THE PROCESSOR

Jo, Heeseung

Introduction

CPU performance factors

- Instruction count
  - Determined by ISA and compiler
- CPI and Cycle time
  - Determined by CPU hardware

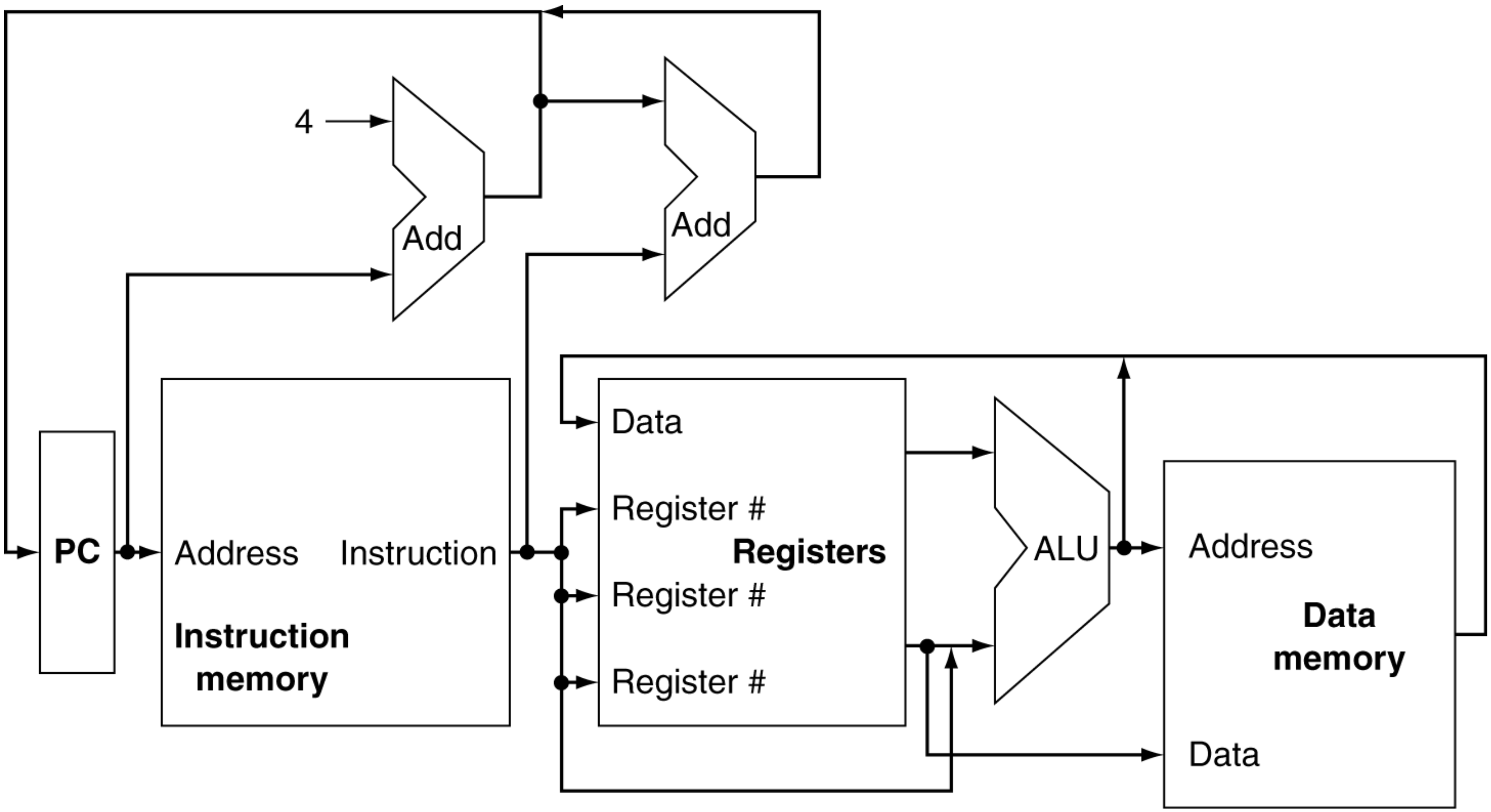
- But, remember that is not all

Instruction Execution

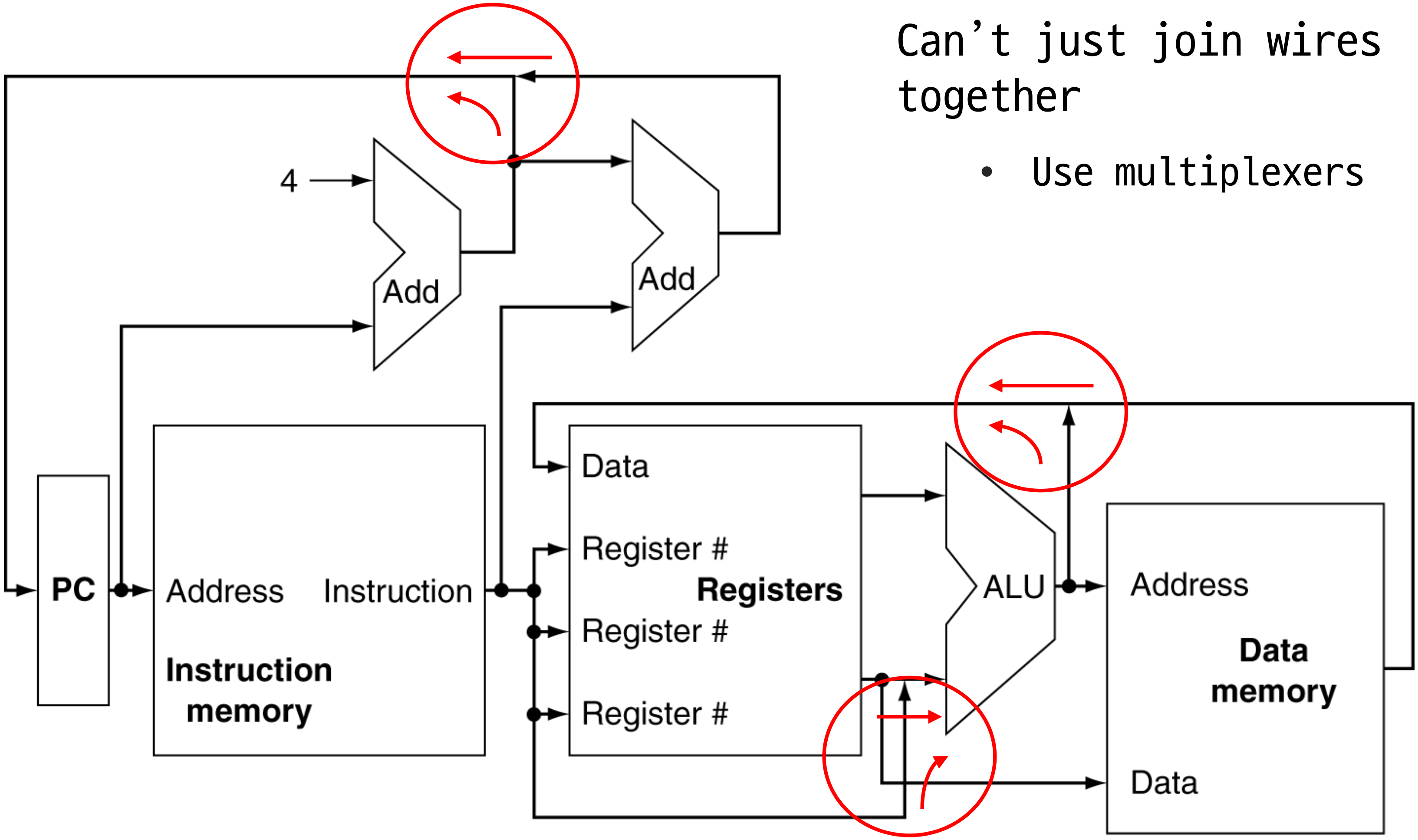
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

1. PC → instruction memory
2. Fetch instruction and decoding
3. Register numbers → register file, read registers
4. Depending on instruction class
    - Use ALU to calculate
        - Arithmetic result
        - Memory address for load/store
        - Branch target address
    - Access data memory for load/store
    - PC ← target address or PC + 4

