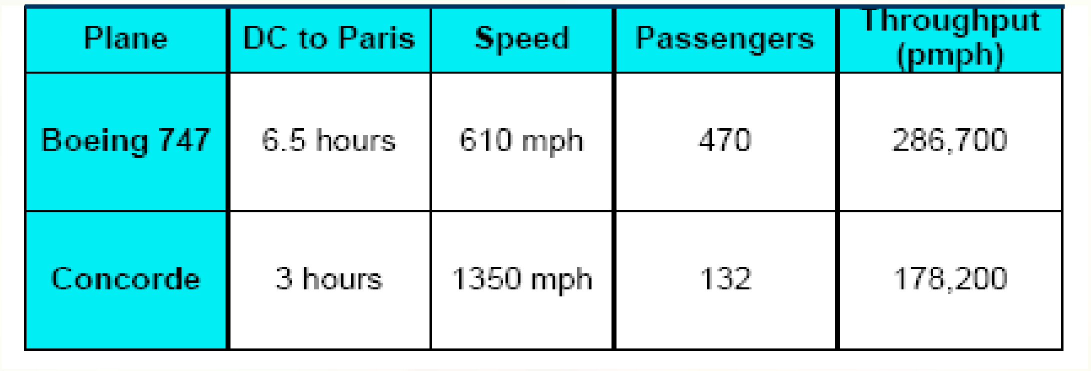
CS352H: Computer Systems Architecture

Lecture 7: Performance Measurement & MIPS Single-Cycle Implementation

September 22, 2009

Performance Is…



Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
Concorde	3 hours	1350 mph	132	178,200

- Which has higher performance?
- What is performance?
  - Time to completion (latency)? – Concorde 2.2x
  - Throughput? – 747 1.6x

Our focus

- We're concerned with performance, but there are other, sometime more important, metrics:
  - Cost
  - Power
  - Footprint
  - Weight, ,,,

Latency of What?

- DC-Paris trip:
  - Drive to airport
  - Park
  - Take shuttle
  - Check in
  - Security
  - Wait at gate
  - Board
  - Wait on plane
  - Wait on runway
  - Fly
  - Land
  - …

- Run application
  - Request resources
  - Get scheduled
  - Run
  - Request resources
  - …

Total CPU time

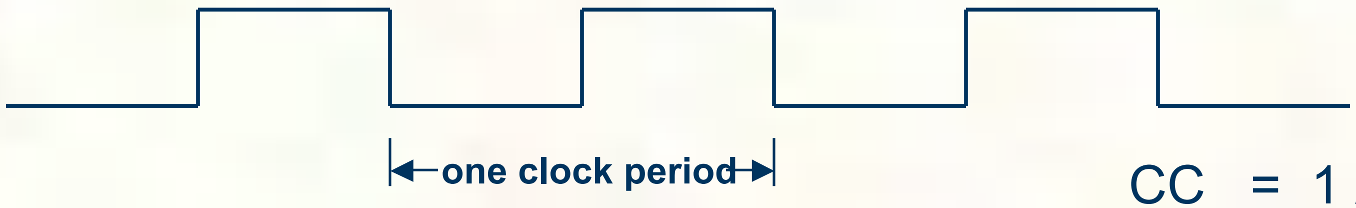
Performance Is…

- Performance is measured in terms of things-per-second
  - Bigger is better
- CPU Latency = Execution Time (ET)
- Performance(x) = $\dfrac{1}{\text{ExecutionTime}(x)}$
- X is n times faster than y means:

$$n = \frac{\text{Performance}(x)}{\text{Performance}(y)} = \frac{\text{ExecutionTime}(y)}{\text{ExecutionTime}(x)}$$

Review: Machine Clock Rate

- Clock rate (MHz, GHz) is inverse of clock cycle time



CC = 1 / CR

10 nsec clock cycle =>	100 MHz clock rate
5 nsec clock cycle =>	200 MHz clock rate
2 nsec clock cycle =>	500 MHz clock rate
1 nsec clock cycle =>	1 GHz clock rate
500 psec clock cycle =>	2 GHz clock rate
250 psec clock cycle =>	4 GHz clock rate
200 psec clock cycle =>	5 GHz clock rate

CPU Performance Factors

- $ET(x) = \#CC(x) * CC$
- $ET(x) = \dfrac{\#CC(x)}{CR}$
- $\#CC(x) = \#Instructions(x) * CPI$ (Cycles per instruction)

$$\boxed{ET = \#I * CPI * CC}$$

We can improve ET by:
- Reducing CC (increasing CR) (Technology)
- Reducing #CC:
  - Fewer instructions (Compiler)
  - Fewer cycles per instruction (Architecture)

Which is Faster?

- Two implementations of the same instruction:

Machine	CC	CPI
A	1 nsec	2.0
B	1.25 nsec	1.2

- $ET(A) = 1 \times 2.0 \times 10^{-9} = 2 \times 10^{-9}$ sec
- $ET(B) = 1 \times 1.2 \times 1.25 \times 10^{-9} = 1.5 \times 10^{-9}$ sec

$$\frac{\text{Performance(A)}}{\text{Performance(B)}} = \frac{\text{ET(B)}}{\text{ET(A)}} = \frac{1.5\text{x}10^{-9}}{2.0\text{x}10^{-9}} = 0.75$$

Groups of Instructions

- Group instructions by CPI
- Consider two assembly language implementations of the same HLL code segment

Group	CPI	#I	Product
A	1	2	2
B	2	1	2
C	3	2	6
		5	10

Group	CPI	#I	Product
A	1	4	4
B	2	1	2
C	3	1	3
		6	9

Fewer instructions may not mean higher performance

Average CPI

- To analyze program-level performance we compute the effective CPI:

$$\text{Effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i)$$

Group	CPI	Rel. Freq.	Product
Arithmetic	4	0.45	1.80
Load/Store	6	0.35	2.10
Test	3	0.05	0.15
Branch	3	0.15	0.45
Sum		1.00	4.50

Note that to be meaningful, the sum of the relative frequencies must be 1.0

Example 1

- Program parameters:
  - #I: $7.5 * 10^9$
  - Clock rate: 600MHz
  - CPI data:

Group	Rel. Freq.	CPI	Product
Load	0.28	5	1.40
Store	0.15	4	0.60
Arithmetic	0.44	4	1.76
Branch	0.09	3	0.27
Other	0.04	4	0.16
Sum	1.00		4.19

$$ET(Pgm) = \frac{\#I * CPI}{Rate} = \frac{7.5 * 10^9 * 4.19}{600 * 10^6} = 52.375 \text{ sec}$$

- Slowest CPU to execute in 35sec?

$$35 = \frac{7.5 * 10^9 * 4.19}{x}$$

$$\boxed{X = 898\text{MHz}}$$

Example 1 (cont')

- Optimizer reduces instruction counts as follows:
- Effect on performance?

Group	Reduction
Load	12%
Store	4%
Arithmetic	20%
Branch	5%
Other	None

Group	Old Rel. Freq.	Reduced to:	Product	New Rel. Freq.
Load	0.28	0.88	0.2464	0.2839
Store	0.15	0.96	0.1440	0.1659
Arithmetic	0.44	0.80	0.3520	0.4056
Branch	0.09	0.95	0.0855	0.0985
Other	0.04	1.00	0.0400	0.0461
Sum	1.00		0.8679	1.0000

$\#I = 0.8679 * 7.5*10^9 = 6.50925*10^9$

Example 1 (cont')

Group	New Rel. Freq.	CPI	Product
Load	0.2839	5	1.4190
Store	0.1659	4	0.6636
Arithmetic	0.4056	4	1.6224
Branch	0.0985	3	0.2955
Other	0.0461	4	0.1844
Sum	1.00		4.1849

$$ET(Pgm) = \frac{\#I * CPI}{Rate} = \frac{6.50925 * 10^9 * 4.1849}{600 * 10^6} = 45.401 \text{ sec}$$

So, how much faster is the optimized code?

$$\frac{Perf(optimized)}{Perf(unoptimized)} = \frac{ET(unoptimized)}{ET(optimized)} = \frac{52.375}{45.401} = 1.15$$

Speeding Up Execution Time

- Execution Time is a function of three things:
  - # of instructions
  - Average CPI
  - Clock rate
- We can improve it by:
  - Choosing the “best” instruction sequence (compiler)
  - Reducing CPI (architecture)
  - Increasing clock rate (technology)
- But changing one can adversely affect the others!

Speeding Up Execution Time

- Compiler technology is quite good at generating sequences with the fewest instructions
  - Recall that this may not mean the fewest clock cycles
- Adoption of RISC architectures has led to significant reductions in average CPI
  - By using simple instructions that lend themselves to fast implementation
  - At a cost of more instructions
- Clock rates have risen between 2 and 3 orders of magnitude:
  - MIPS R2000 ca. 1985: 8MHz
  - MIPS R16000 ca. 2002: 1GHz
- Rate of change is slowing down!

Example 2

Op	Freq	CPI_i	Freq x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			Σ = 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

  CPU time new = 1.6 x #I x CC so 2.2/1.6 means 37.5% faster

- How does this compare with using branch prediction to shave a cycle off the branch time?

  CPU time new = 2.0 x #I x CC so 2.2/2.0 means 10% faster

- What if two ALU instructions could be executed at once?

  CPU time new = 1.95 x #I x CC so 2.2/1.95 means 12.8% faster

Which Programs Should We Analyze?

- Kernels: Livermore loops, LINPACK
  - Small programs that are easy to implement across architectures
  - Capture heart of a class of computations
- Synthetic programs: Whetstone, Dhrystone
  - Don't perform any meaningful computation
  - But represent a model of what goes on in real computations
- Benchmark programs: SPEC, TPC
  - A collection of programs that represent what users do
- Actual applications
  - Meaningful to you
  - May not port to all systems
  - May require large data sets

Sidebar: A (Real) Anecdote

- Without optimizer: 5.2 seconds
- With optimizer: 0.1 seconds!!!
- Optimizer determined that *sum* was never used and therefore didn't generate any code to compute it!

```
void main()
{
        int i;
        double x, sum;

        sum = 0.0;
        for (i=0; i<10000; i++) {
                x = (double)i;
                sum += sqrt(x);
        }
}
```

Printing *sum* resulted in optimized version running twice as fast as the unoptimized code

Moral: Understand what is going on

SPEC

- An industry consortium
  - System Performance Evaluation Cooperative
- Series of benchmarks consisting of real programs:
  - Computation-intensive
  - Graphics
  - Web servers
  - Java client/server
  - SIP
  - Virtualization
  - …
- Evolve over time:
  - CPU92 ➝ CPU95 ➝ CPU2000 ➝ CPU2006

SPEC CPU2006

- SPEC CPU2006
  - CINT2006: 12 integer-only programs (C and C++)
  - CFP2006: 17 floating point programs (FORTRAN, C and C++)
- Detailed benchmark specification for reproducibility:
  - Detailed hardware characteristics
    - # of CPUS and their clock rates
    - Memory size, …
  - Detailed software characteristics:
    - Operating system version
    - Compiler used
    - Flag setting during compilation, …
- Results are relative to a baseline: Sun Ultra Enterprise 2
  - Ratio of measured wall clock time to baseline
  - A larger number is better
- Rating is a geometric mean of the individual results
- Great, but suffers from the “No child left behind” syndrome
  - Relevance
  - Manipulation

