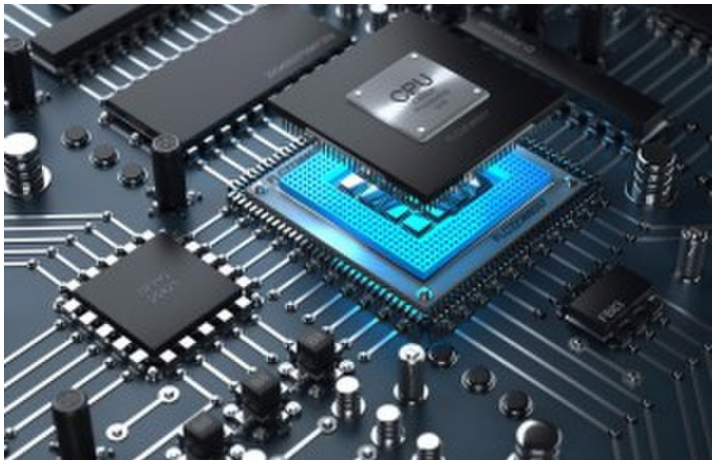


CPU Datapaths 3: Intro to Pipelining

CS 154: Computer Architecture

Lecture #13

Winter 2020

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB

Administrative

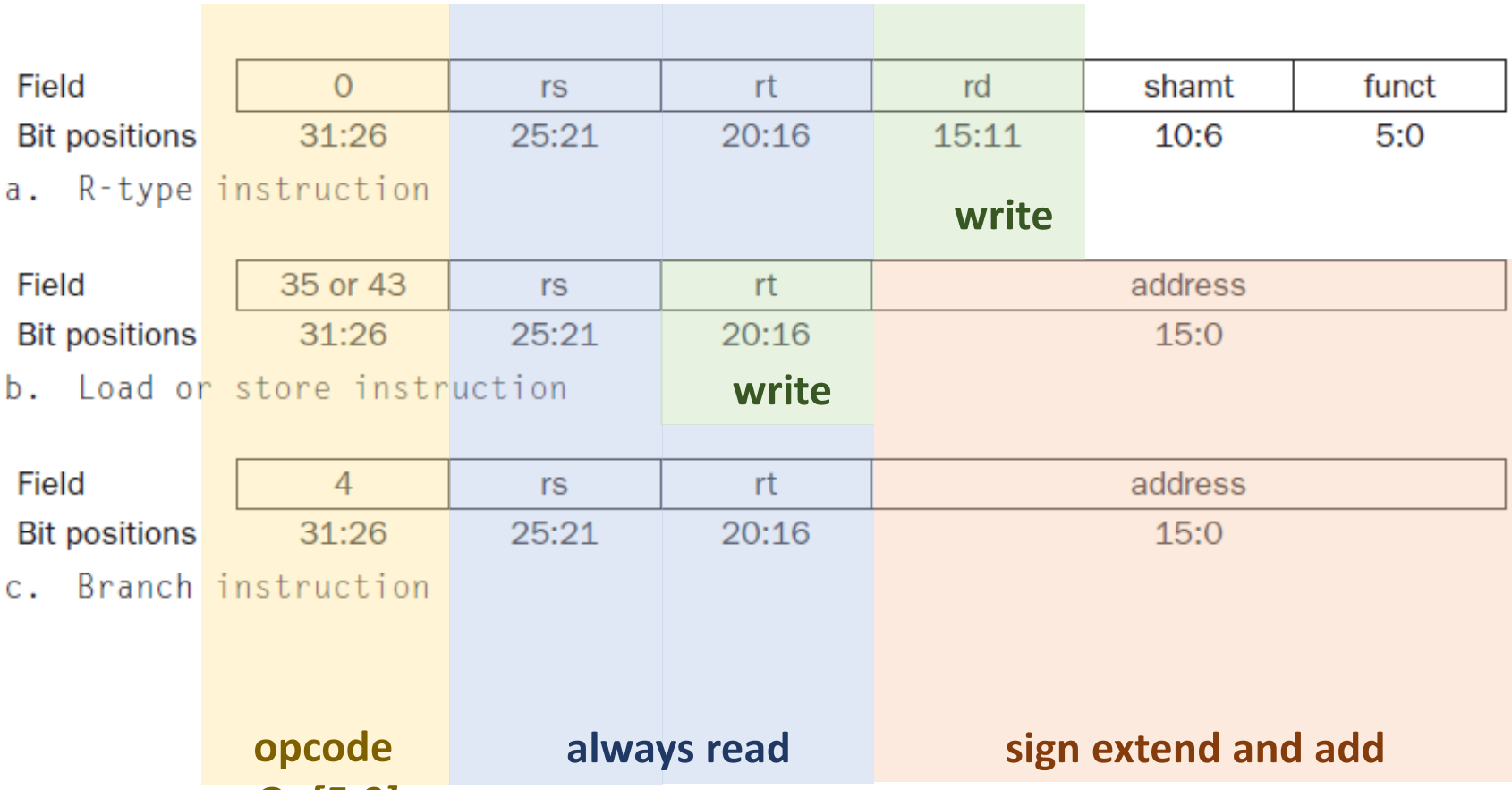
- Talk next week – must attend
  - Tuesday at 5:00 PM

Lecture Outline

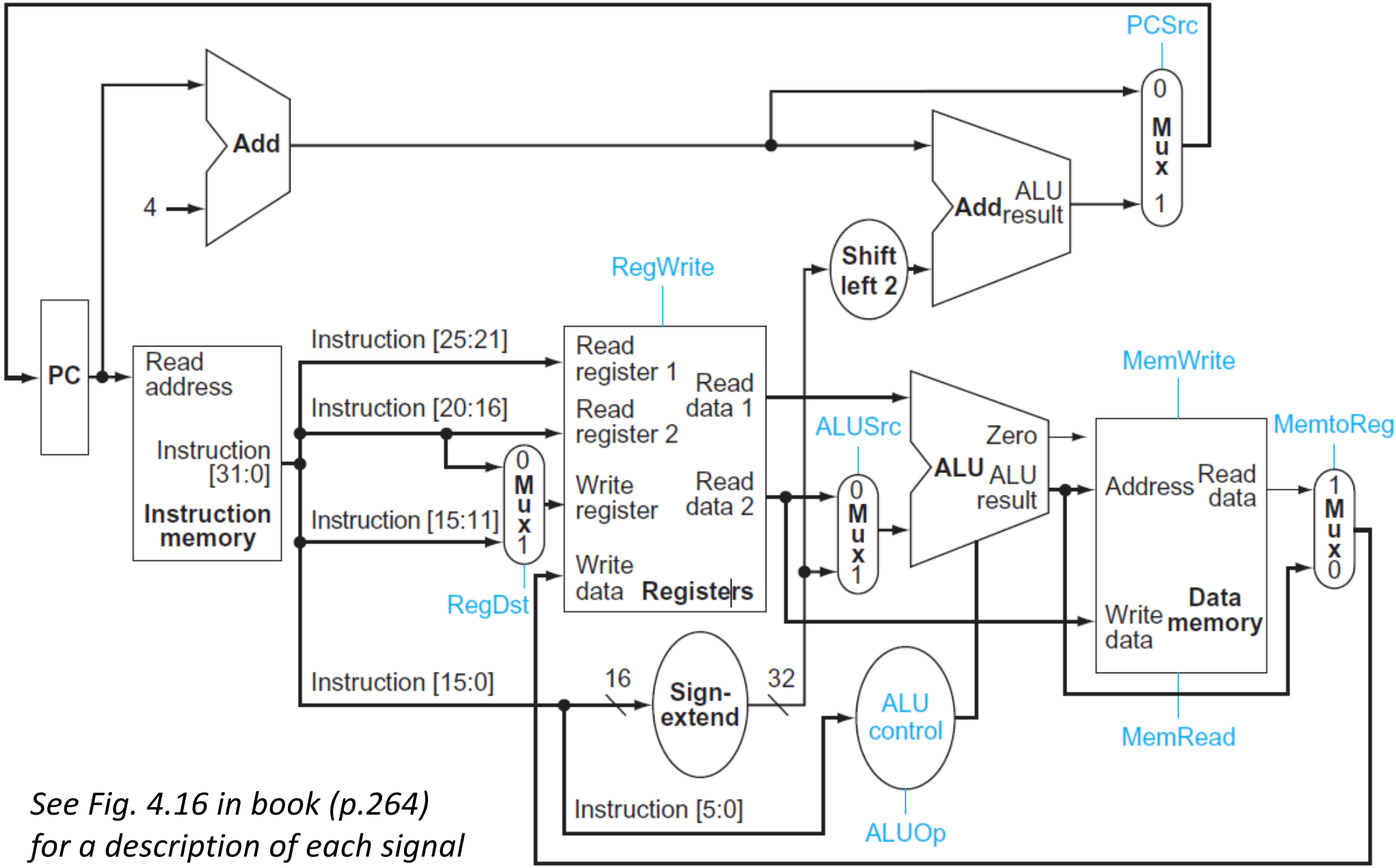
- Full Single-Cycle Datapaths
- Pipelining