CPU Overview



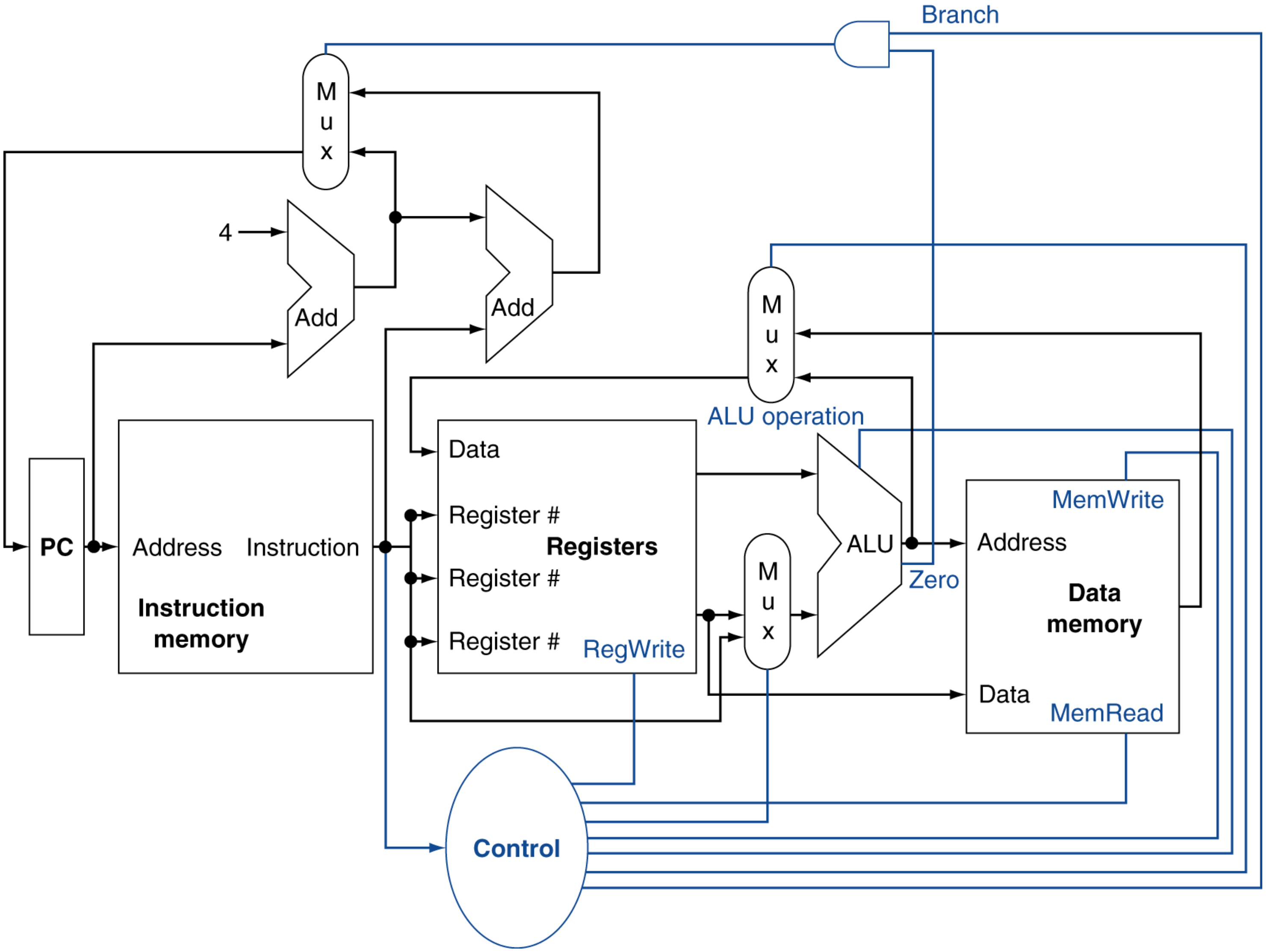
Multiplexers



Can’t just join wires together

- Use multiplexers

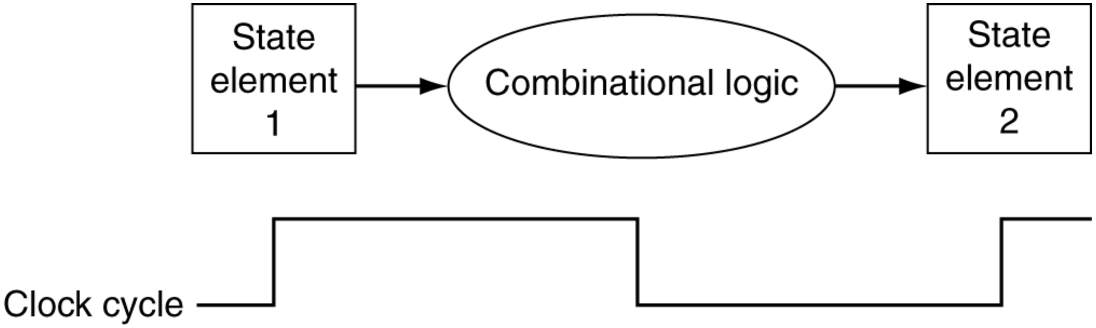
Control

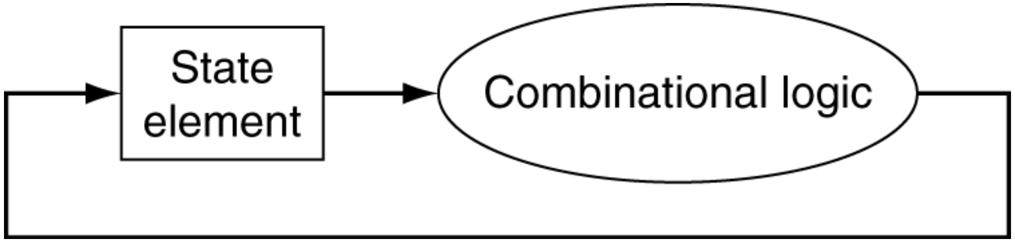

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
Data
Register #
Registers
Register #
Register #
RegWrite
PC
Address
Instruction
Instruction memory
M
u
x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control

Clocking Methodology

Combinational logic transforms data during clock cycles

- Between clock edges
- Input from state elements, output to state element
- Longest delay determines clock period



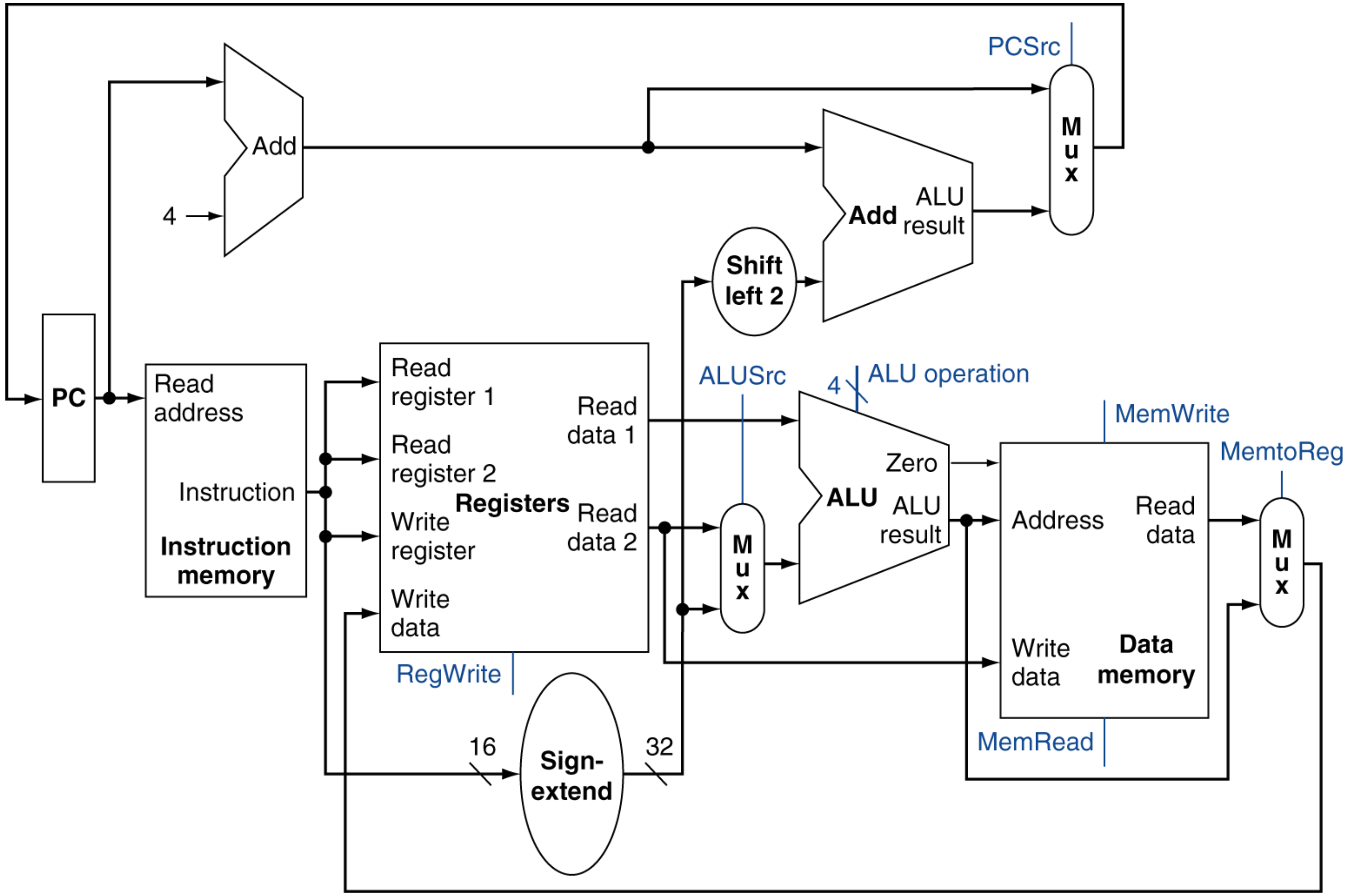


Building a Datapath

Datapath

- Elements that process data and addresses in the CPU
  - Registers, ALUs, mux's, memories, …