SPEC CPU2006 Components

CINT2006

perlbench	Several Perl apps
bzip2	Data compression
gcc	C compiler
mcf	Combinatorial optimization
gobmk	Go program
hmmer	Protein sequence analysis
sjeng	Chess program
libquantum	Simulates a quantum computer
h264ref	Video compression
omnetpp	Campus-wide ethernet simulation
astar	Path finding algorithms
xalancbmk	XML processor

CFP2006

bwaves	Fluid dynamics
gamess	Quantum chemistry
milc	Quantum chromodynamics
zeusmp	Computational fluid dynamics
gromacs	Molecular dynamics
cactusADM	General relativity
leslie3D	Computational fluid dynamics
namd	Molecular dynamics
dealII	Finite element analysis
soplex	Simplex algorithm
povray	Image ray tracing
calculix	Structural mechanics
GemsFDTD	Computational electromagnetics
tonto	Quantum chemistry
lbm	Fluid dynamics
wrf	Weather modeling
sphinx3	Speech recognition

TPC: Transaction Processing Benchmarks

- Define an application scenario
  - Involves end-users
  - Remote access over a network
  - Databases
- Considers throughput, latency, and price
- Originally developed for ATM-like transactions
- Now focused on order-entry application developed at MCC in mid-80's
- Attempt at reality and completeness
  - But at the price of tremendous complexity

Performance Speed Up

$$\text{SpeedUp} = \frac{\text{ET Before Change}}{\text{ET After Change}}$$

Speedup depends on:

Goodness of enhancement (s)

Fraction of time it's used (p)

ET After Change =

$$\text{ET Before Change} * \left[(1 - p) + \frac{p}{s} \right]$$

Amdahl's Law

- Gene Amdahl: IBM S/360 Chief Architect
- Speedup = $$\frac{1}{\left[(1-p)+\frac{p}{s}\right]}$$
- Speedup bounded by: $$\frac{1}{\text{Fraction of time not enhanced}}$$
- Duh!

Example 3

- Can double performance of floating point instructions
  - Cut latency by a factor of 2
- Floating point operations represent 10% of workload

$$\text{ETAfter} = \text{ETBefore} * \left[\, 0.9 + \frac{0.1}{2} \,\right] = 0.95 * \text{ETBefore}$$

$$\text{SpeedUp} = \frac{1}{0.95} = 1.053$$

Example 4

- Application takes 100sec to run
- Multiplication represents 80% of the work
- How much faster would multiplication have to be in otder to get performance to improve by a factor of 4?

$$25 = 100 * \left[0.2 + \frac{0.8}{x} \right]$$

- How about a factor of 6?

Amdahl's Law

- Make the common case fast!
- Performance improvement depends on:
  - Goodness of the enhancement

  <u>And</u>

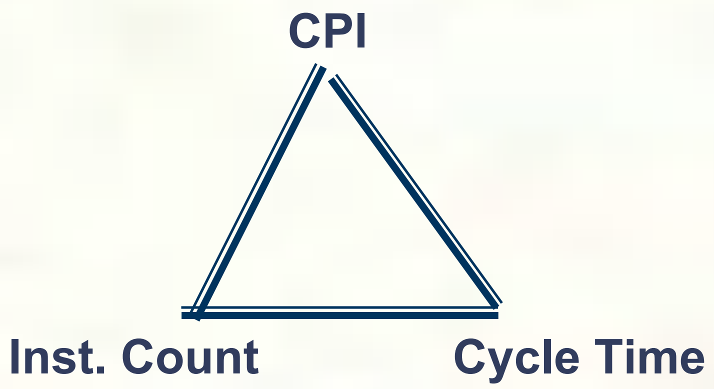
  - Frequency of use
- Examples
  - All instructions require instruction fetch, only a fraction require data
    - Optimize instruction access first
  - Programs exhibit data locality; small memories are faster
    - Storage hierarchy: most frequent access to small, fast, local memory

Summary: Evaluating ISAs

- Design-time metrics:
  - Can it be implemented, in how long, at what cost?
  - Can it be programmed? Ease of compilation?
- Static Metrics:
  - How many bytes does the program occupy in memory?
- Dynamic Metrics:
  - How many instructions are executed? How many bytes does the processor fetch to execute the program?
  - How many clocks are required per instruction?
  - How fast can the clock be made?

Best Metric: Time to execute the program!

depends on the instructions set, the processor organization, and compilation techniques.

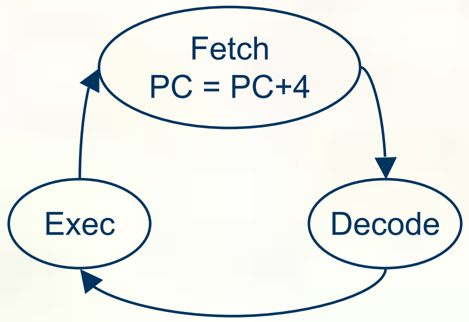


Beauty is in the Eye of the Beholder

- The right metric depends on the application:
  - Desktop
  - Game console
  - Microwave oven microcontroller
  - Web server
- The right metric depends on the perspective:
  - CPU designer
  - System architect
  - Customer
- Opportunity for manipulation galore!

The Processor: Datapath & Control

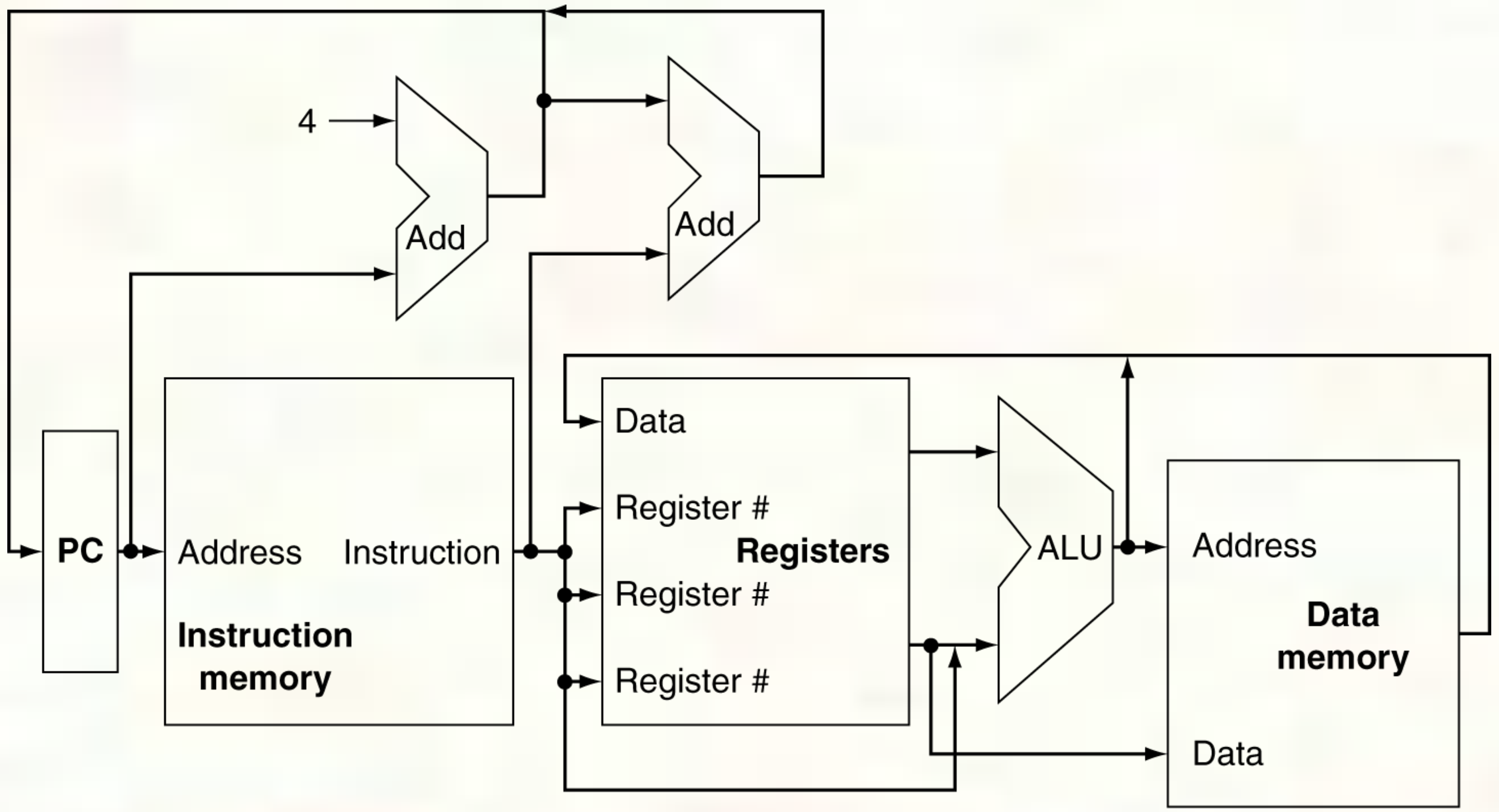
- Our implementation of the MIPS is simplified
  - memory-reference instructions: **lw, sw**
  - arithmetic-logical instructions: **add, sub, and, or, slt**
  - control flow instructions: **beq, j**
- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction
- Later - more realistic pipelined version
- All instructions (except **j**) use the ALU after reading the registers



Instruction Execution

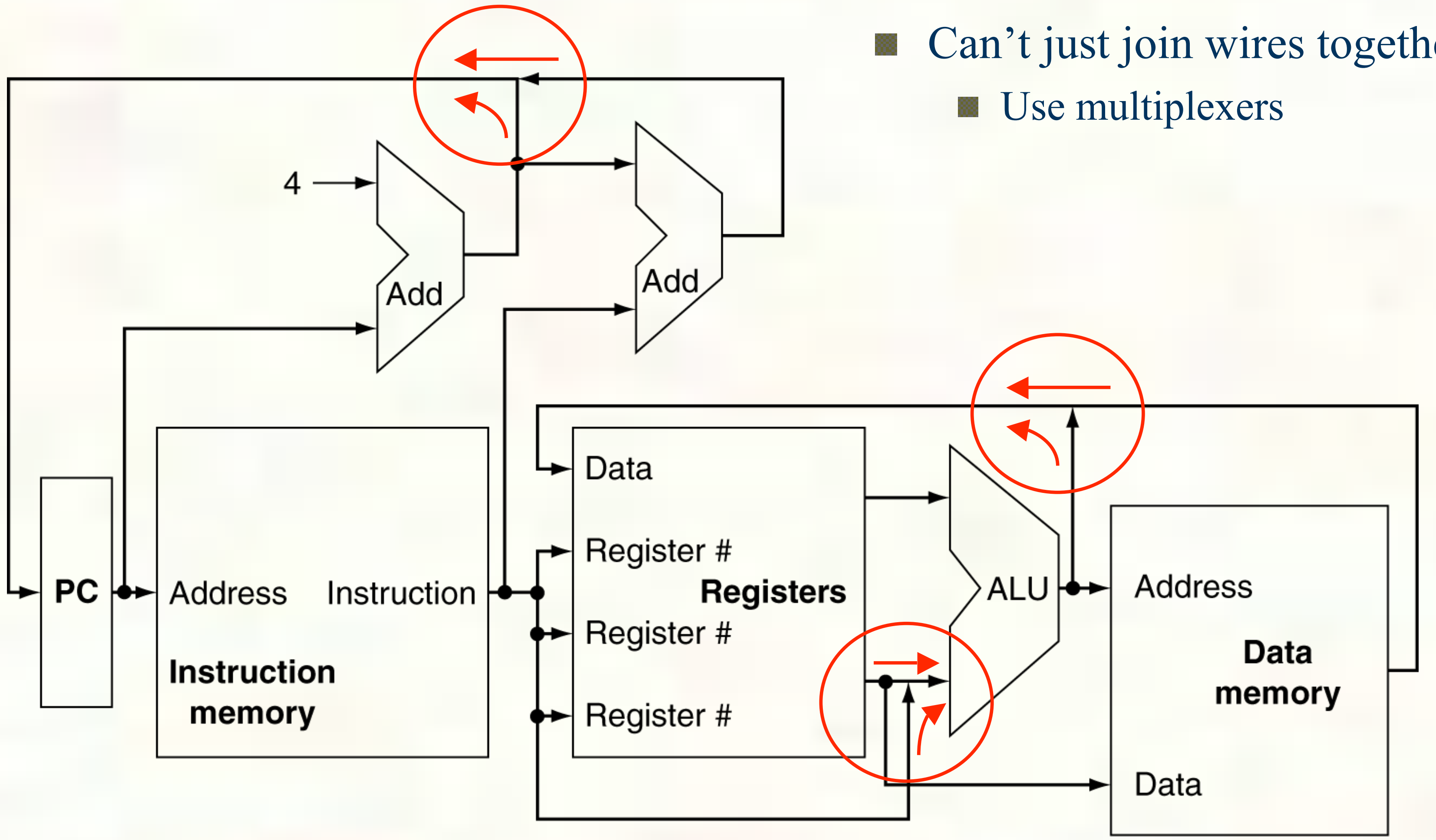
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address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