The Main Control Unit

- Control signals derived (i.e. decoded) from instruction



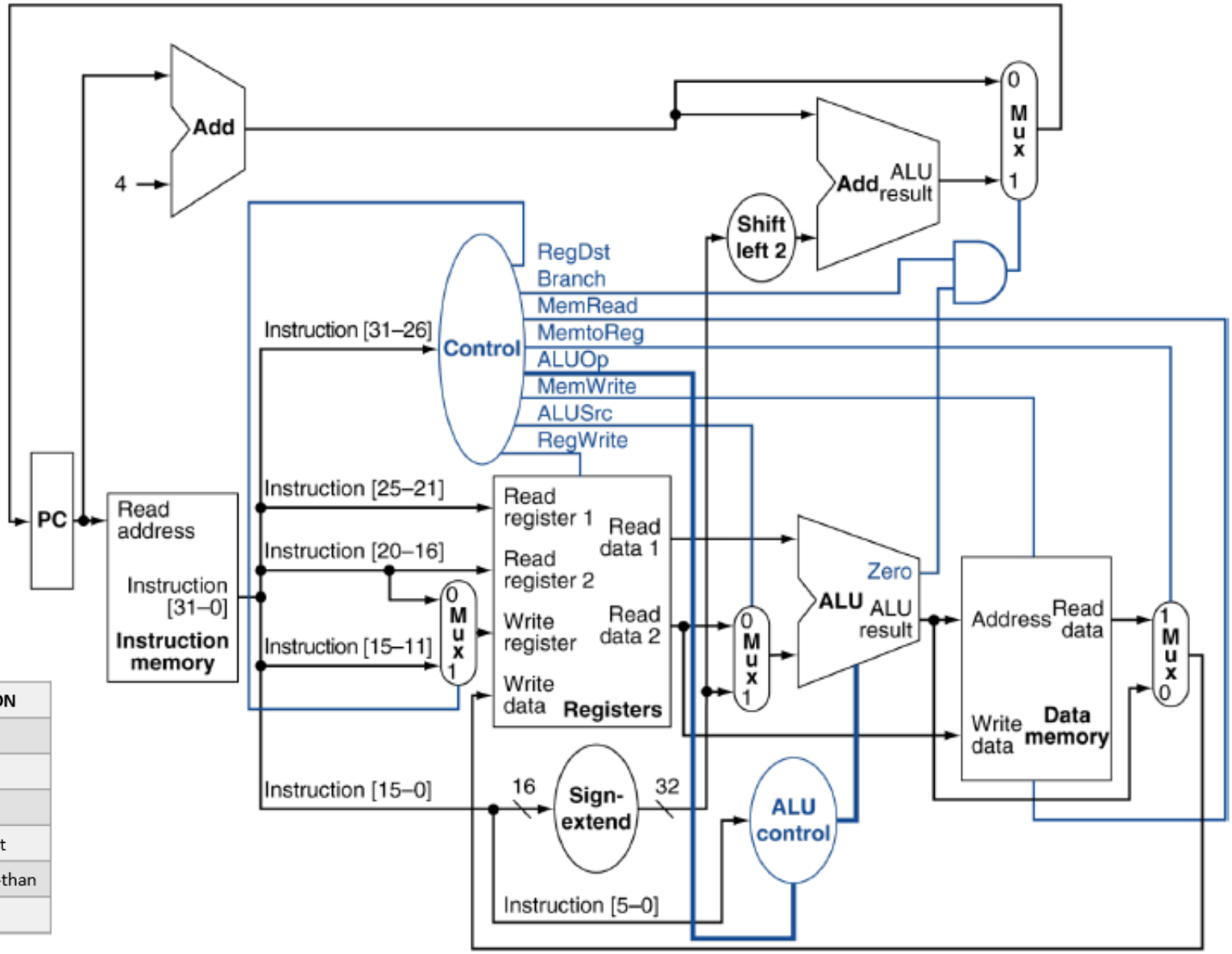
Full Datapath showing 7 Control Signals



See Fig. 4.16 in book (p.264)
for a description of each signal

One Control Unit to Set them All…

my precious



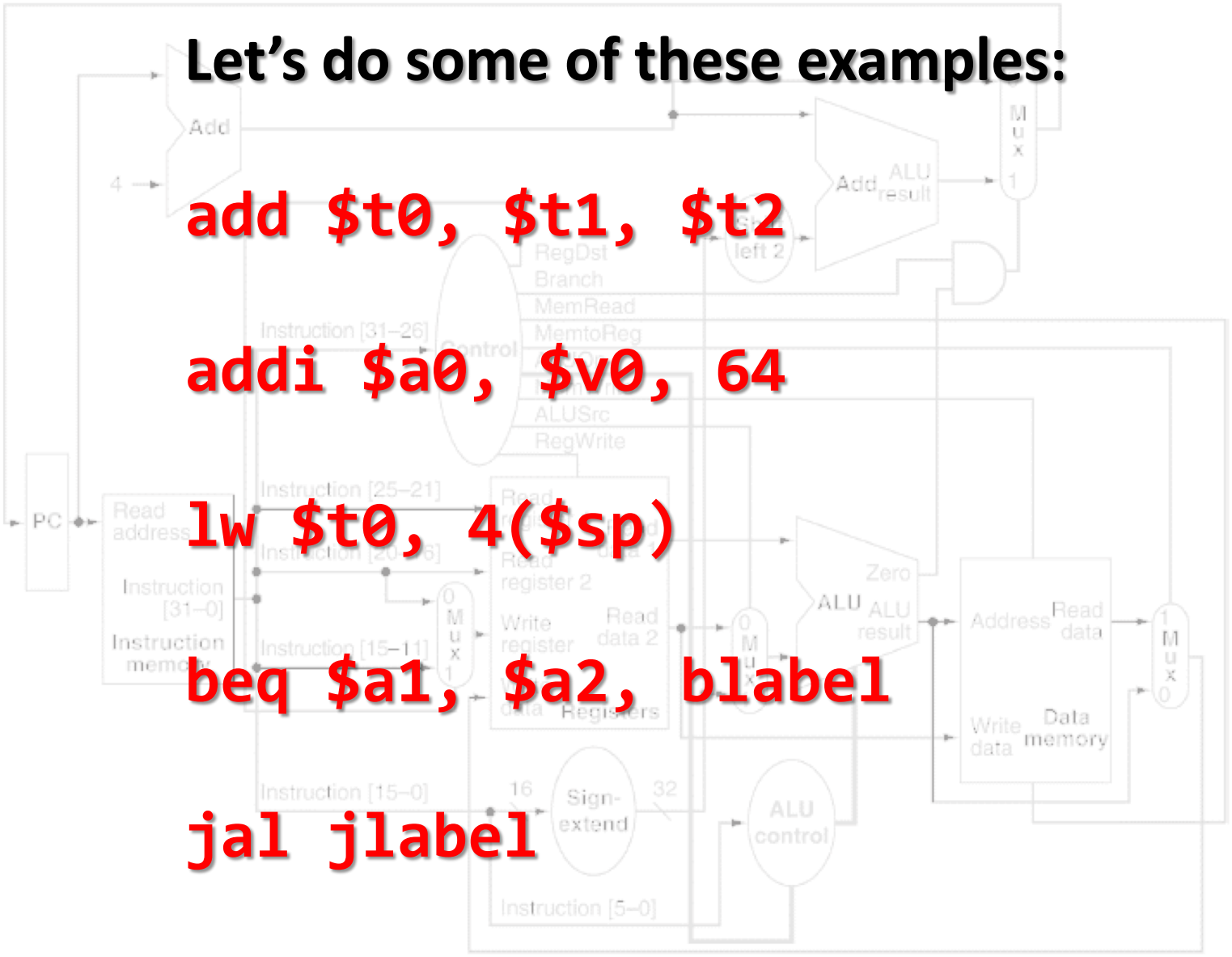
ALUOp	FUNCTION
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