Refining the overview design



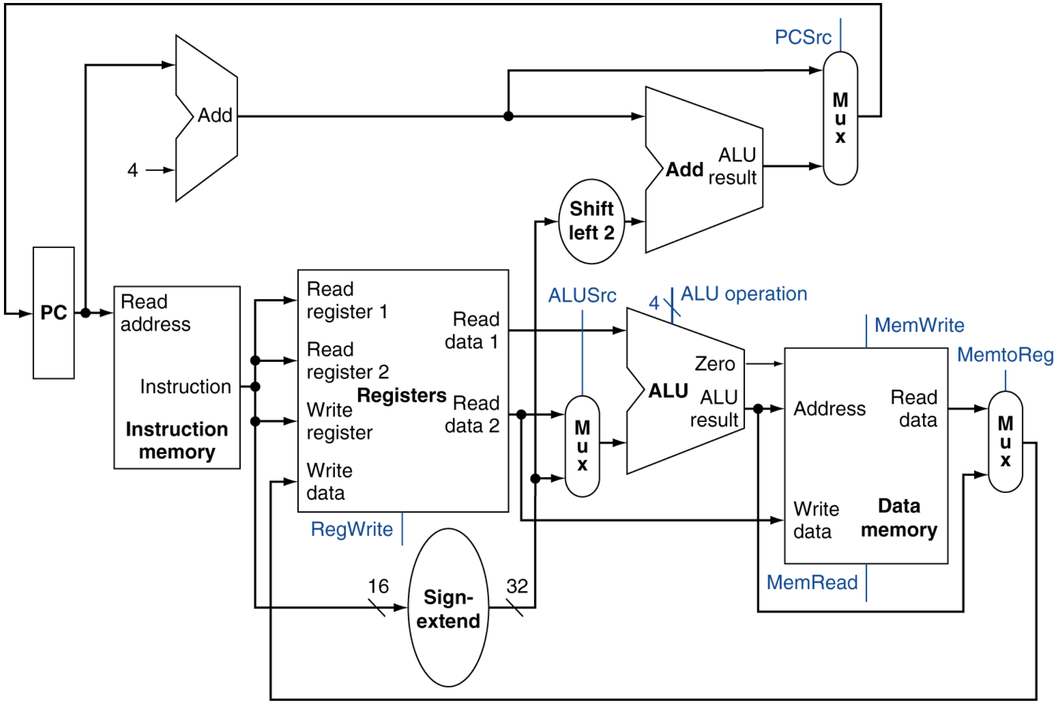
Memory in CPU

Instruction memory

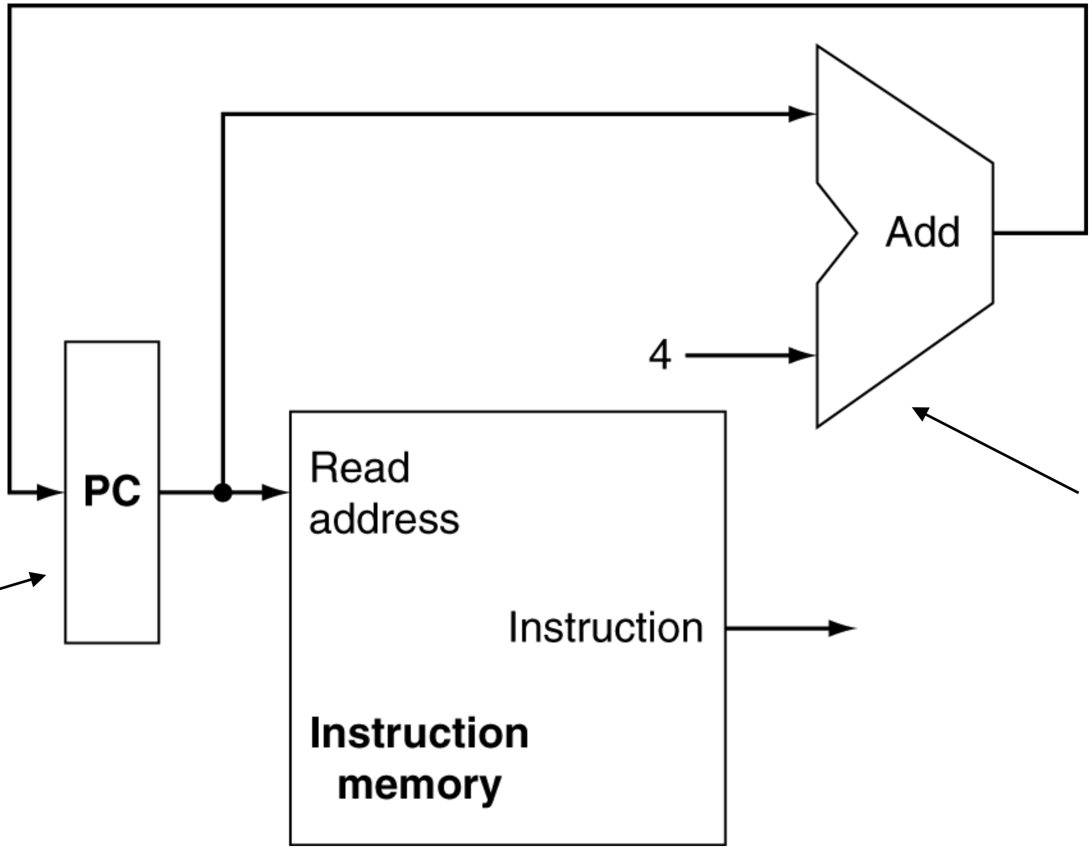
- Cache memory for instruction codes
- Replaced automatically
- e.g., L1 instruction cache

Data memory

- Cache memory for data (content of main memory)
- Replaced automatically
- e.g., L1 data cache



Instruction Fetch



32-bit register

Increment by 4 for next instruction

R-Format Instructions

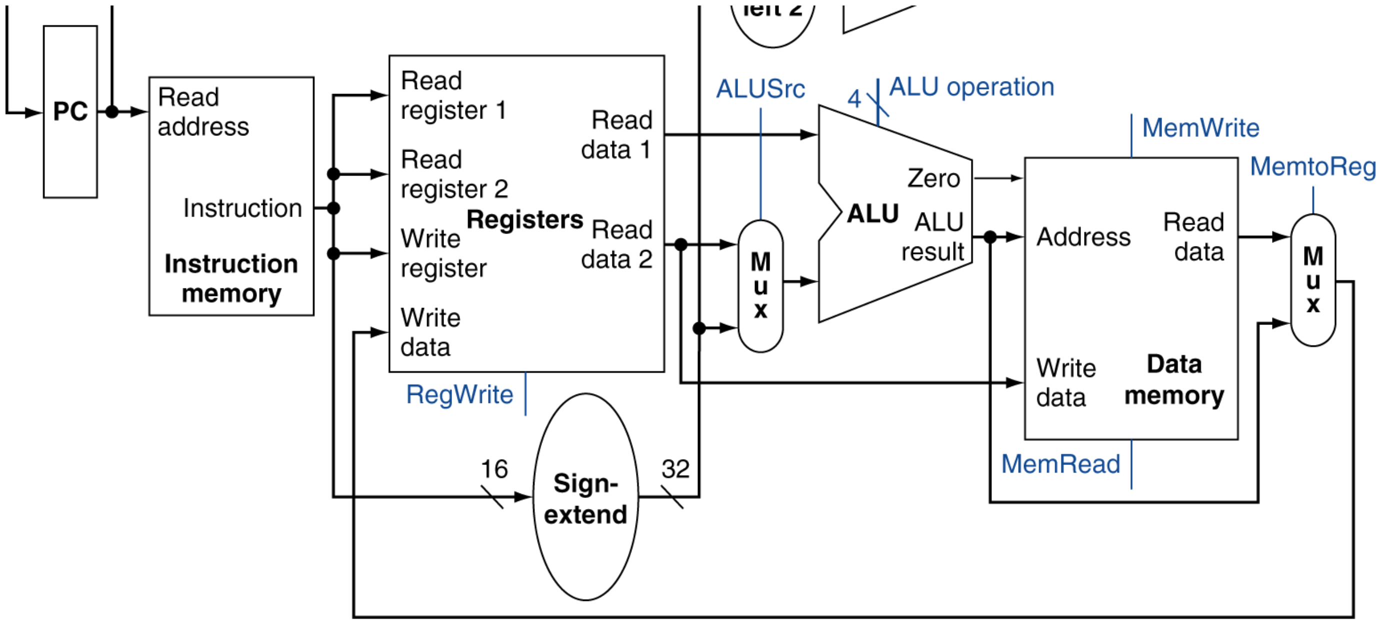
```
add $t0, $s1, $s2
```

1. Read two register operands
2. Perform arithmetic/logical operation
3. Write register result

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

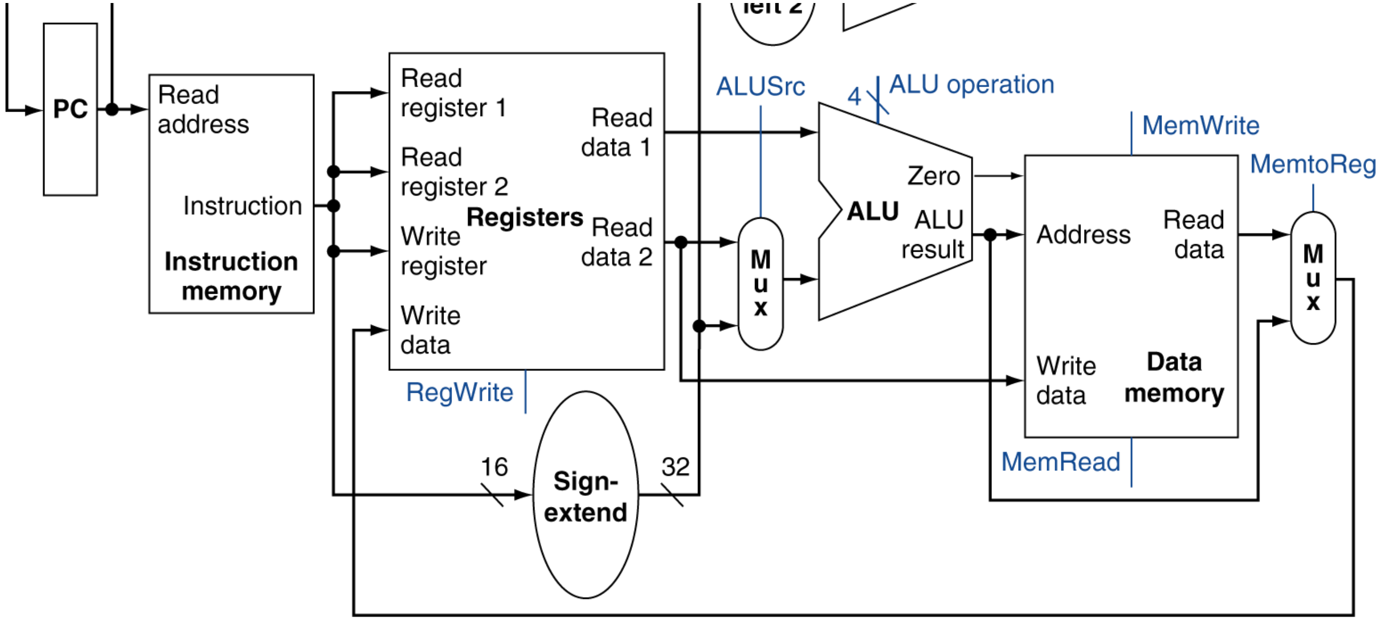


Load/Store Instructions

```
lw  $t0, 32($s3)
```

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

1. Read register operands
2. Calculate address using 16-bit offset
    1. Use ALU, but sign-extend offset
3. Load: Read memory and update register
4. Store: Write register value to memory