CPU Overview

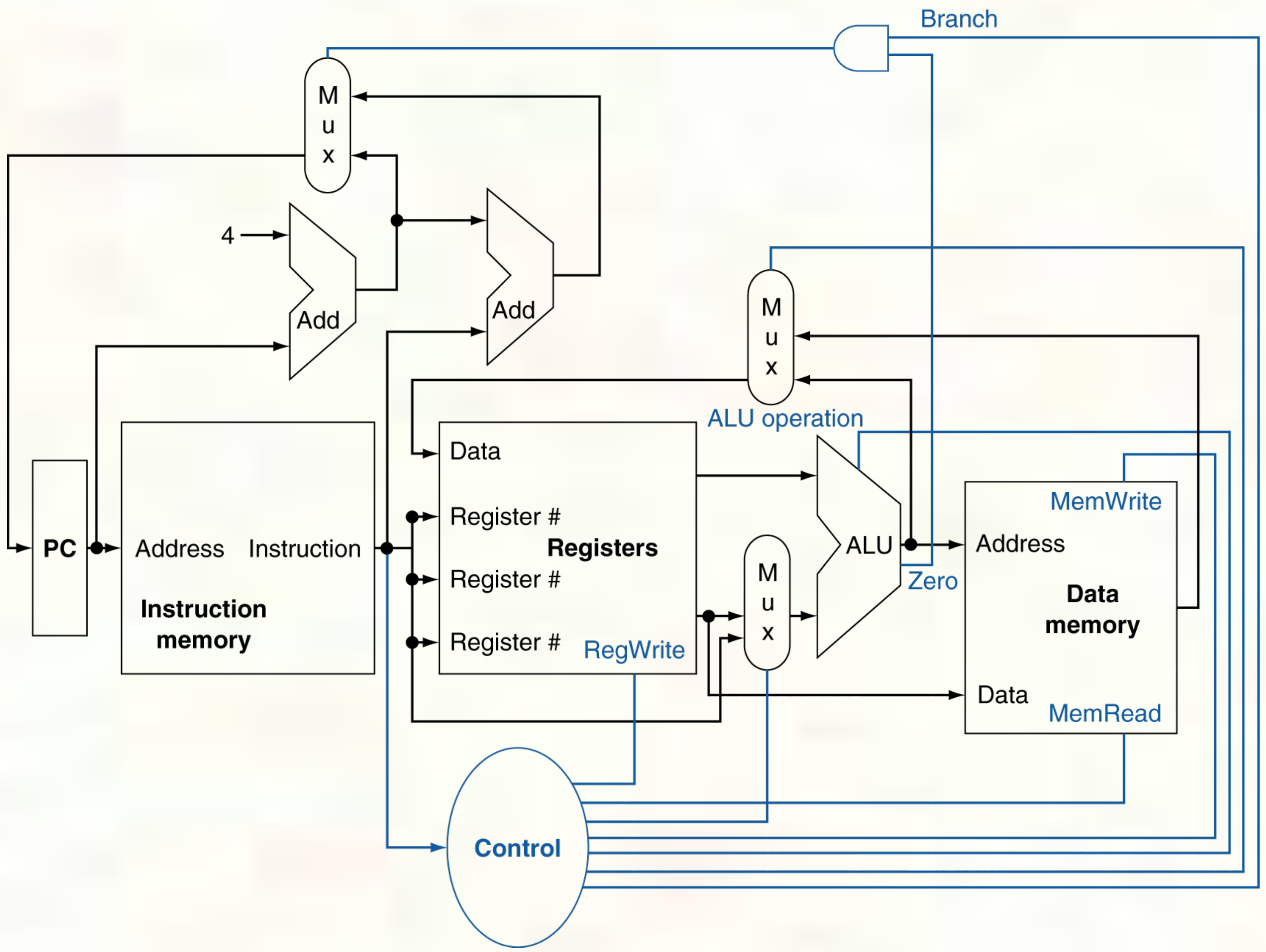


Multiplexers

- Can’t just join wires together
  - Use multiplexers



Control



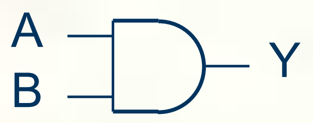
Branch
M u x
4
Add
Add
M u x
ALU operation
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control

Logic Design Basics

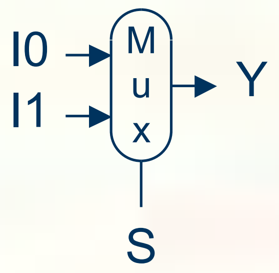
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

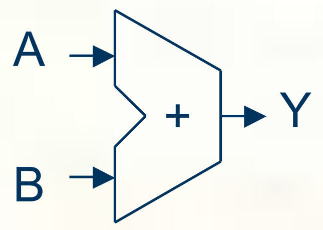
- AND-gate
  - Y = A & B



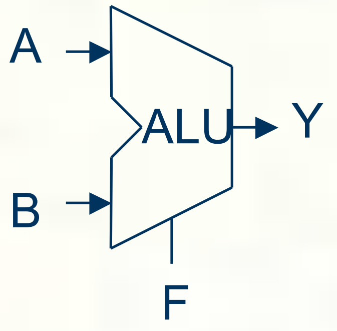
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

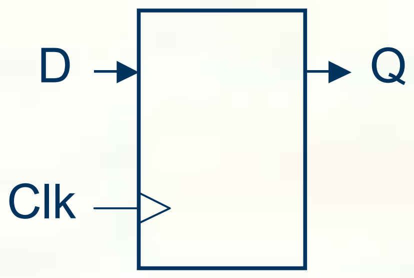


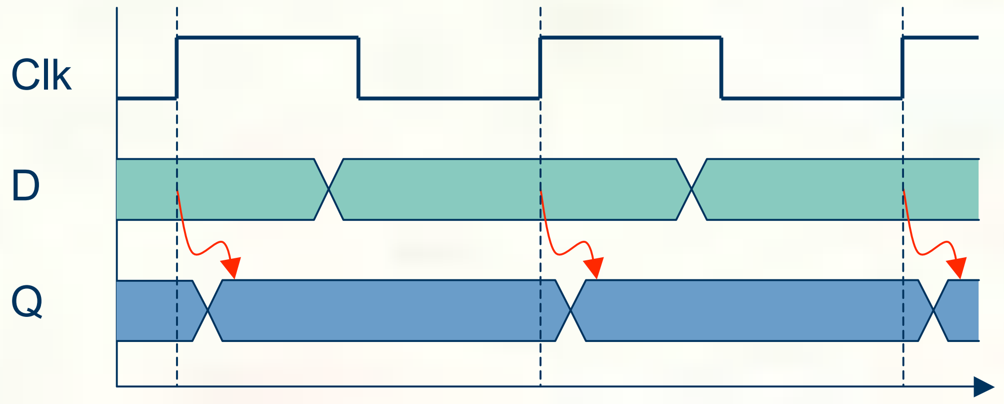
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

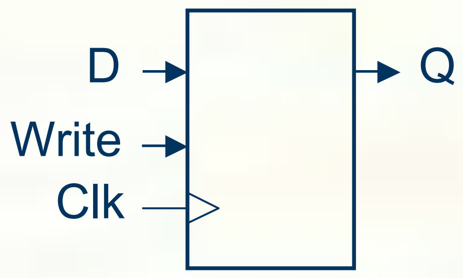
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

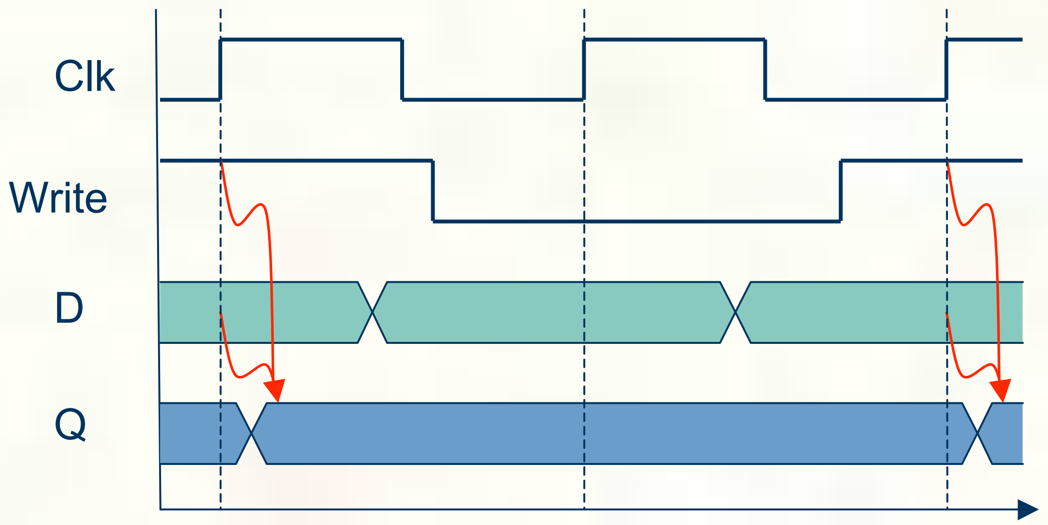




Sequential Elements

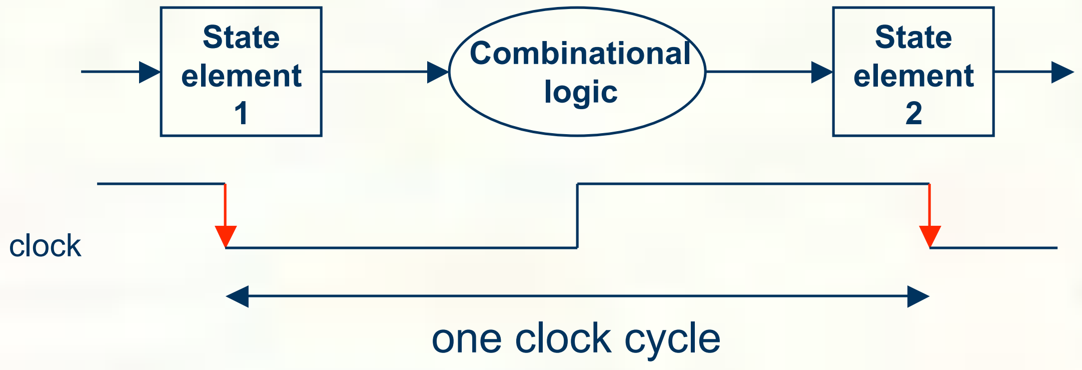
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