One Control Unit to Set them All…

my precious

Let's do some of these examples:

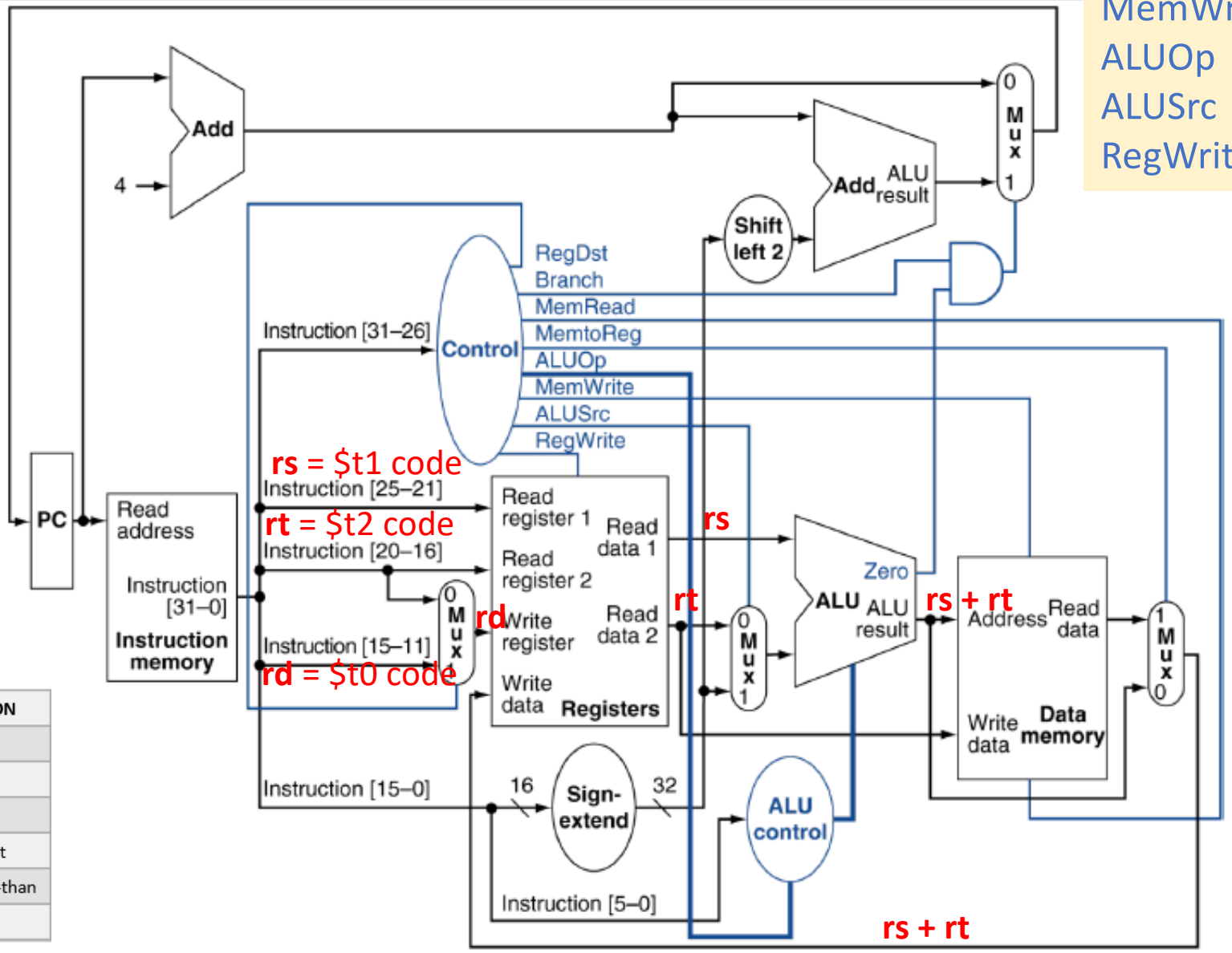
```
add $t0, $t1, $t2

addi $a0, $v0, 64

lw $t0, 4($sp)

beq $a1, $a2, blabel

jal jlabel
```



add $t0, $t1, $t2

rd = rs + rt

RegDst	1
Branch	0
Zero	X
MemRead	0
MemtoReg	0
MemWrite	0
ALUOp	0010
ALUSrc	0
RegWrite	1