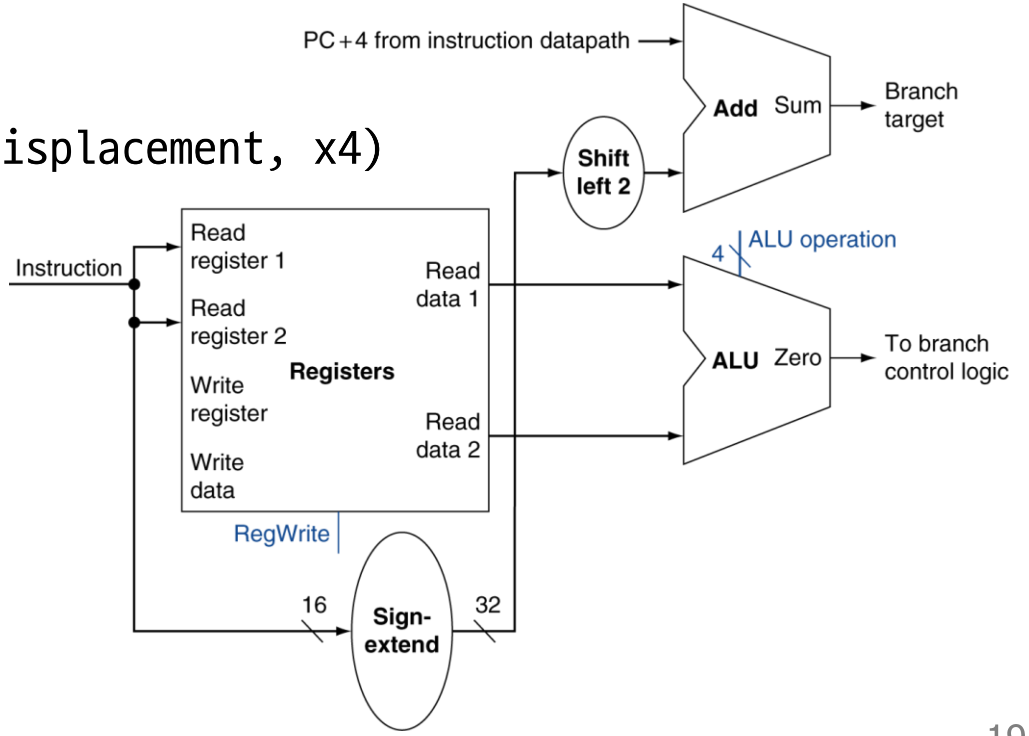


Branch Instructions

bne $t0, $s5, Exit

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

1. Read register operands
2. Compare operands
   - Use ALU, subtract and check Zero output
3. Calculate target address
   - Sign-extend displacement
   - Shift left 2 places (word displacement, x4)
   - Add to PC + 4
     - Already calculated by instruction fetch



PC-relative addressing

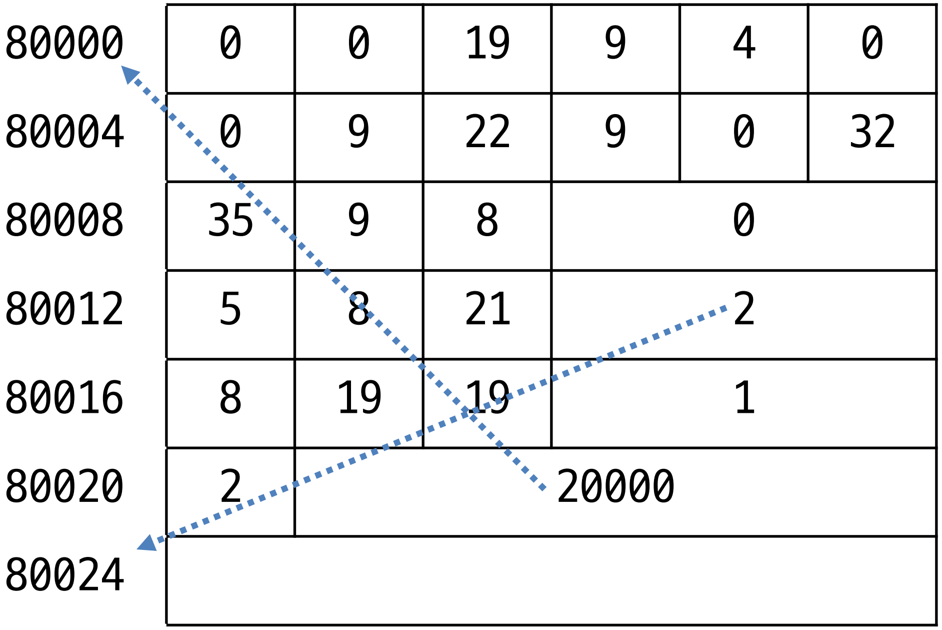
- Target address = PC + offset × 4

Branch Instructions

PC-relative addressing

- Target address = PC + offset × 4
- PC already incremented by 4 by this time

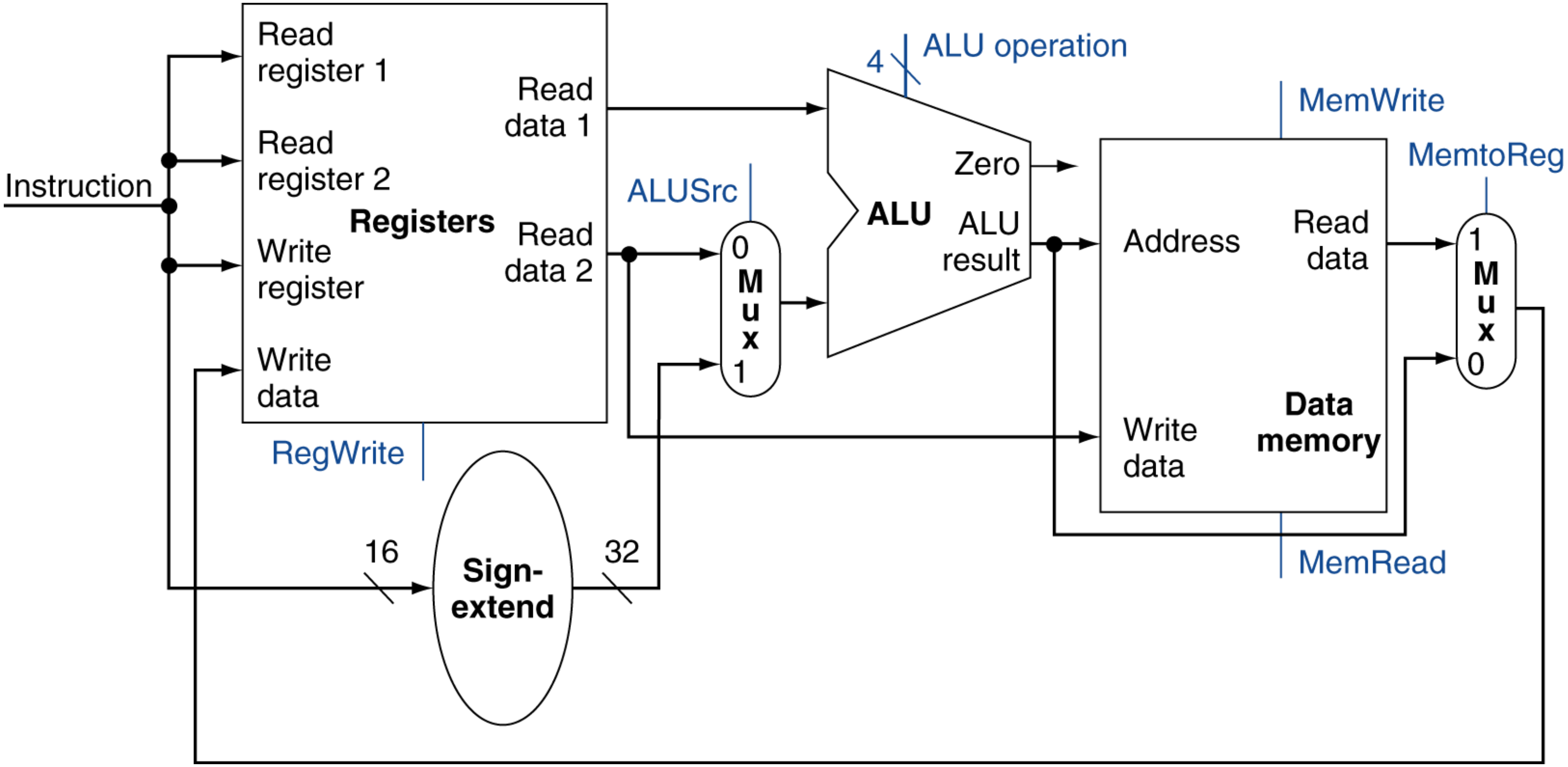
```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

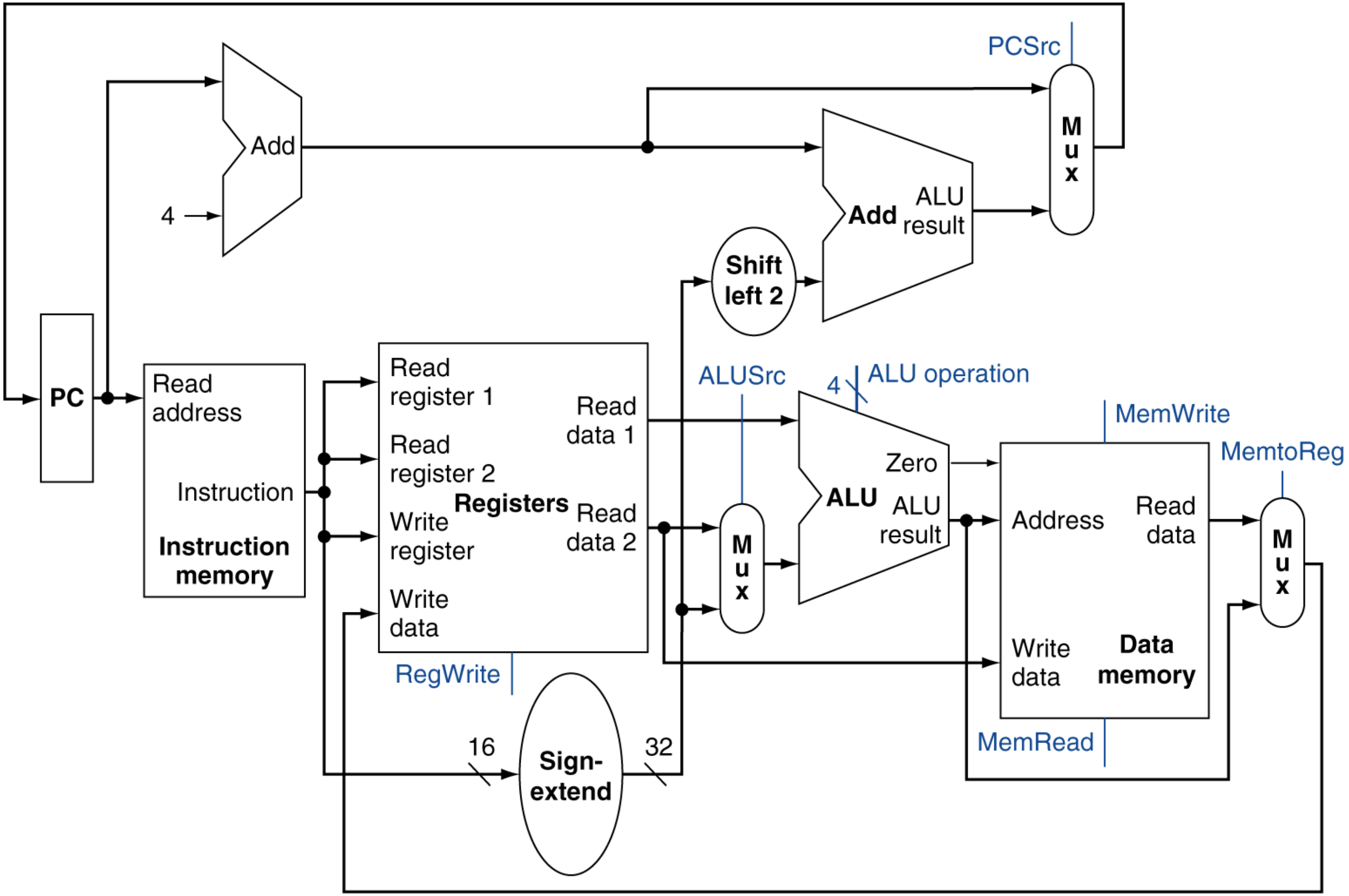
R-Type/Load/Store Datapath