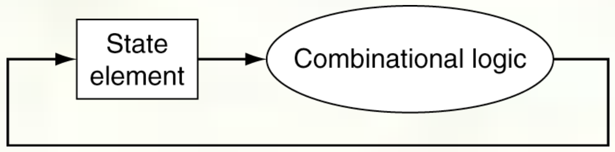




Clocking Methodologies

- The clocking methodology defines when signals can be read and when they are written
  - An edge-triggered methodology
  - Longest delay determines clock period
- Typical execution
  - read contents of state elements
  - send values through combinational logic
  - write results to one or more state elements





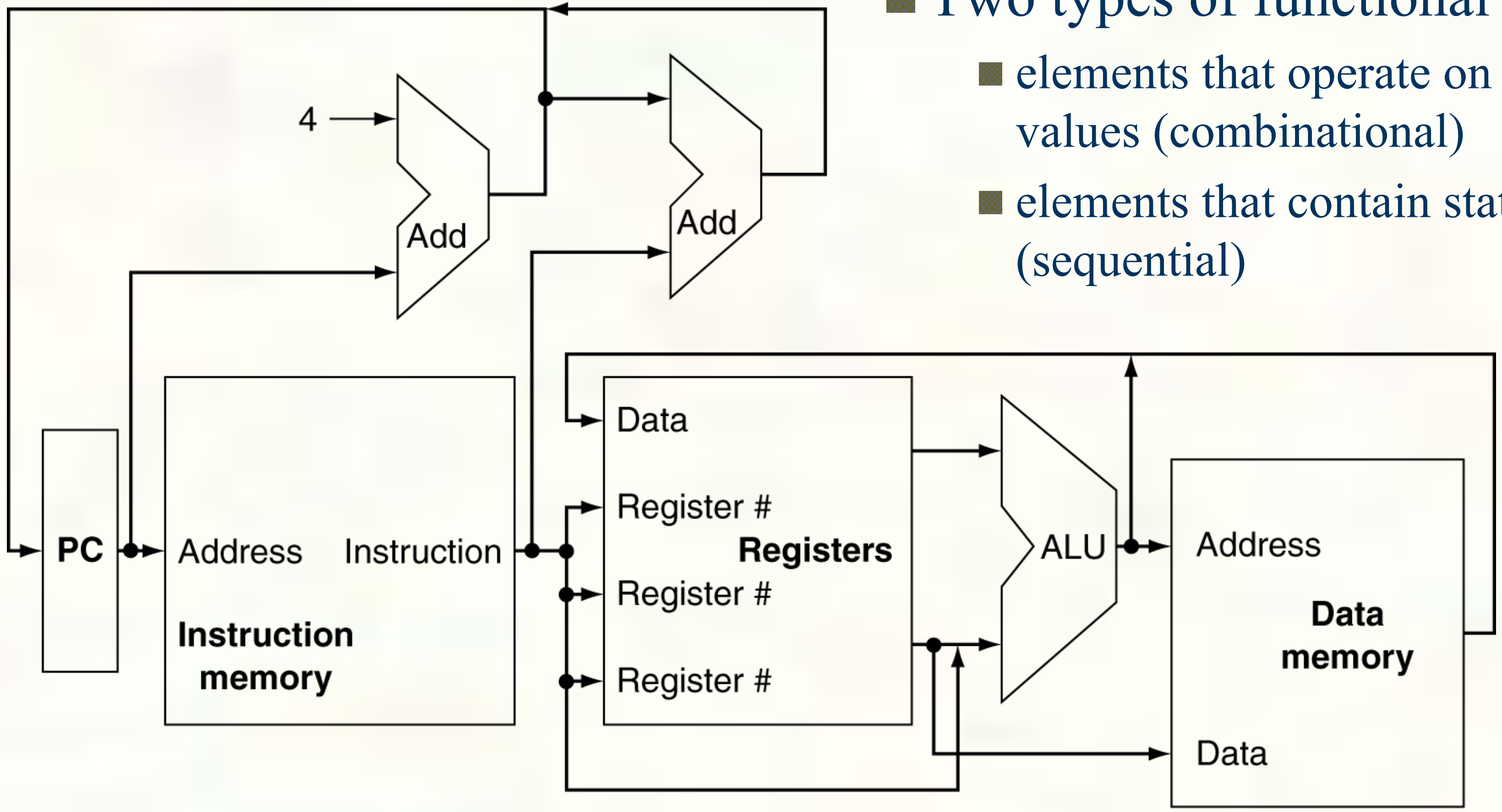
Assumes state elements are written on every clock cycle; if not, need explicit write control signal

write occurs only when **both** the write control is asserted and the clock edge occurs

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Abstract Implementation View



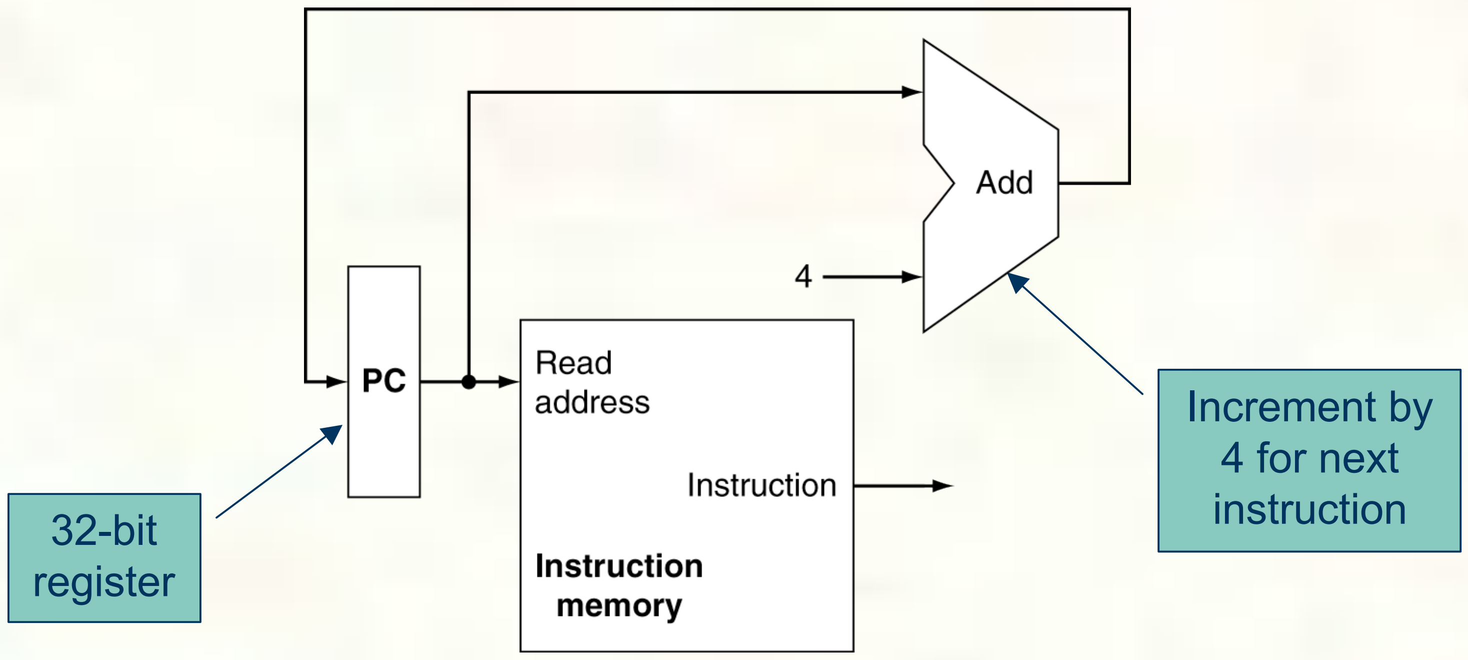
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

Single cycle operation

Split memory model - one memory for instructions and one for data

Instruction Fetch

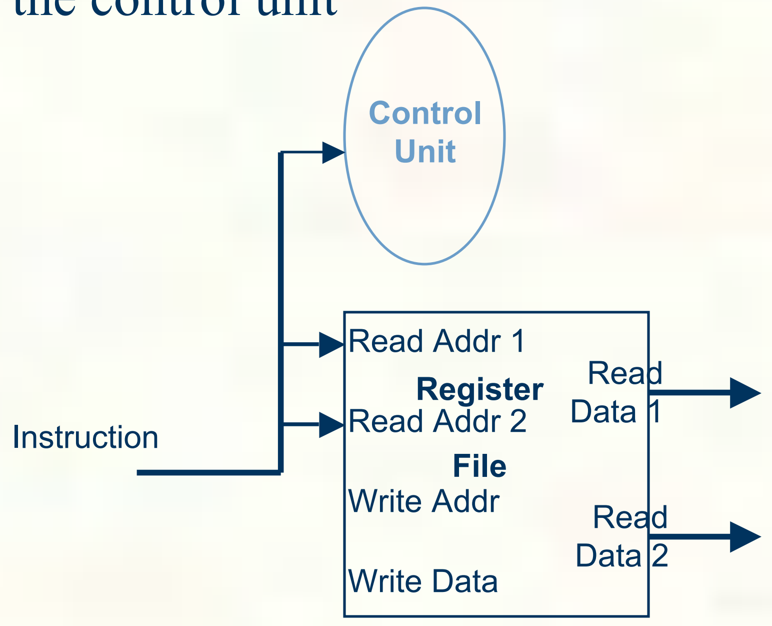
- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC to hold the address of the next instruction



PC is updated every cycle, so it does not need an explicit write control signal
Instruction Memory is read every cycle, so it doesn't need an explicit read control signal

Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction’s opcode and function field bits to the control unit



reading two values from the Register File

Register File addresses are contained in the instruction

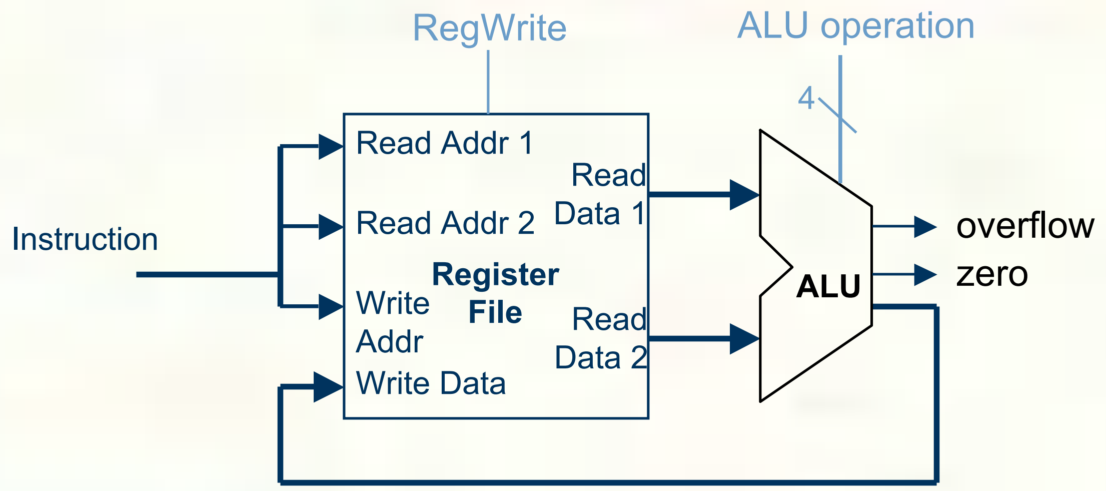
R Format Instructions

- R format operations (**add, sub, slt, and, or**)

31 25 20 15 10 5 0

R-type:	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**

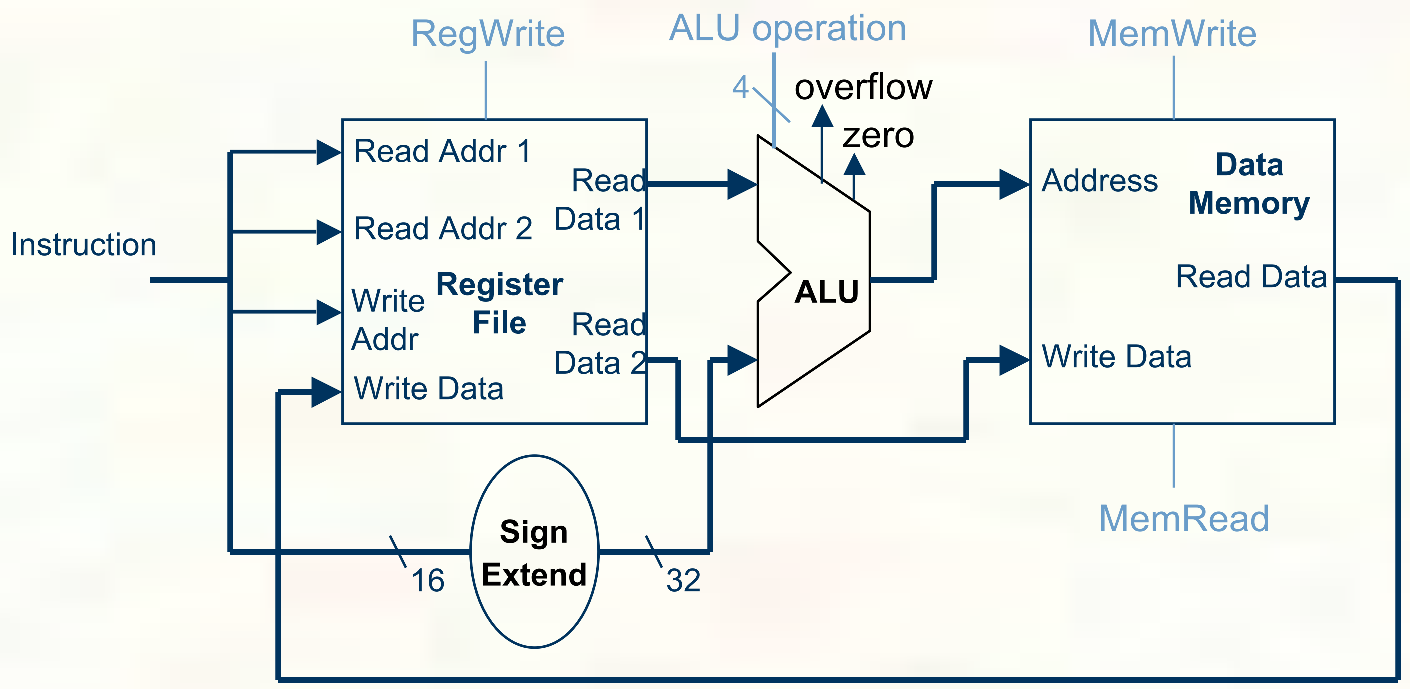
  - perform the (**op** and **funct**) operation on values in **rs** and **rt**
  - store the result back into the Register File (into location **rd**)



The Register File is not written every cycle (e.g. **sw**), so we need an explicit write control signal for the Register File

Load and Store Instructions

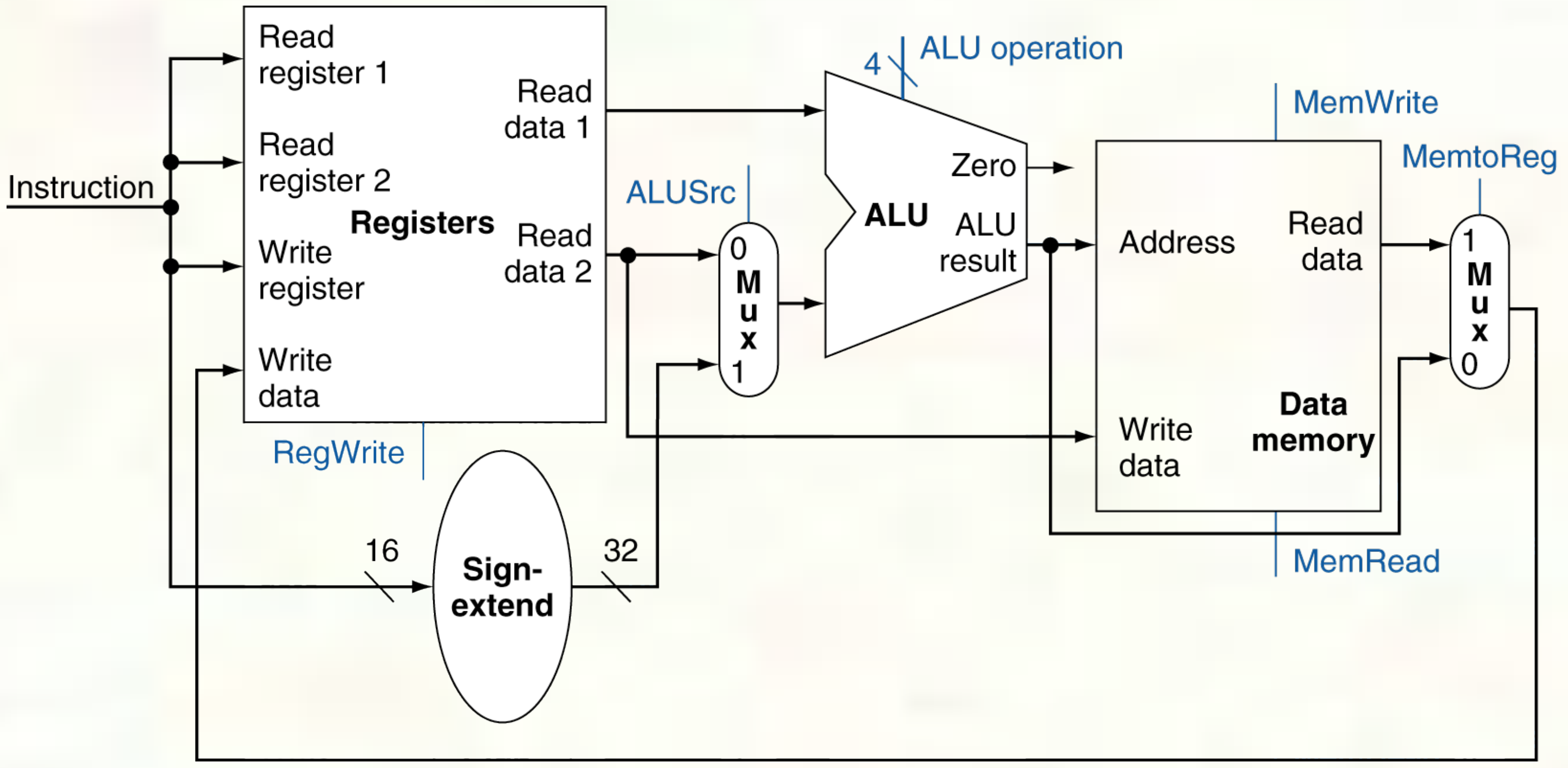
- Load and store operations involve:
  - Read register operands
  - Compute memory address by adding the base register to the 16-bit signed-extended offset field in the instruction
  - **Store** value (read from the Register File) written to the Data Memory
  - **Load** value, read from the Data Memory, written to the Register File



Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath

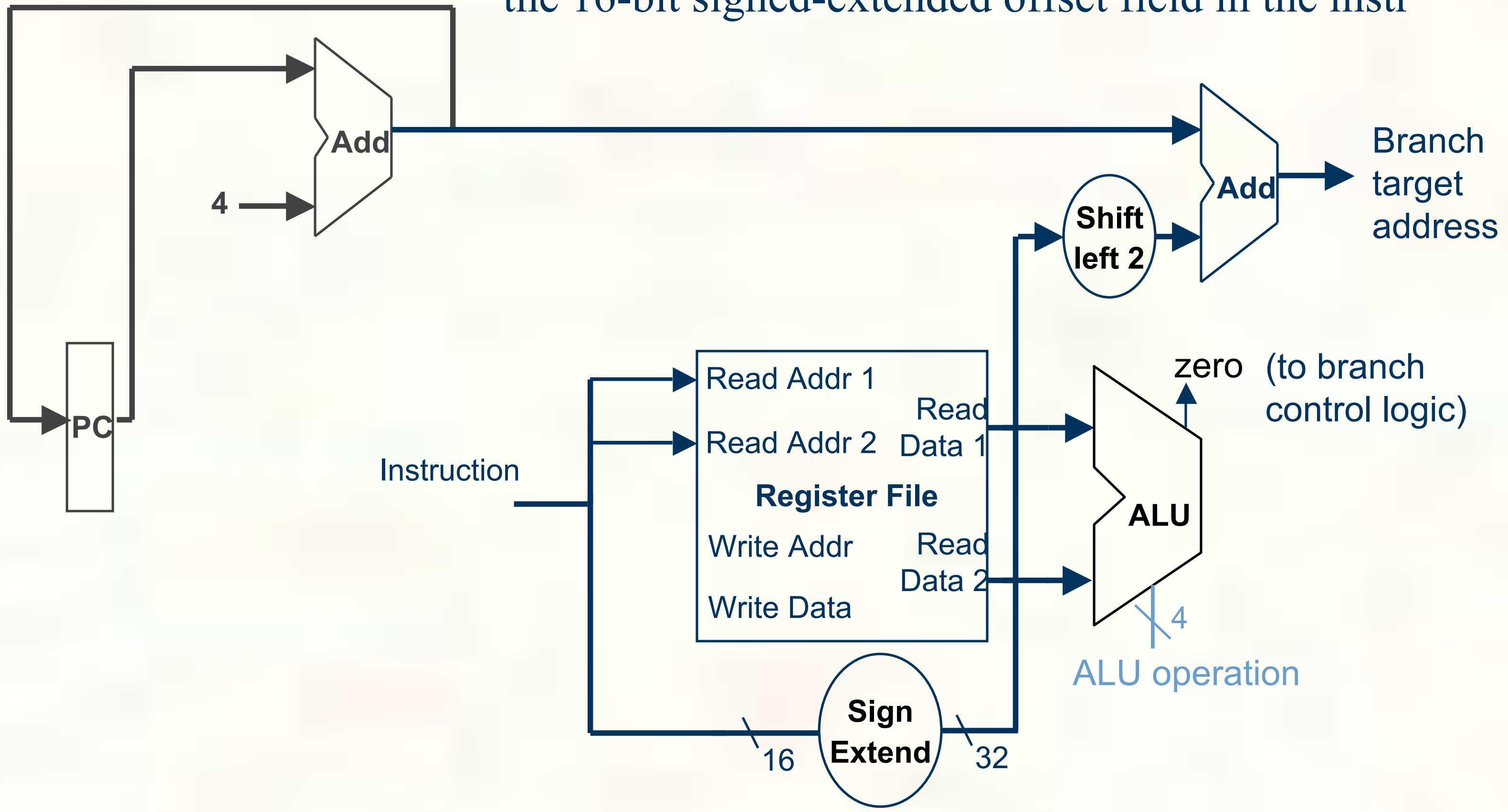


Branch Instructions

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

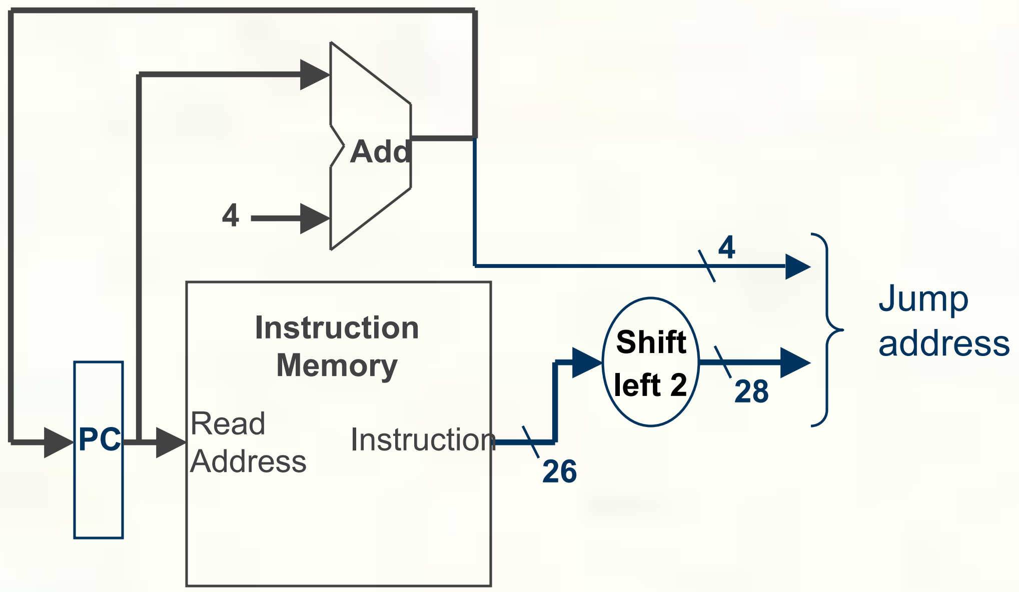
Branch Instructions

- Branch operations involve:
  - compare the operands read from the Register File during decode for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



Jump Instruction

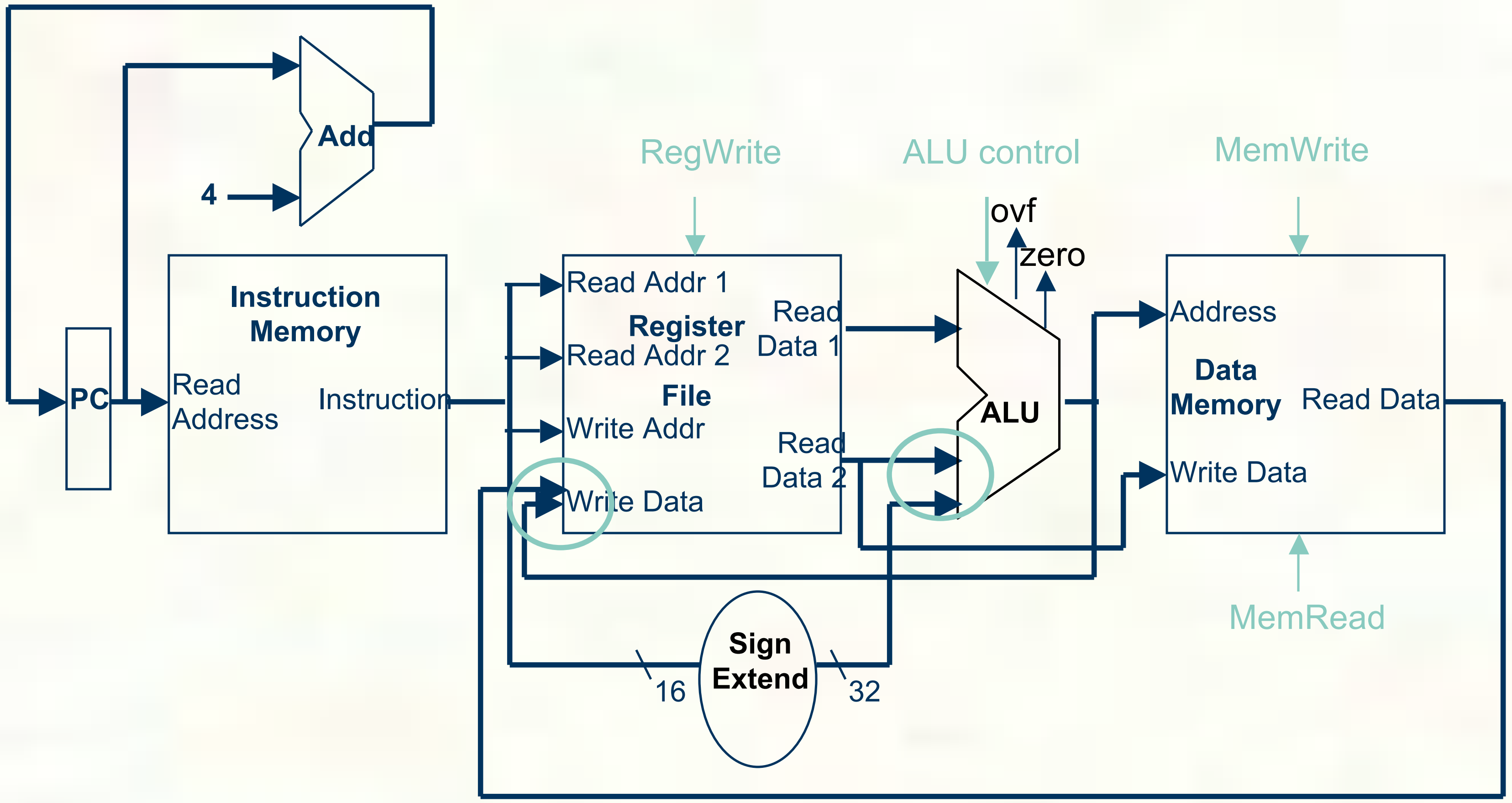
- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



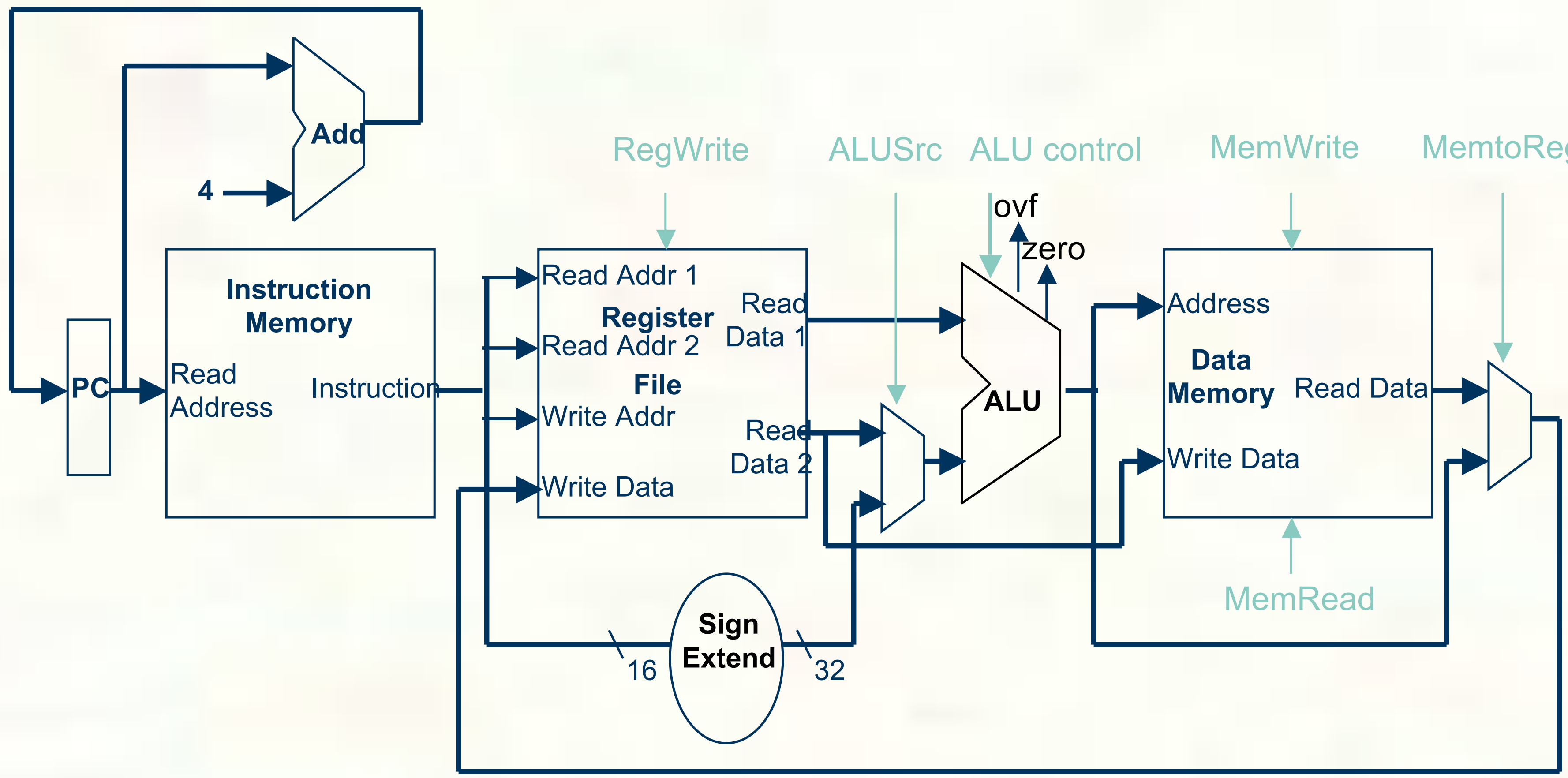
Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