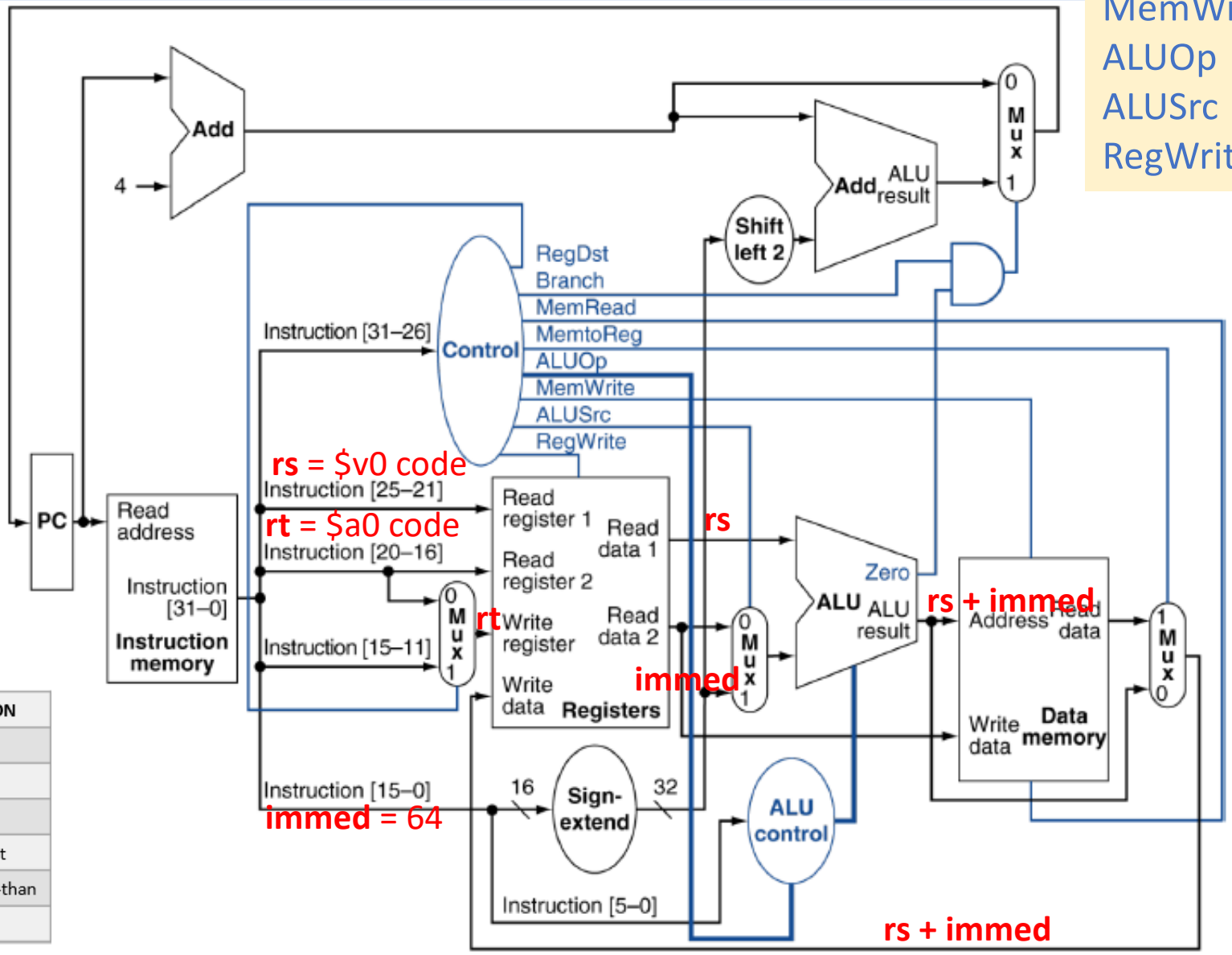


ALUOp	FUNCTION
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

addi $a0, $v0, 64

rt = rs + immed

RegDst	**0**
Branch	**0**
Zero	**X**
MemRead	**0**
MemtoReg	**0**
MemWrite	**0**
ALUOp	**0010**
ALUSrc	**1**
RegWrite	**1**

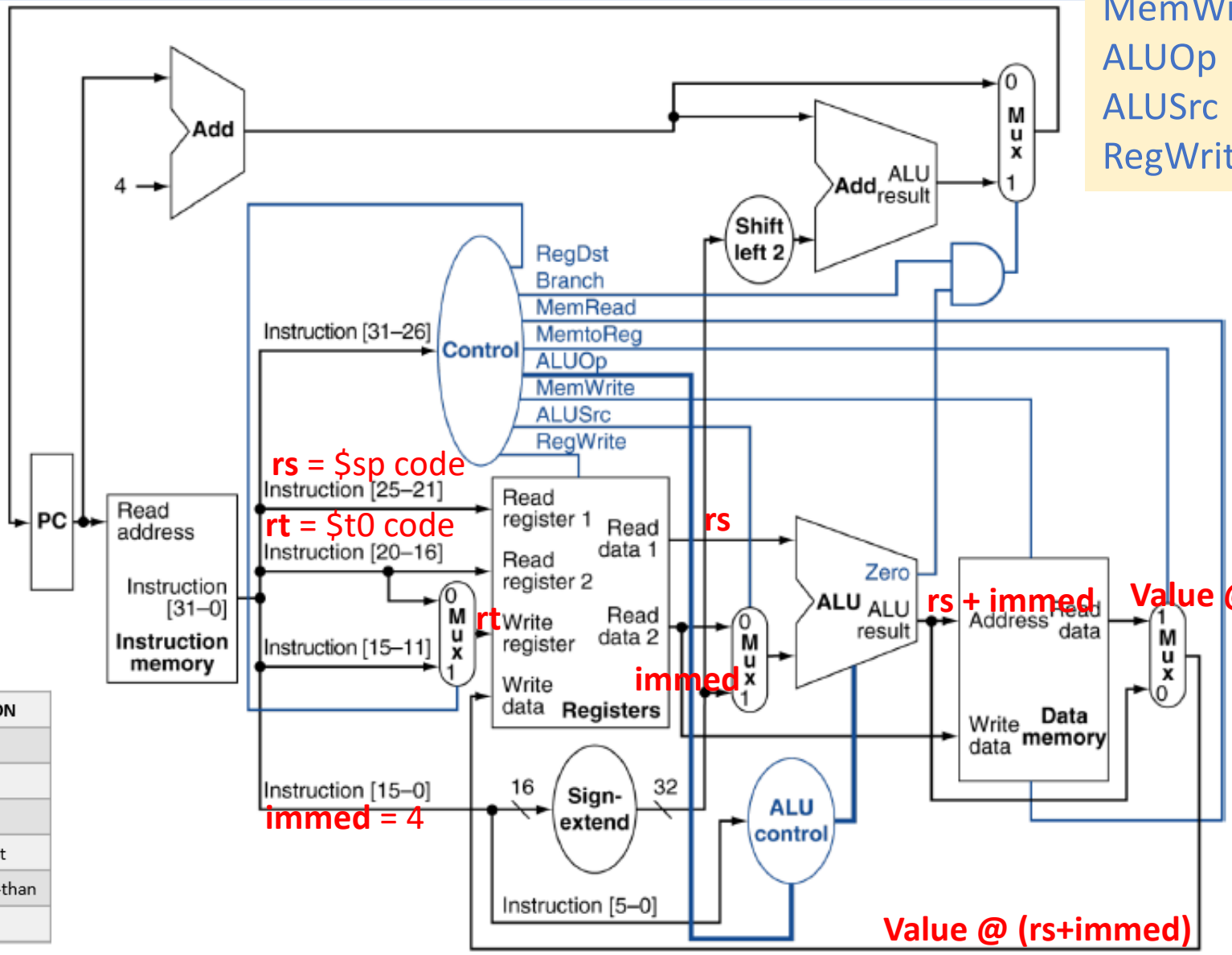


ALUOp	FUNCTION
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

lw $t0, 4($sp)

*rt = *(rs + immed)*

RegDst	0
Branch	0
Zero	X
MemRead	1
MemtoReg	1
MemWrite	0
ALUOp	0010
ALUSrc	1
RegWrite	1