```
add $t0, $s1, $s2
```



Full Datapath

```
lw  $t0, 32($s3)
```



Performance Issues

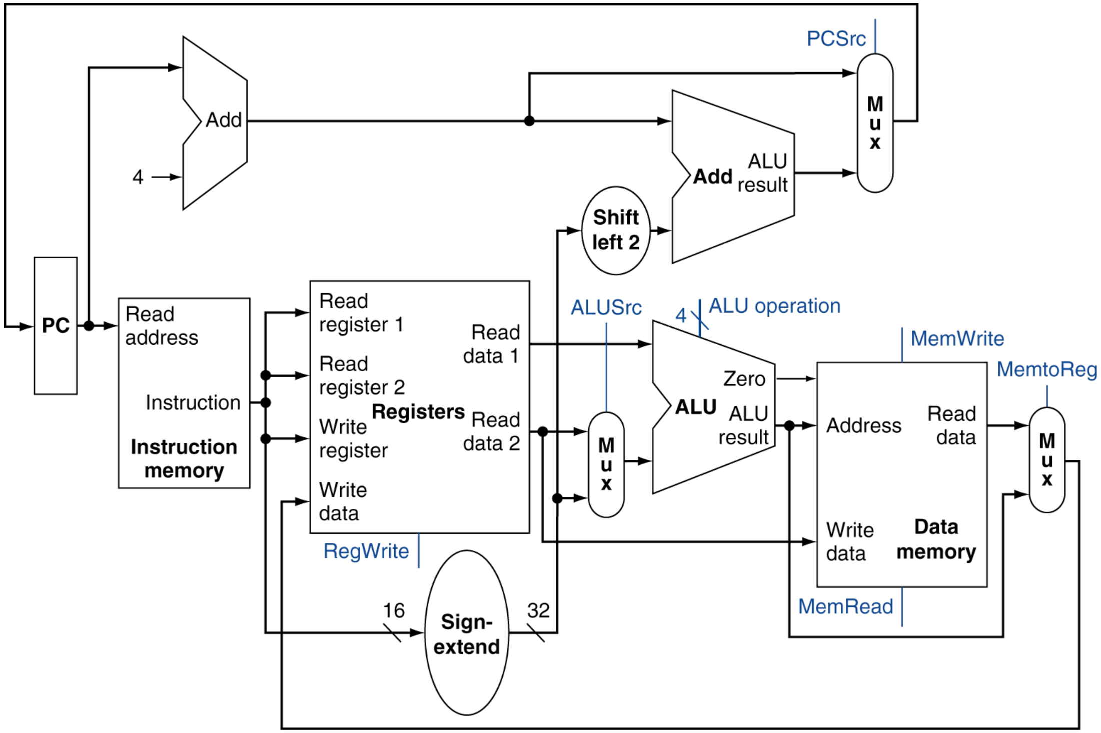
Longest delay determines clock period

- Critical path: load instruction
- Instruction memory → register file → ALU → data memory → register file

Various period for different instructions is not feasible

- Violates design principle
- Making the common case fast

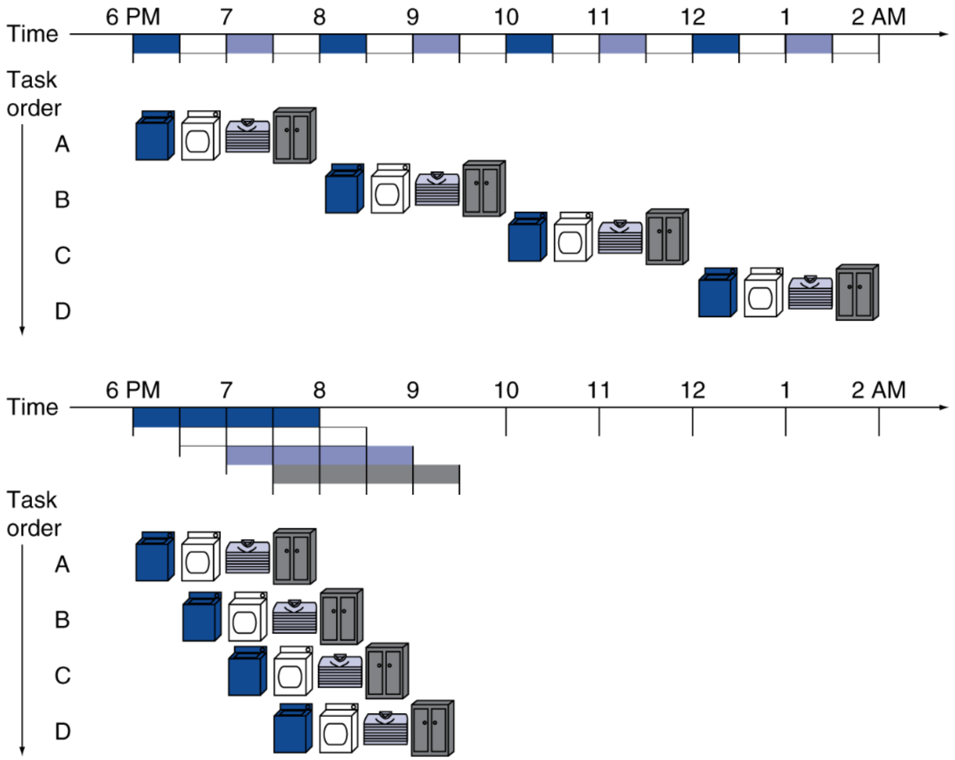
We will improve performance by pipelining



Pipelining Analogy

Pipelined laundry: overlapping execution

- Parallelism improves performance



Four loads:

- Speedup
  = 8/3.5 = 2.3

Non-stop:

- Speedup
  ≈ 4
  = number of stages

MIPS Pipeline

Five stages, one step per stage

- IF: Instruction fetch from memory
- ID: Instruction decode & register read
- EX: Execute operation or calculate address
- MEM: Access memory operand
- WB: Write result back to register

Pipeline Performance

Assume time for stages is

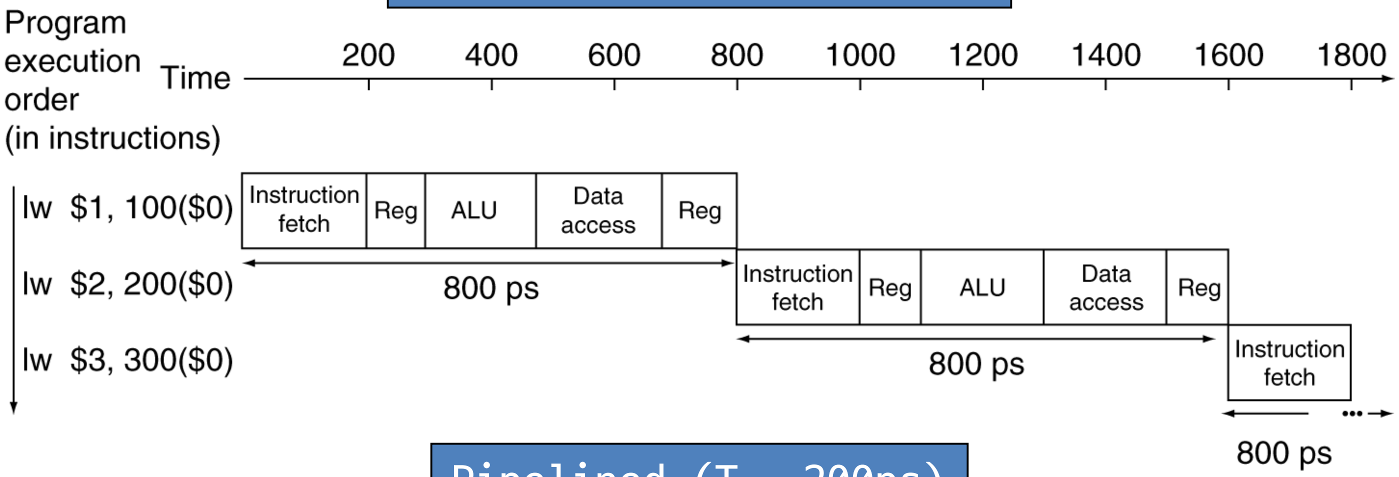
- 100ps for register read or write
- 200ps for other stages

Compare pipelined datapath with single-cycle datapath

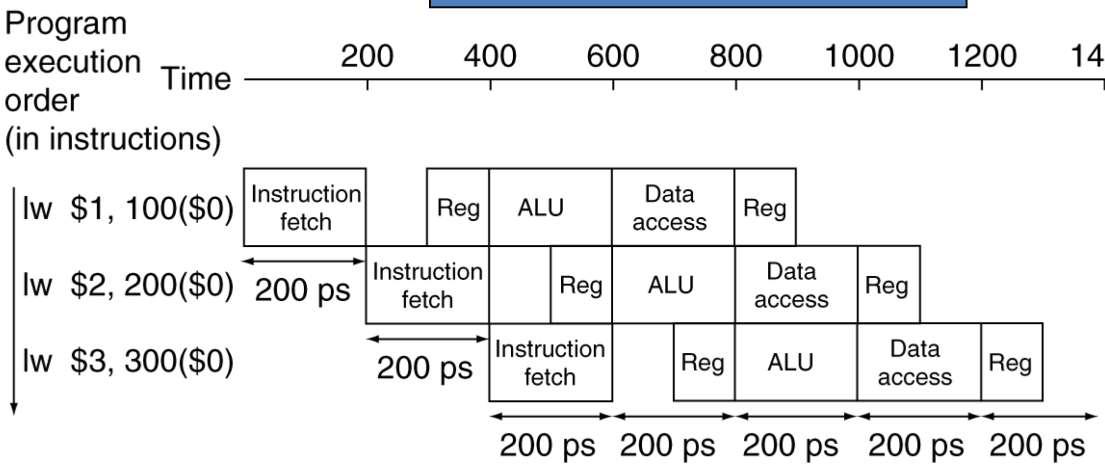
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

If all stages are balanced

- i.e., all take the same time

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

Speedup is due to increased throughput

- Latency (time for each instruction) does not decrease

Pipelining and ISA Design

MIPS ISA designed for pipelining

- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
  - Can decode and read registers in one step

Hazards

Situations that prevent starting the next instruction in the next cycle

Structure hazards

- A required resource is busy

Data hazard

- Need to wait for previous instruction to complete its data read/write

Control hazard

- Deciding on control action depends on previous instruction

Structure Hazards

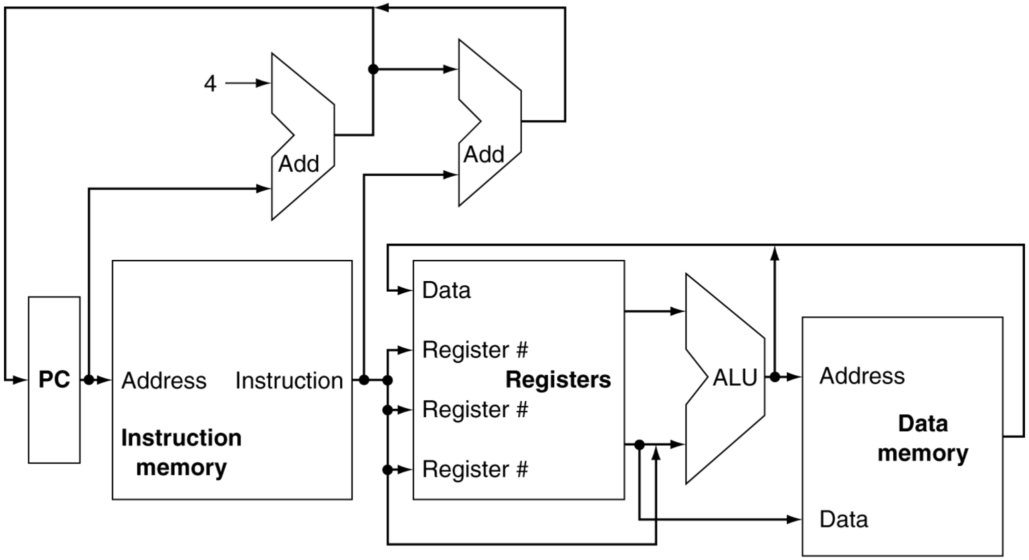
Conflict for use of a resource

If MIPS pipeline uses a single memory (1 inst/data memory)

- Load/store requires data access
- Instruction fetch would have to *stall* for that cycle
  - Would cause a pipeline "bubble"

Hence, pipelined datapaths require separate instruction/data memories

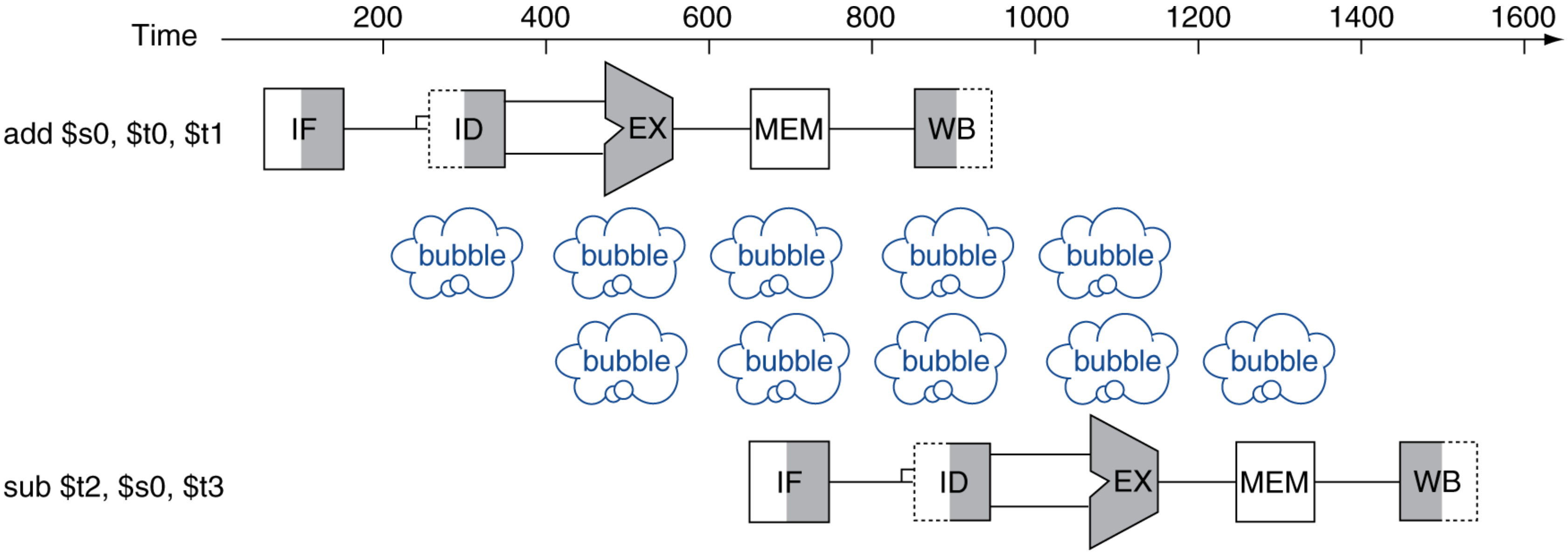
- Or separate instruction/data caches



Data Hazards

An instruction depends on completion of data access by a previous instruction

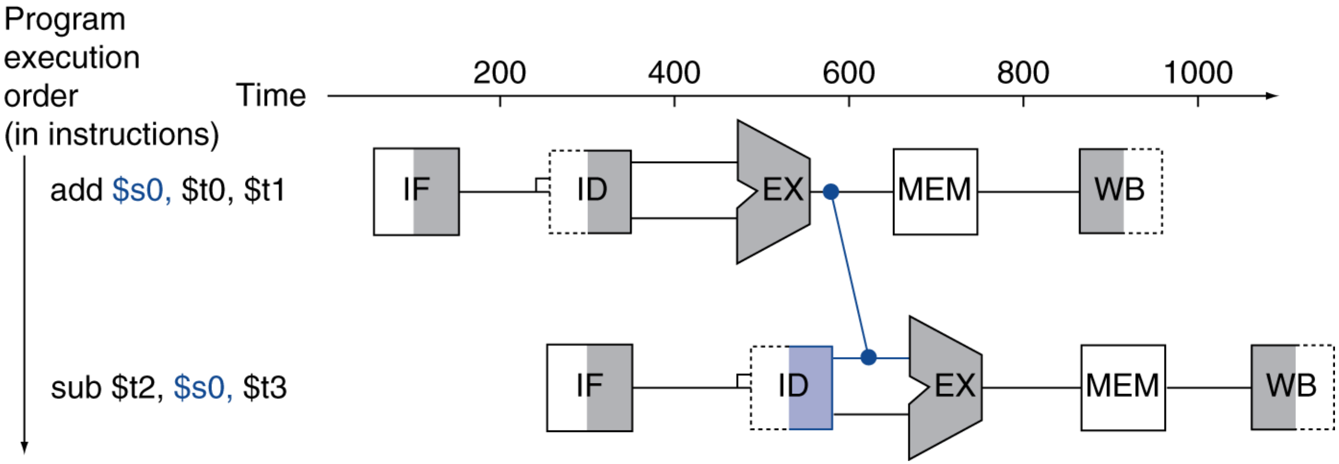
```
add     $s0, $t0, $t1
sub     $t2, $s0, $t3
```



Forwarding (aka Bypassing)

Use result when it is computed

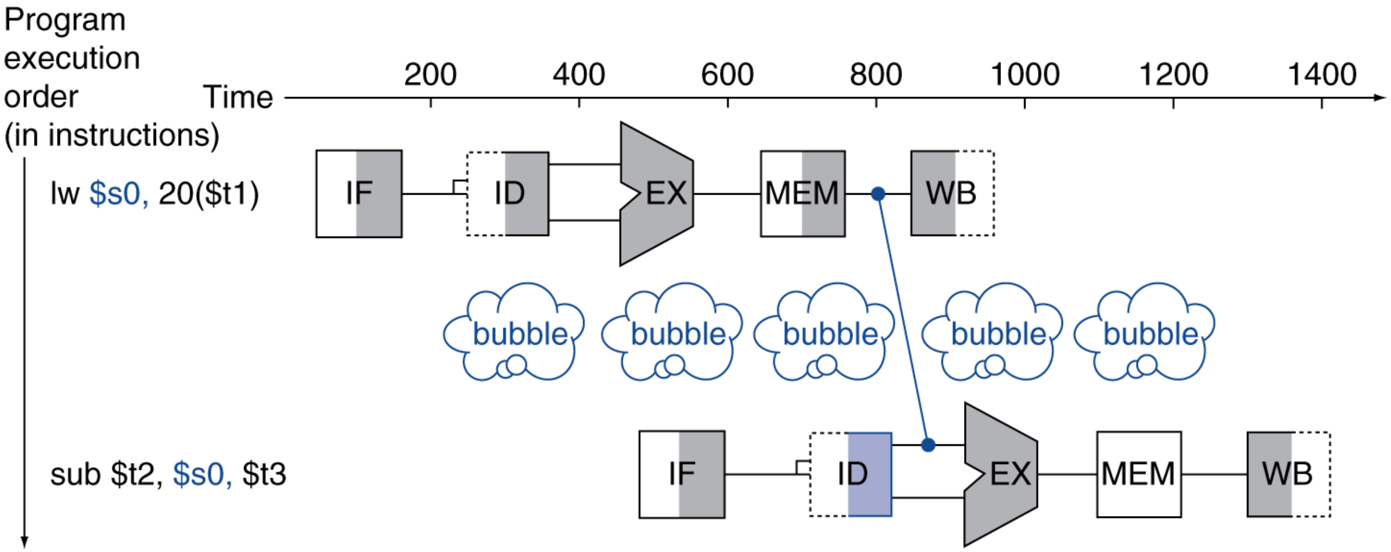
- Don't wait for it to be stored in a register
- Requires extra connections in the datapath



Load-Use Data Hazard

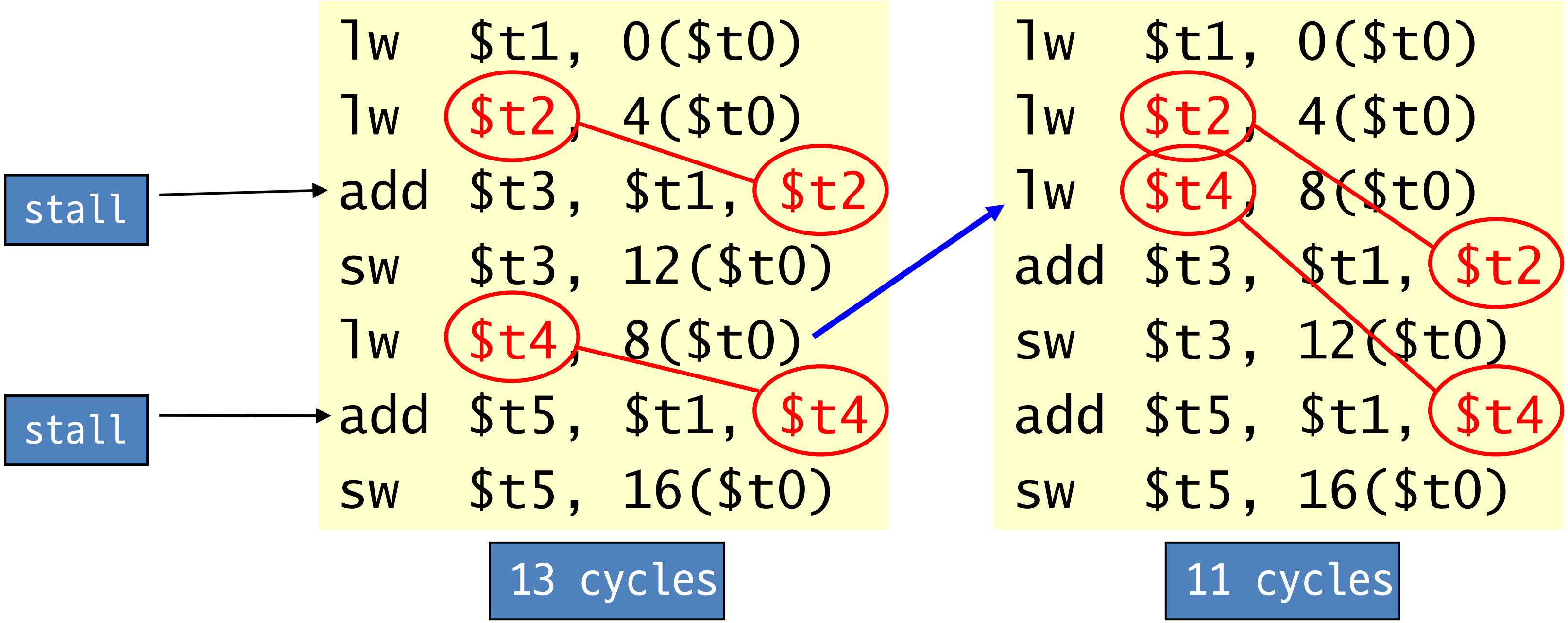
Can't always avoid stalls by forwarding

- If value not computed when needed
- Can't forward backward in time!



Code Scheduling to Avoid Stalls

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