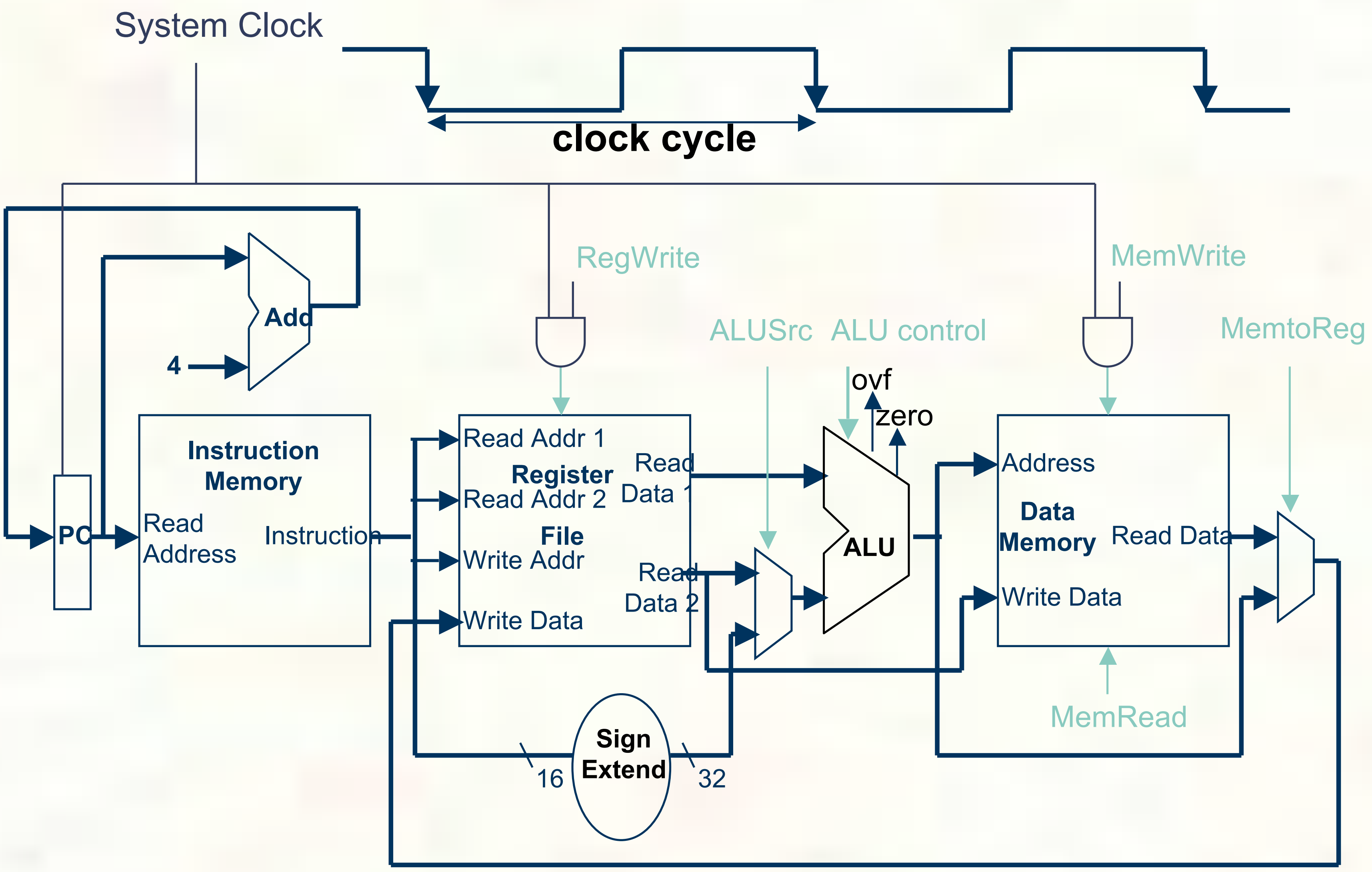
Fetch, R, and Memory Access Portions



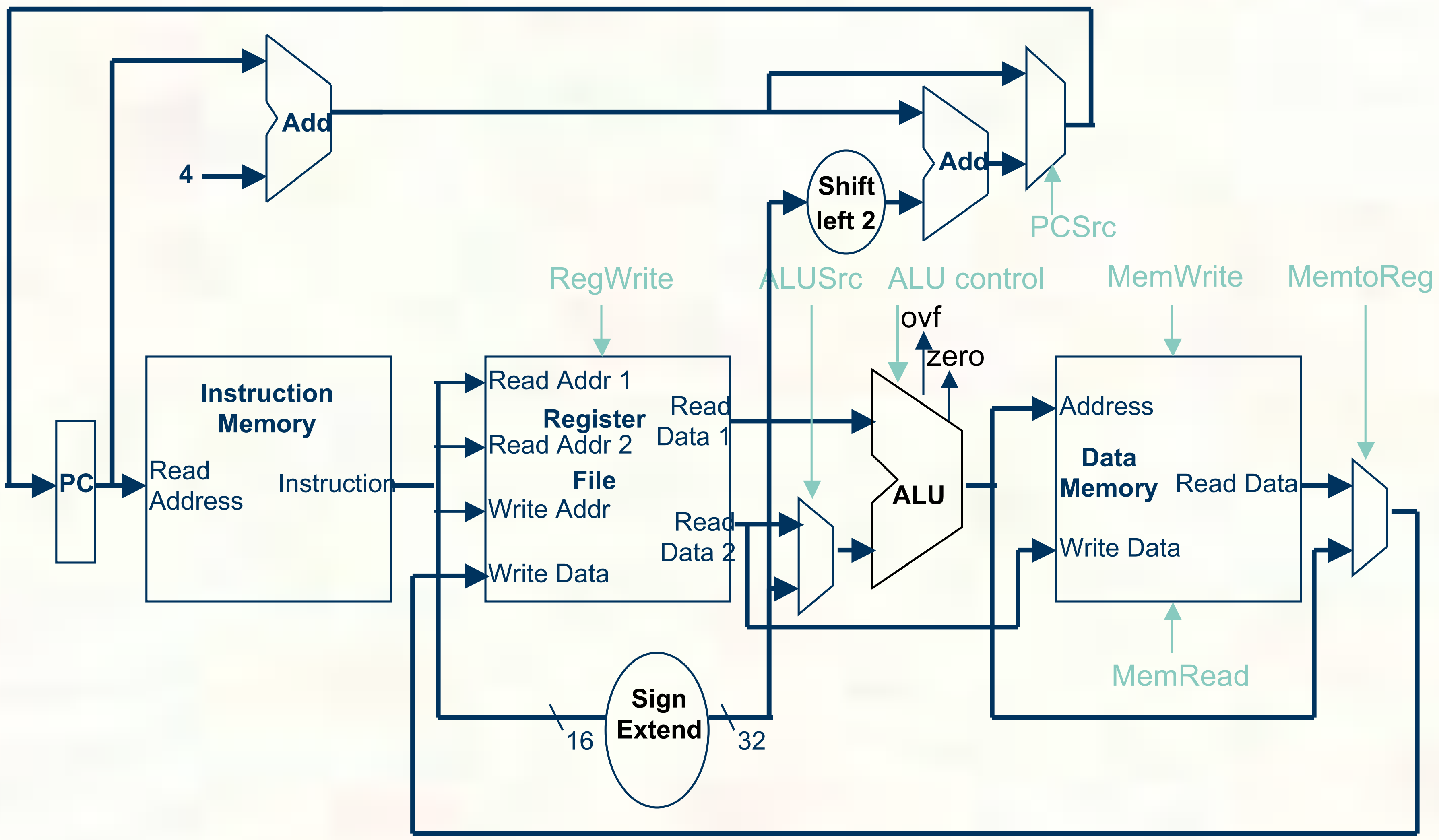
Multiplexor Insertion



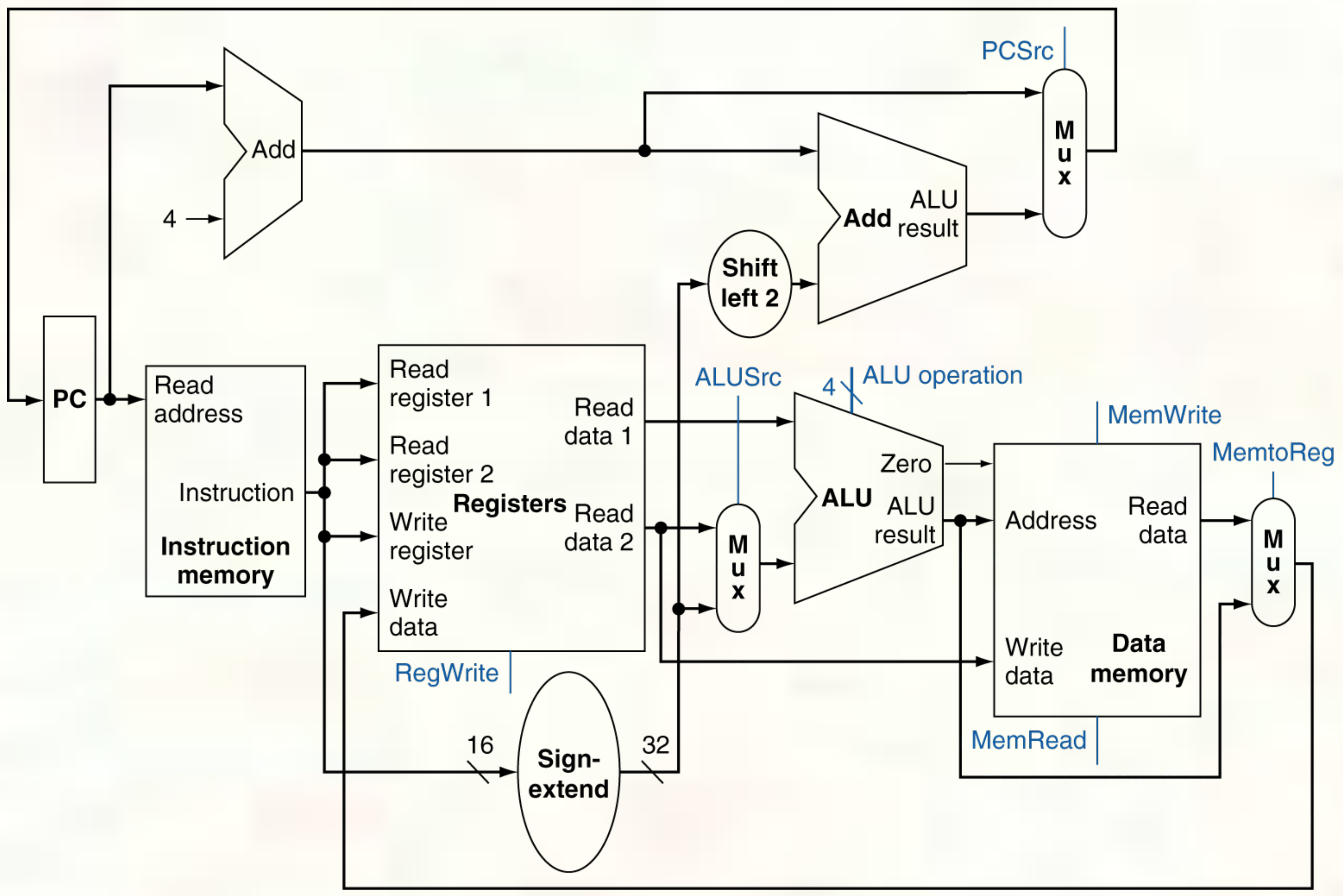
Clock Distribution



Adding the Branch Portion



Full Datapath



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

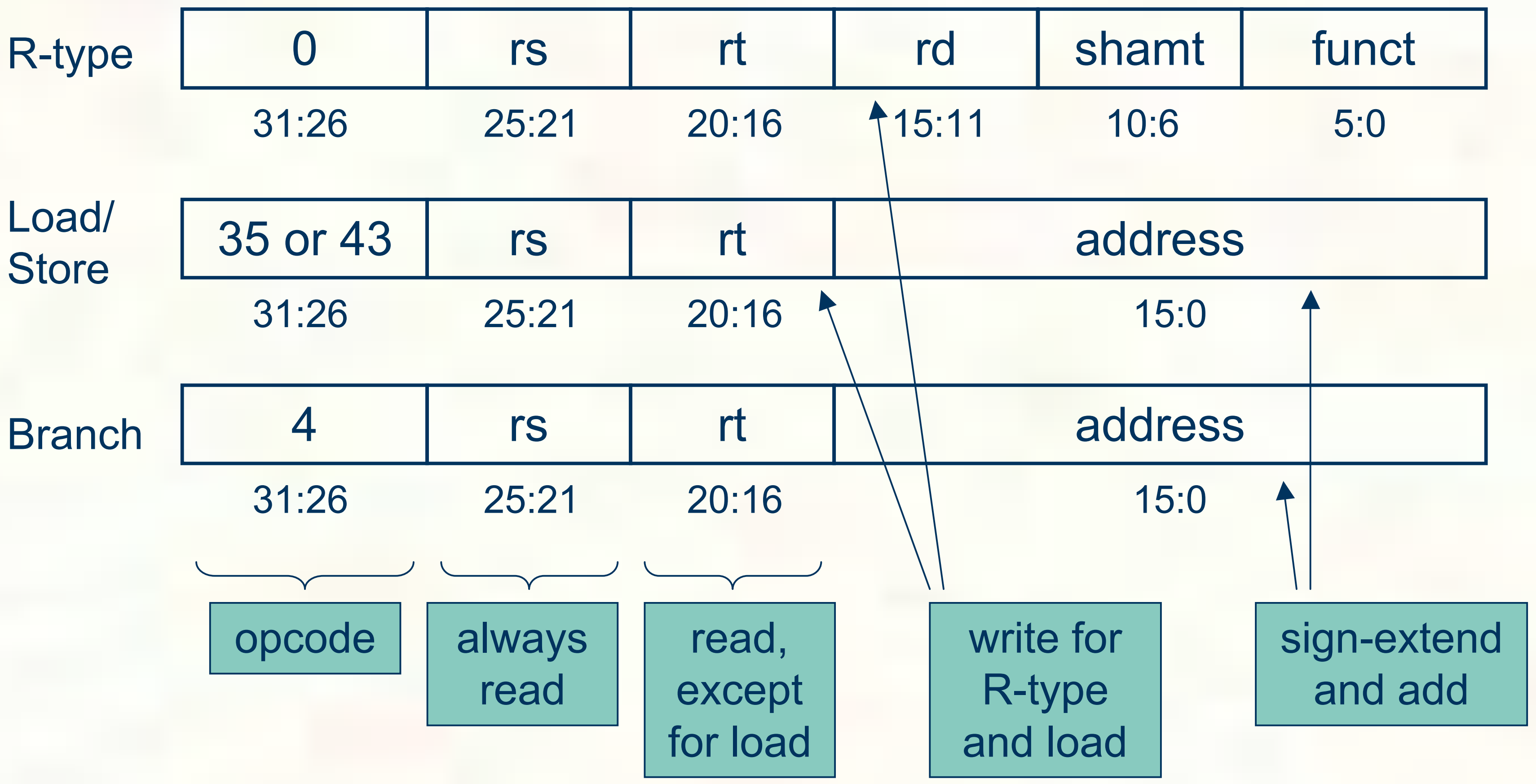