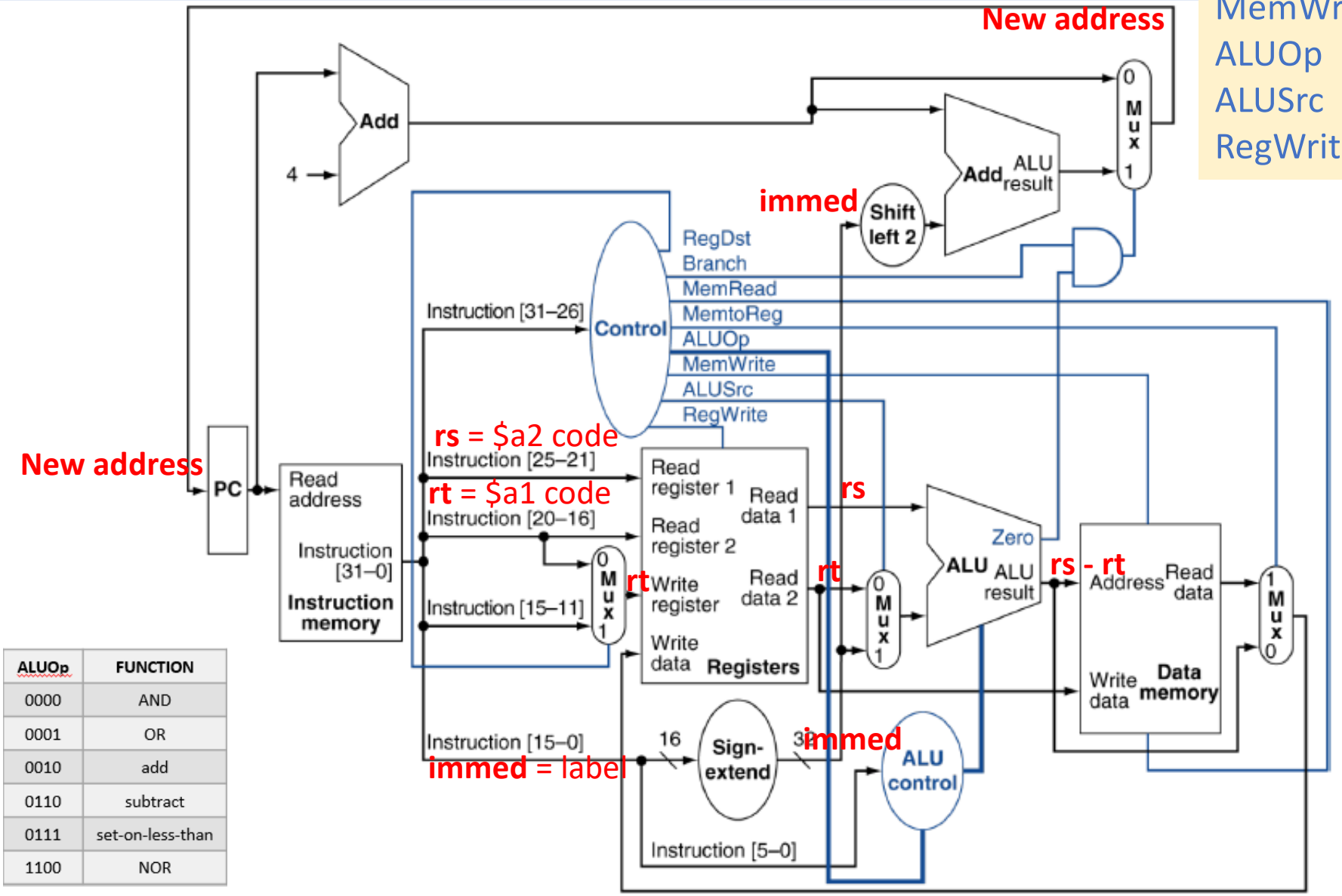


ALUOp	FUNCTION
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

beq $a1, $a2, blabel

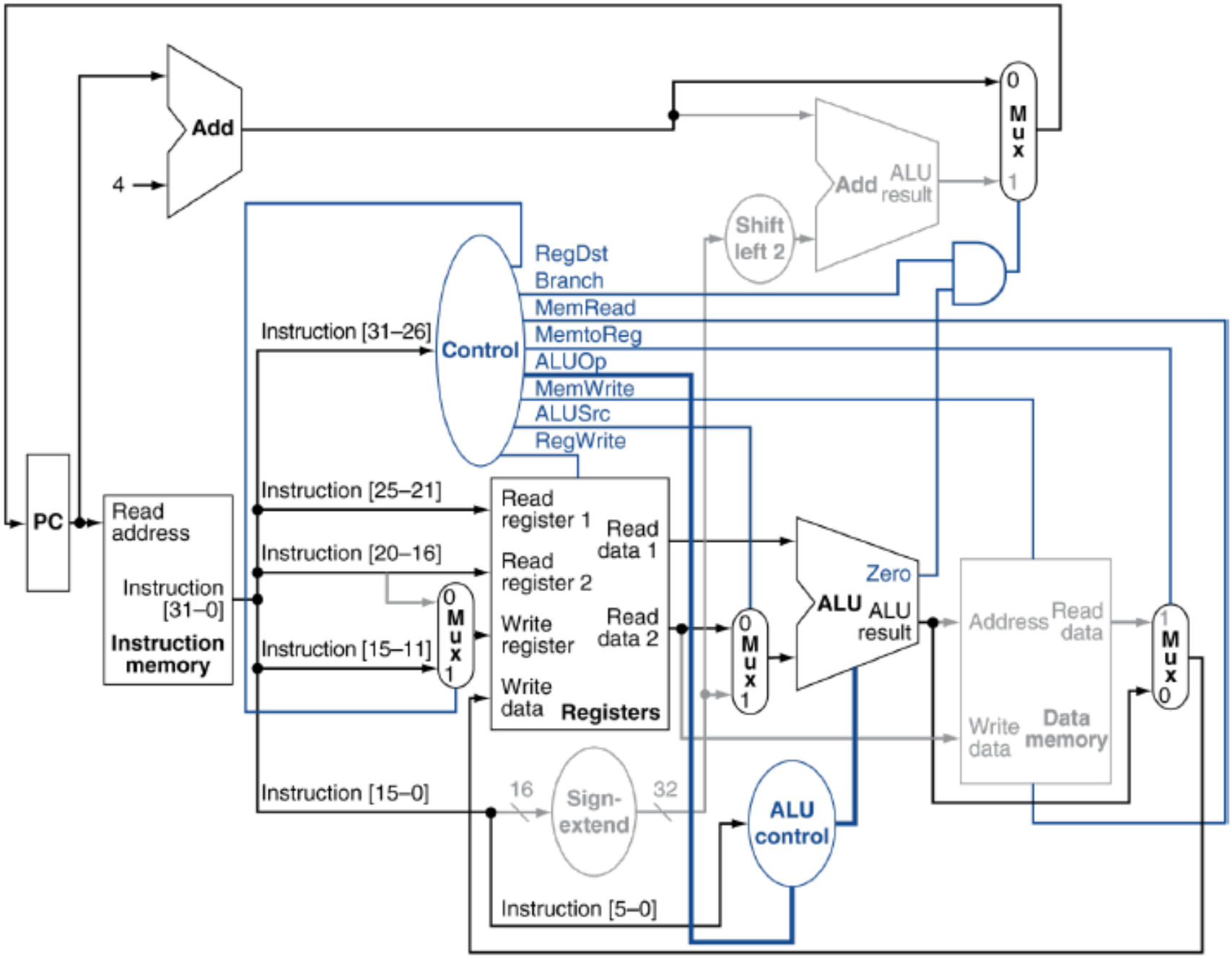
Assume in this example that a1 = a2

RegDst	1
Branch	1
Zero	1
MemRead	0
MemtoReg	0
MemWrite	0
ALUOp	0110
ALUSrc	0
RegWrite	0