```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

13 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles

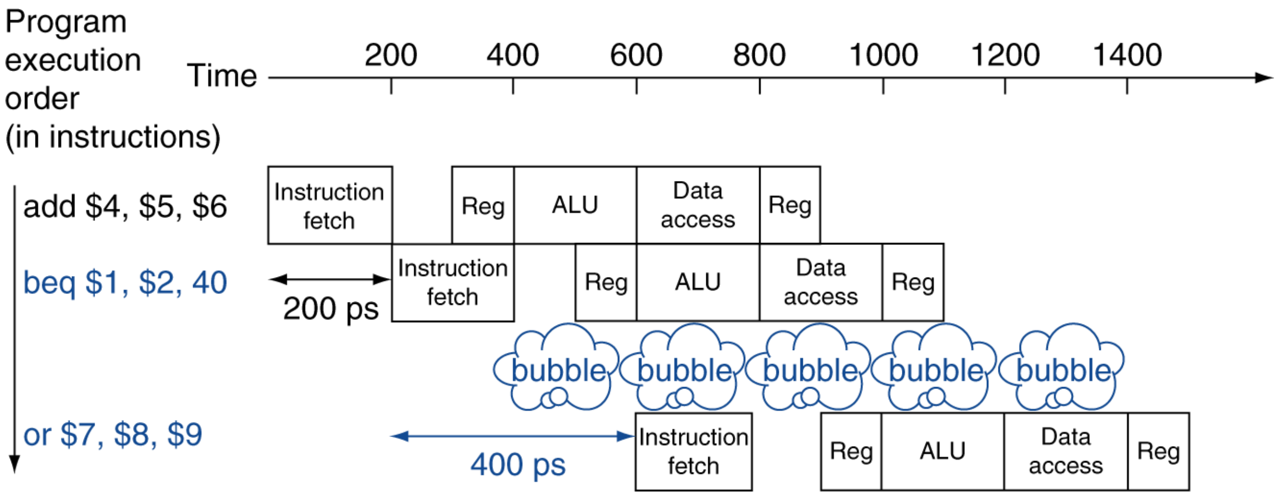
Control Hazards

Branch determines flow of control

- Fetching next instruction depends on branch outcome
- Pipeline can't always fetch correct instruction
  - Still working on ID stage of branch

