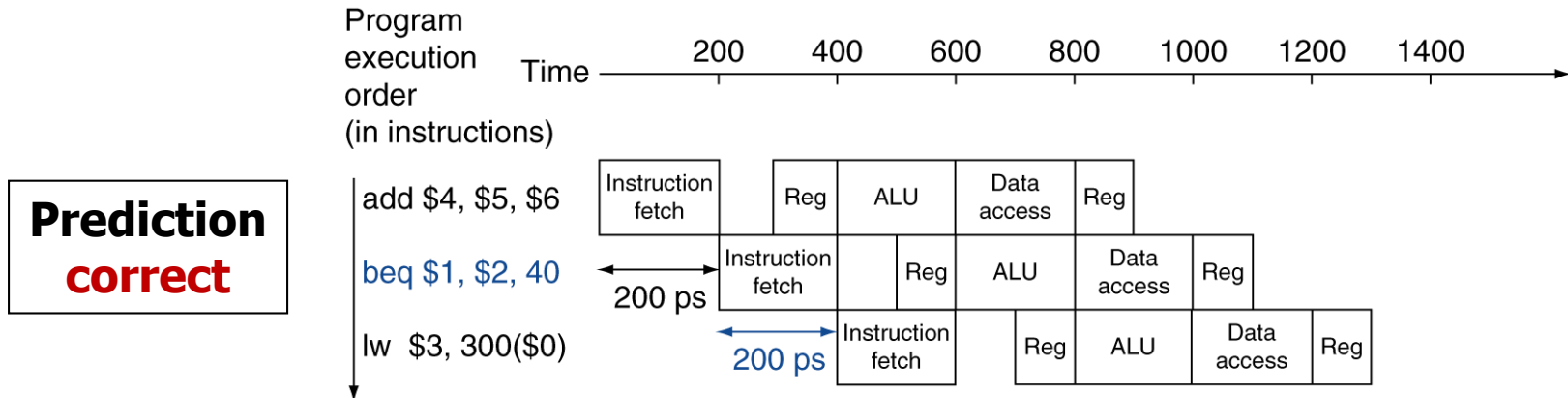
```
bne $t0, $zero, L1
add $s1,$s2,$s3 // delay slot
addi $t1, $t1, 1
L1: addi $t1, $t1, 2
```

Fill the delay slot with a useful and valid instruction

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 predictor (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



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)

Released · Canceled · Future · Microarchitecture name