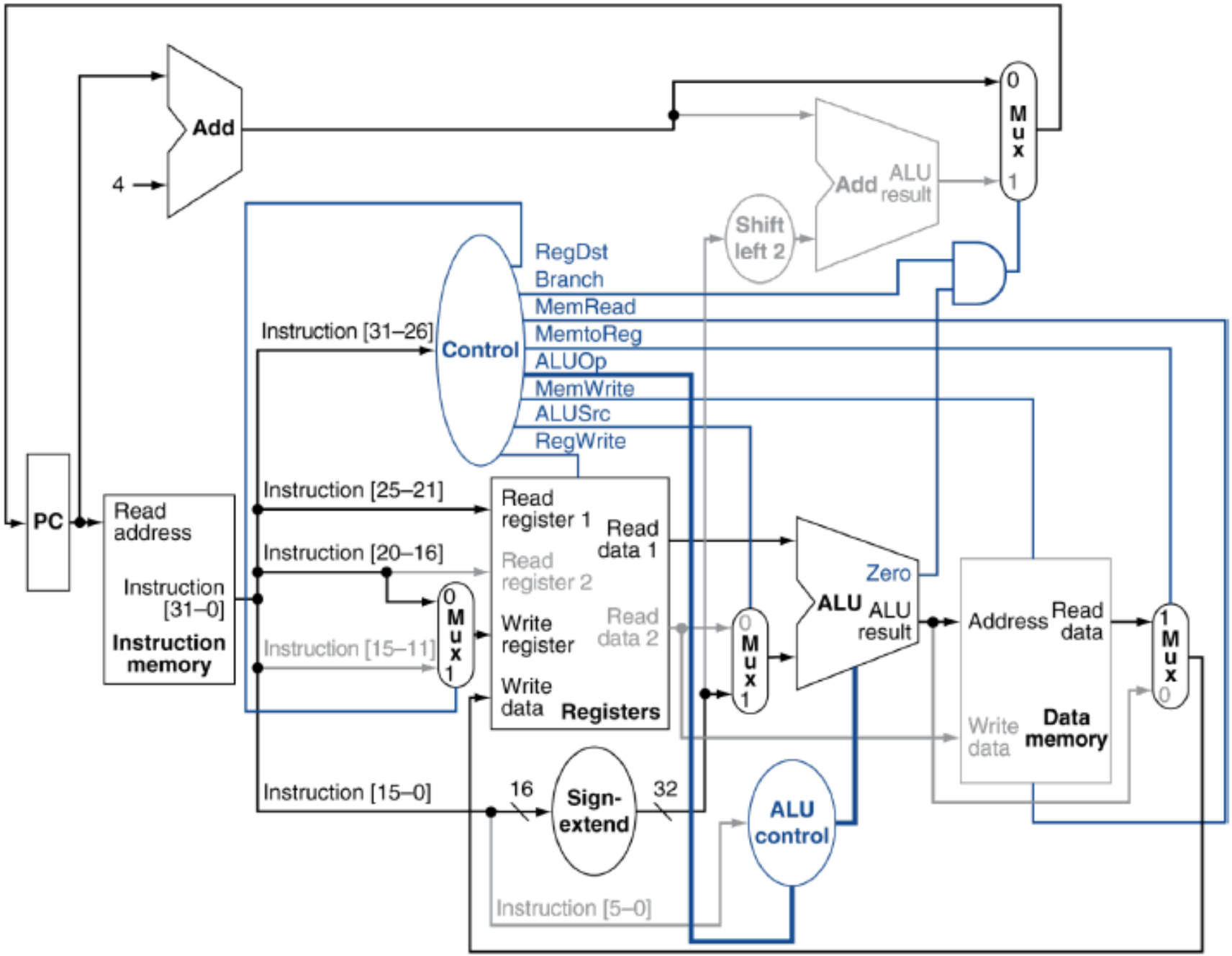


ALUOp	FUNCTION
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

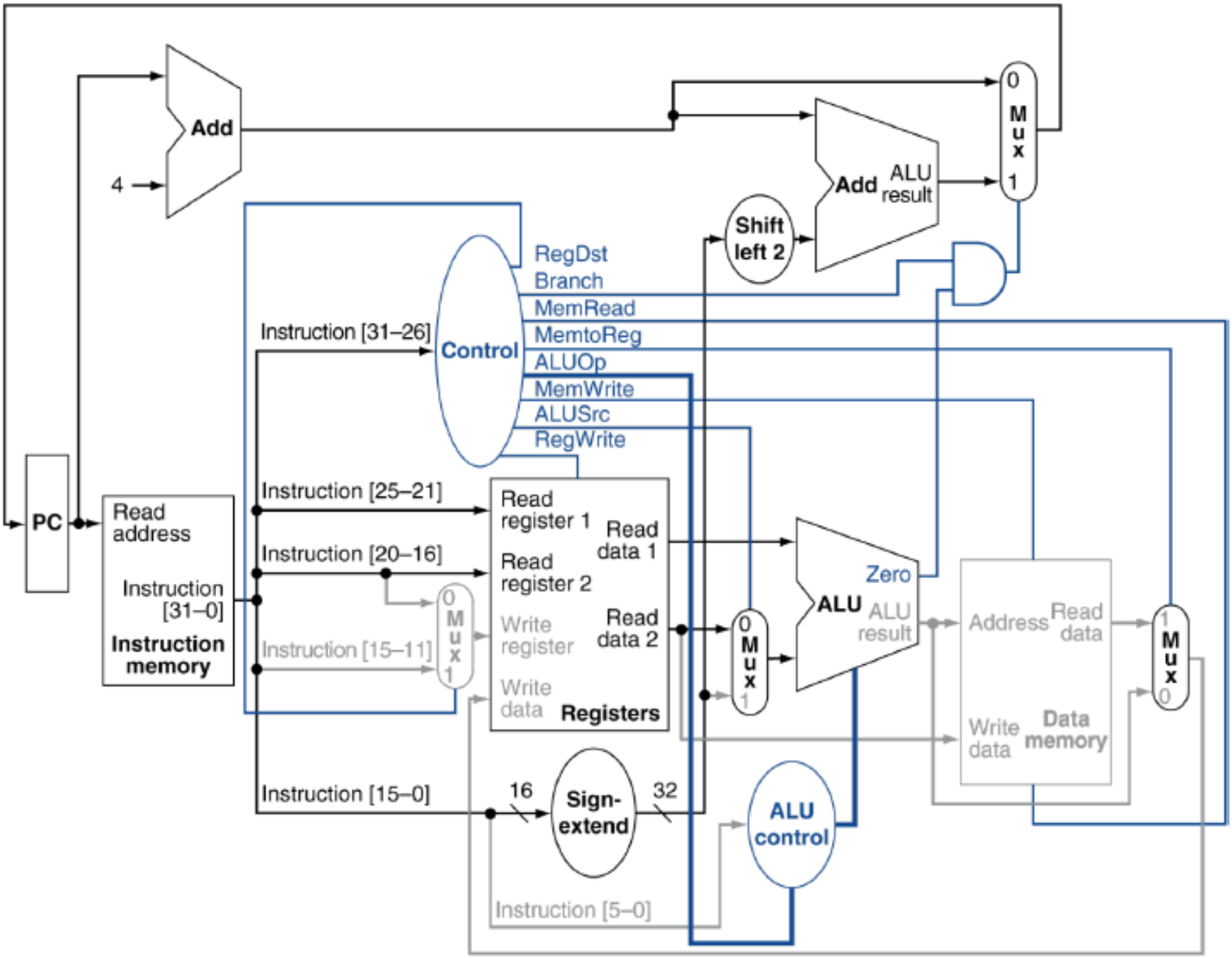
R-Type Instruction



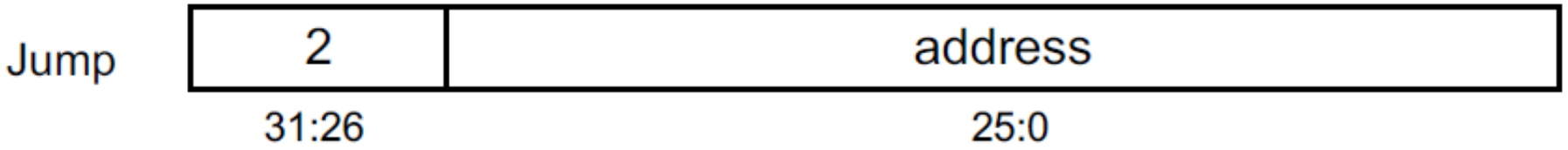
Load Instruction



Branch-on-Equal Instruction



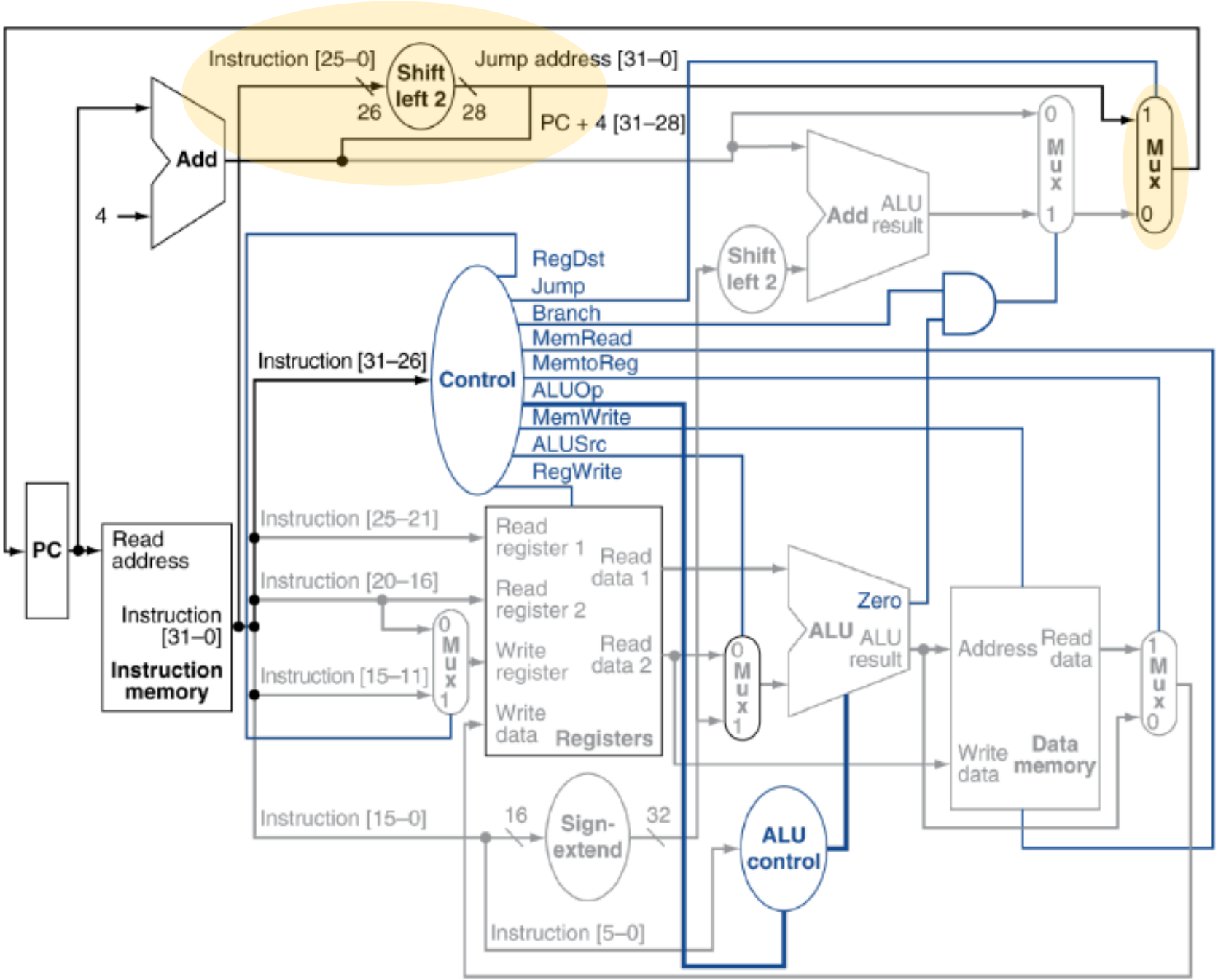
Reminder: Implementing Jumps



- Jump uses word address
  - Update PC with concatenation of 4 MS bits of old PC, 26-bit jump address, and 00 at the end

- Need an extra control signal decoded from opcode
- Need to implement a couple of other logic blocks…