Stall on Branch

Wait until branch outcome determined before fetching next instruction



Branch Prediction

Longer pipelines can't readily determine branch outcome early

- Stall penalty becomes unacceptable

Predict outcome of branch

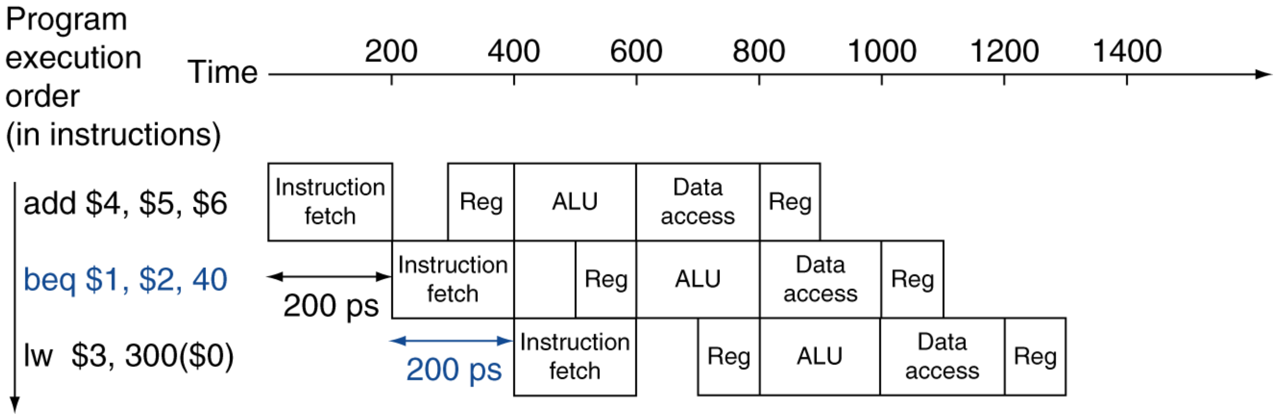
- Only stall if prediction is wrong

In MIPS pipeline

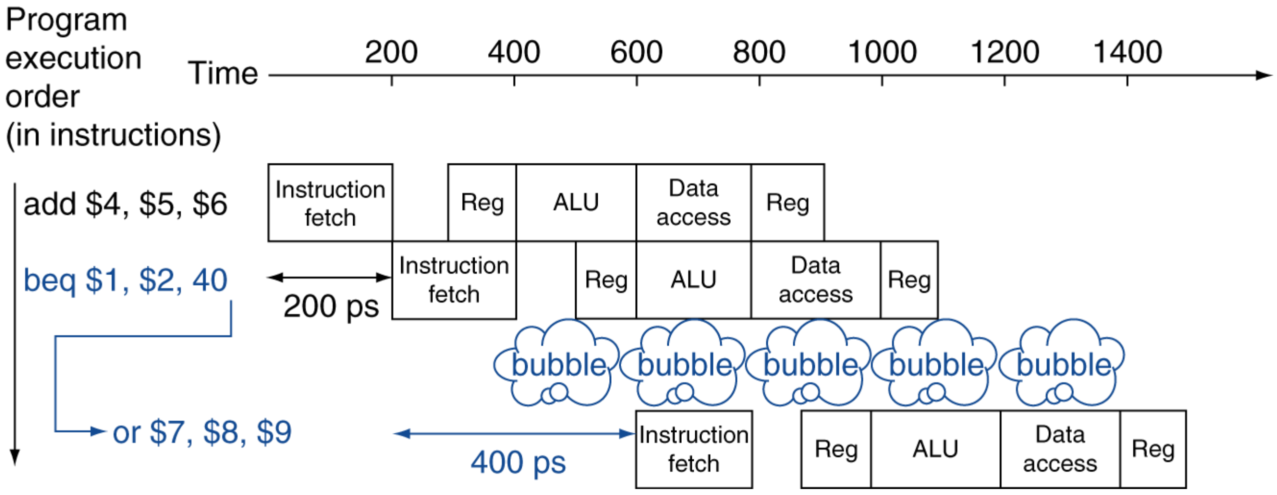
- Can predict branches not taken
- Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



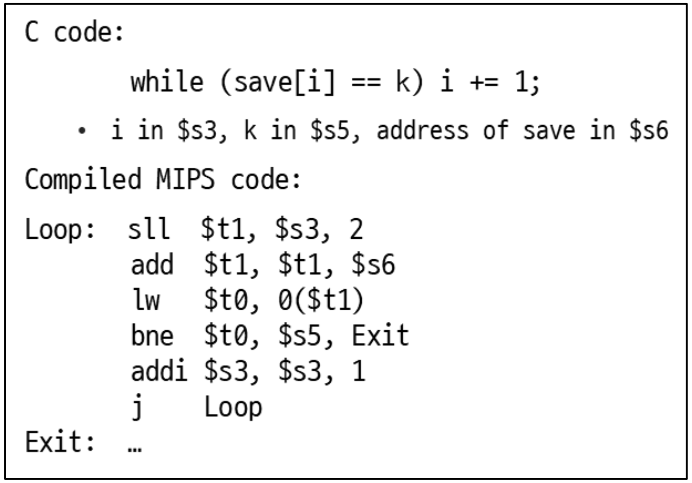
More-Realistic Branch Prediction

Static branch prediction

- Based on typical branch behavior
- Example: loop and if-statement branches
  - Predict backward branches taken
  - Predict forward branches not taken

Dynamic branch prediction

- Hardware measures actual branch behavior
  - e.g., record recent history of each branch
- Assume future behavior will continue the trend
  - When wrong, stall while re-fetching, and update history



C code:

```
while (save[i] == k) i += 1;
```

- i in $s3, k in $s5, address of save in $s6

Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Dynamic Branch Prediction

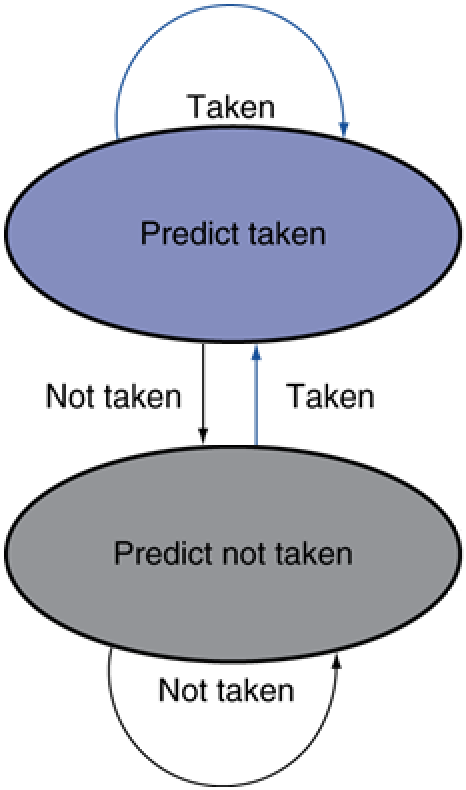
In deeper and superscalar pipelines, branch penalty is more significant

Use dynamic prediction

- Branch prediction buffer (aka branch history table)
- Indexed by recent branch instruction addresses
- Stores outcome (taken/not taken)
- To execute a branch
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction

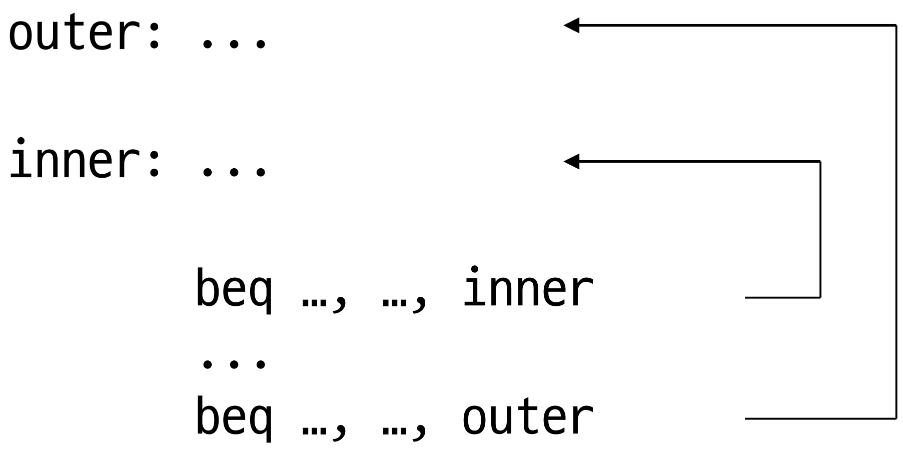
1-Bit Predictor

Only change prediction on a misprediction



1-Bit Predictor: Shortcoming

Inner loop branches mispredicted twice!

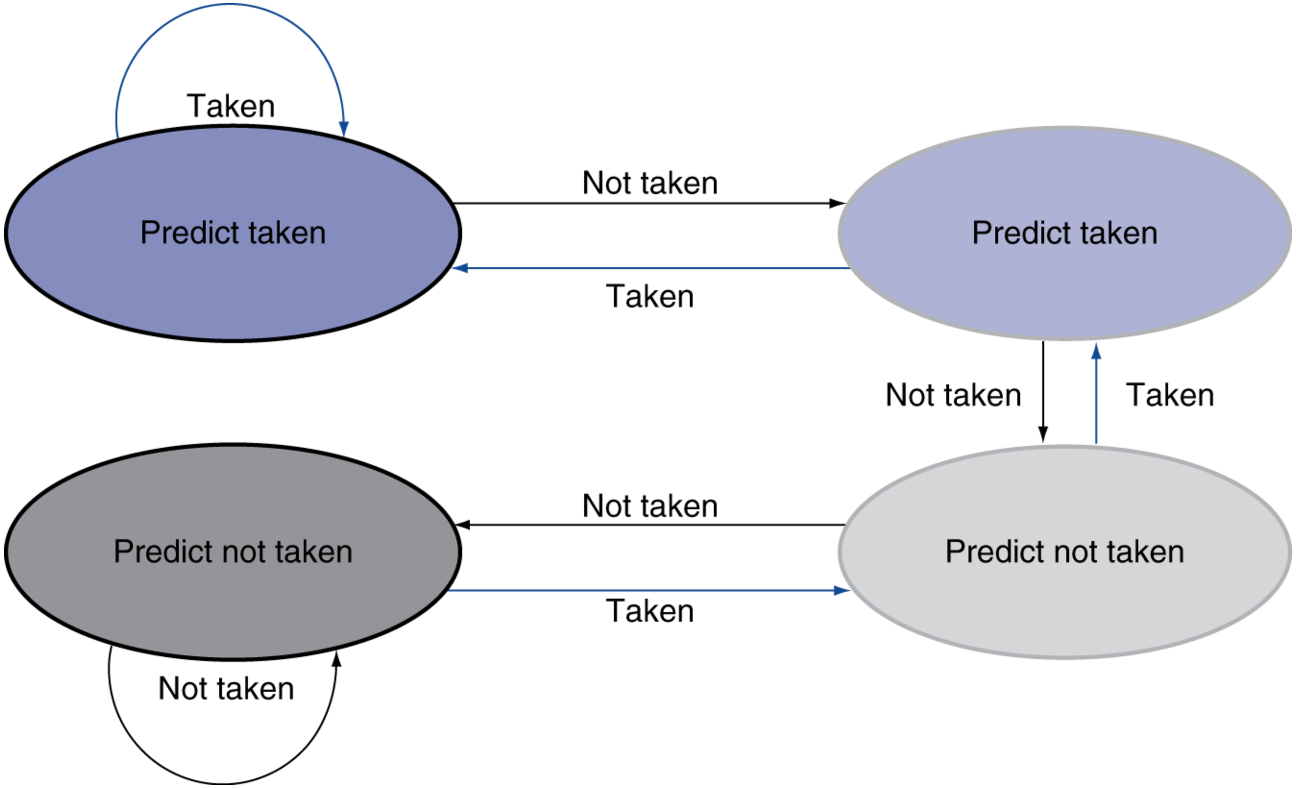


```
outer: ...

inner: ...

            beq …, …, inner
            ...
            beq …, …, outer
```

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

Only change prediction on two successive mispredictions



Pipeline Summary

Pipelining improves performance by increasing instruction throughput

- Executes multiple instructions in parallel
- Each instruction has the same latency

Subject to hazards

- Structure, data, control

Instruction set design affects complexity of pipeline implementation