Jump Instruction



Performance Issues

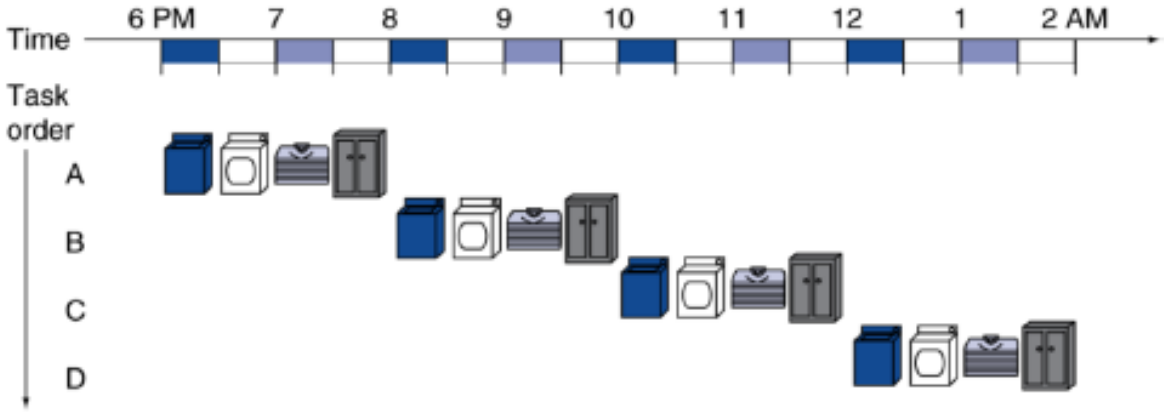
- Longest delay determines clock period
  - Critical path: load instruction
  - Goes:

Instruction memory → register file → ALU → data memory → register file

- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We can/will improve performance by pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - An example of how parallelism improves performance



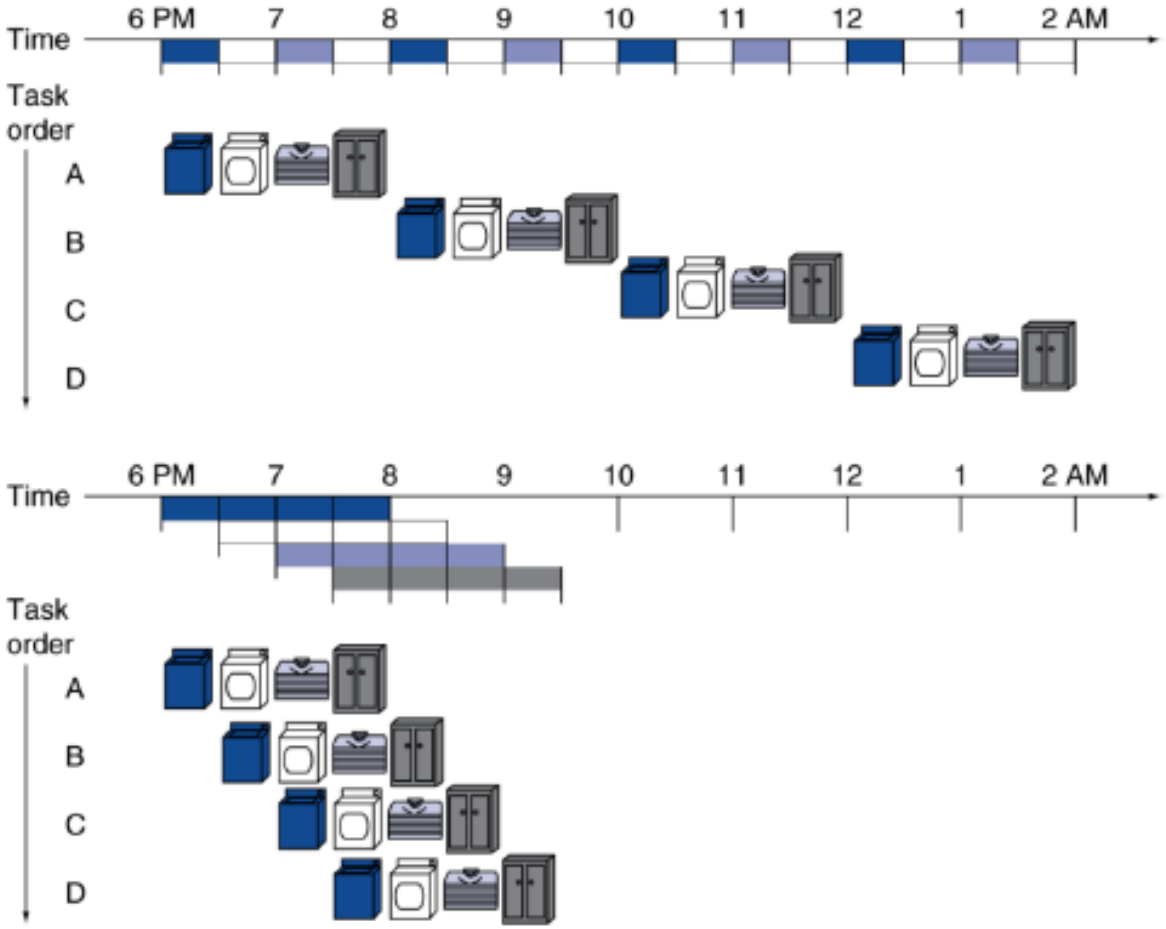
- 4 loads speeded up:
- From 8 hrs to 3.5 hrs
- Speed-up factor: 2.3

But for infinite loads:

- Speed-up factor ≈ 4 = number of stages

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - An example of how parallelism improves **throughput** performance



- 4 loads speeded up:
- From 8 hrs to 3.5 hrs
- Speed-up factor: 2.3

But for infinite loads:

- Speed-up factor ≈ 4 = number of stages

MIPS Pipeline

Five stages, one step per stage

1. **IF**: Instruction fetch from memory
2. **ID**: Instruction decode & register read
3. **EX**: Execute operation or calculate address
4. **MEM**: Access memory operand
5. **WB**: Write result back to register

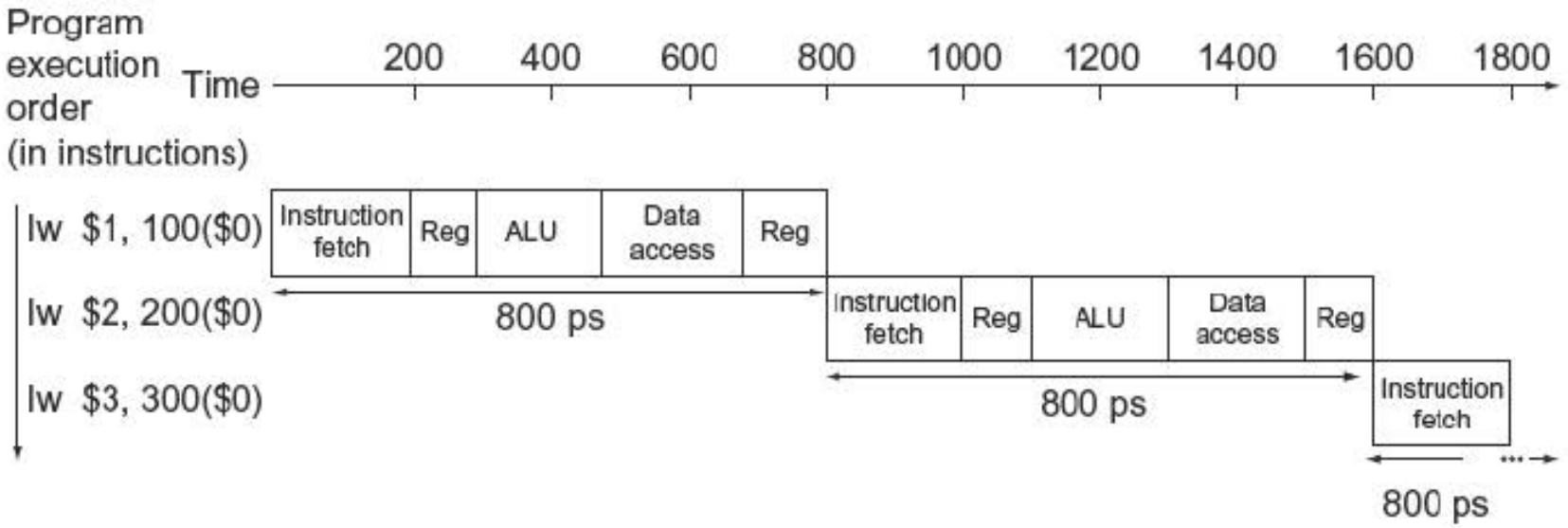
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

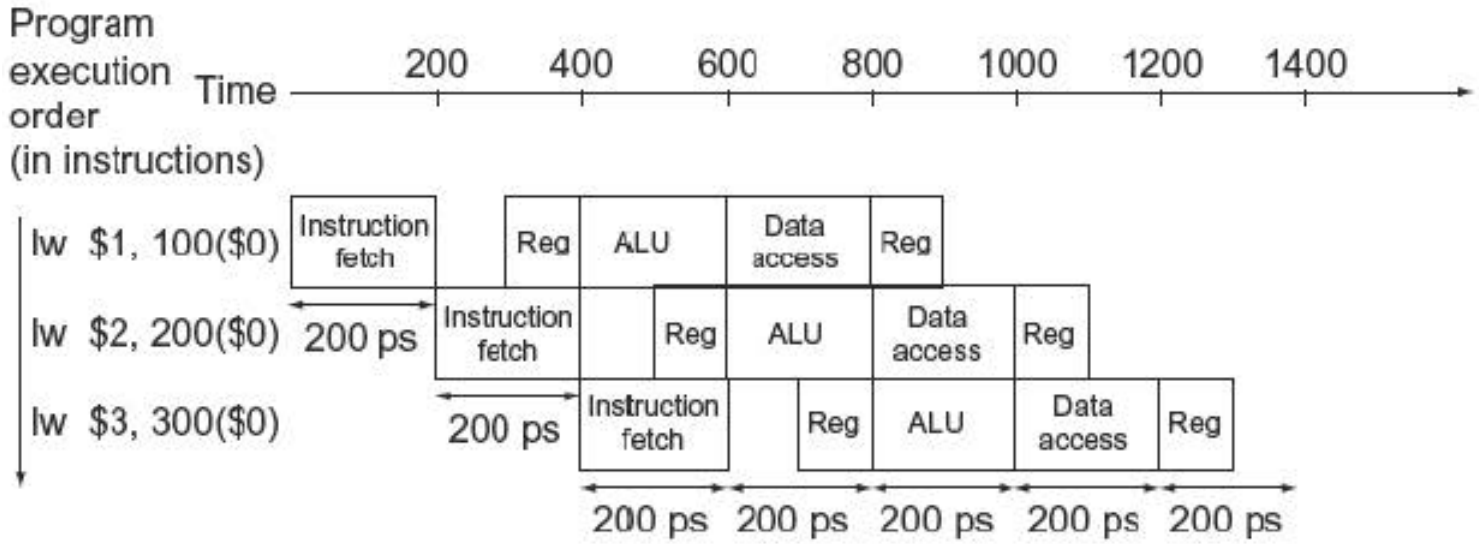
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Comparison of Per-Instruction Time

Tc = 800 ps



Tc = 200 ps



Improvement

- In the previous example, per-instruction improvement was 4x
  - 800 ps to 200 ps
- But total execution time went from 2400 ps to 1400 ps (~1.7x imp.)
- That's because we're only looking at 3 instructions…

- What if we looked at 1,000,003 instructions?
  - Total execution time = **1,000,000 x 200 ps** + 1400 ps = 200,001,400 ps
  - In non-pipelined, total time = **1,000,000 x 800 ps** + 2400 ps = 800,002,400 ps
  - Improvement = $\frac{800{,}002{,}400 \text{ ps}}{200{,}001{,}400 \text{ ps}} \approx 4.00$

About Pipeline Speedup

- If all stages are balanced, i.e. all take the same time
  - Time between instructions (pipelined)
    = Time between instructions (non-pipelined) / # of stages
- If not balanced, speedup will be less
- Speedup is due to *increased throughput*,
  but *instruction latency* does not change

MIPS vs Others' Pipelining

MIPS (and RISC-types in general) simplification advantages:

- All instructions are the same length (32 bits)
- x86 has variable length instructions (8 bits to 120 bits)

- MIPS has only 3 instruction formats (R, I, J) – rs fields all in the same place
- x86 requires extra pipelines b/c they don't

- Memory ops only appear in load/store
- x86 requires extra pipelines b/c they don't

YOUR TO-DOs for the Week

- Lab 6 due soon…


</LECTURE>