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction

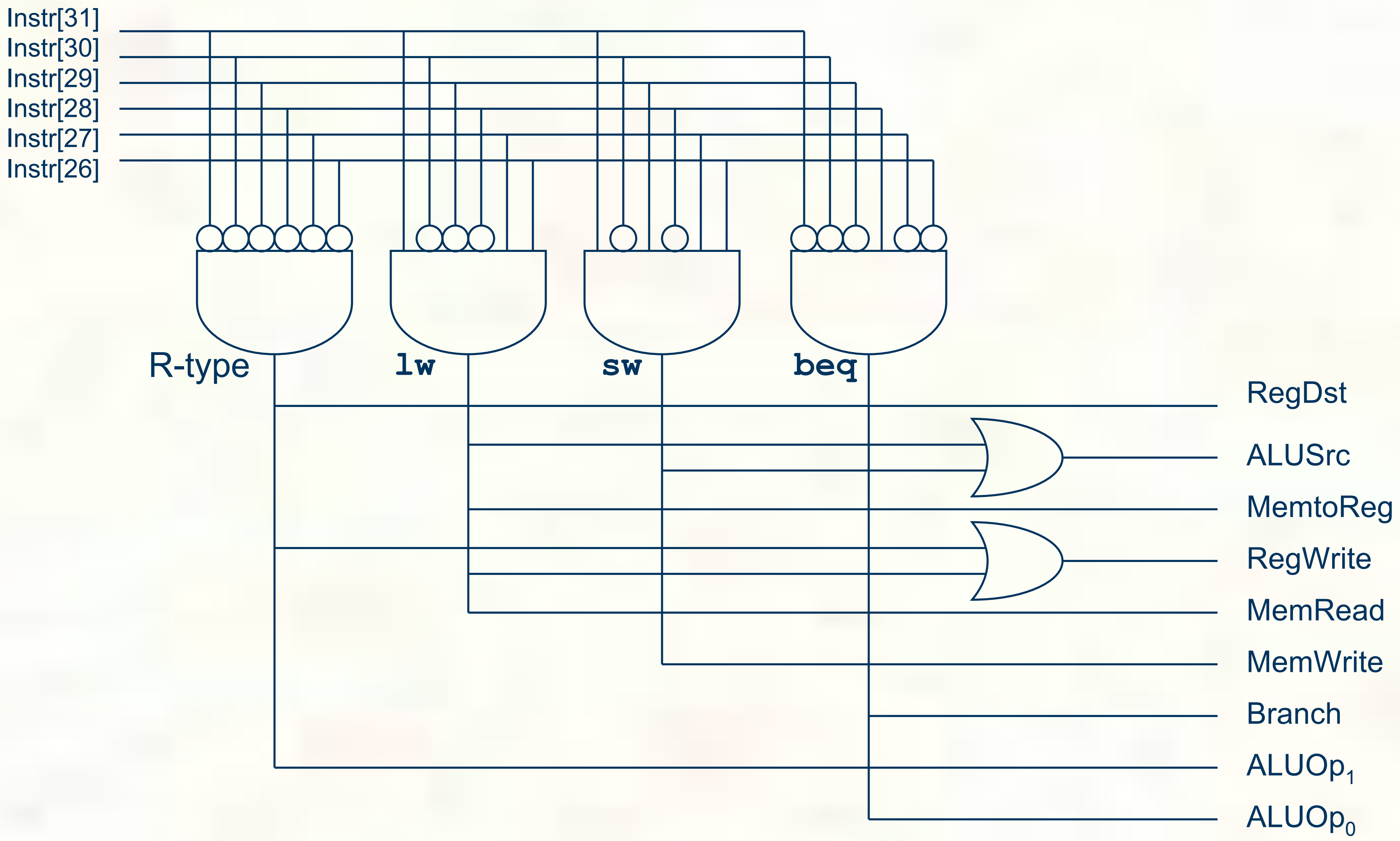


Main Control Unit

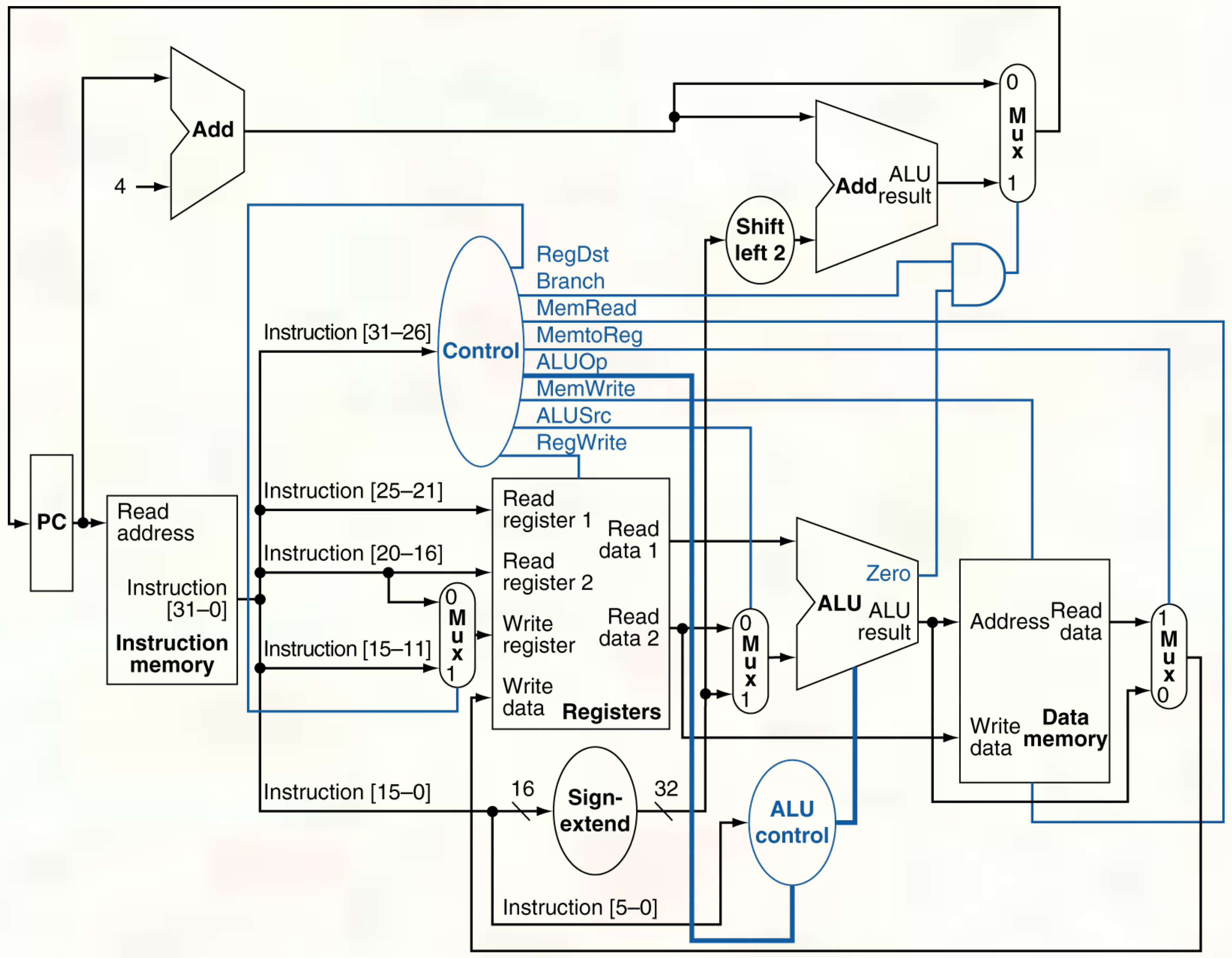
Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011	0	1	1	1	1	0	0	00
sw 101011	X	1	X	0	0	1	0	00
beq 000100	X	0	X	0	0	0	1	01

Control Unit Logic

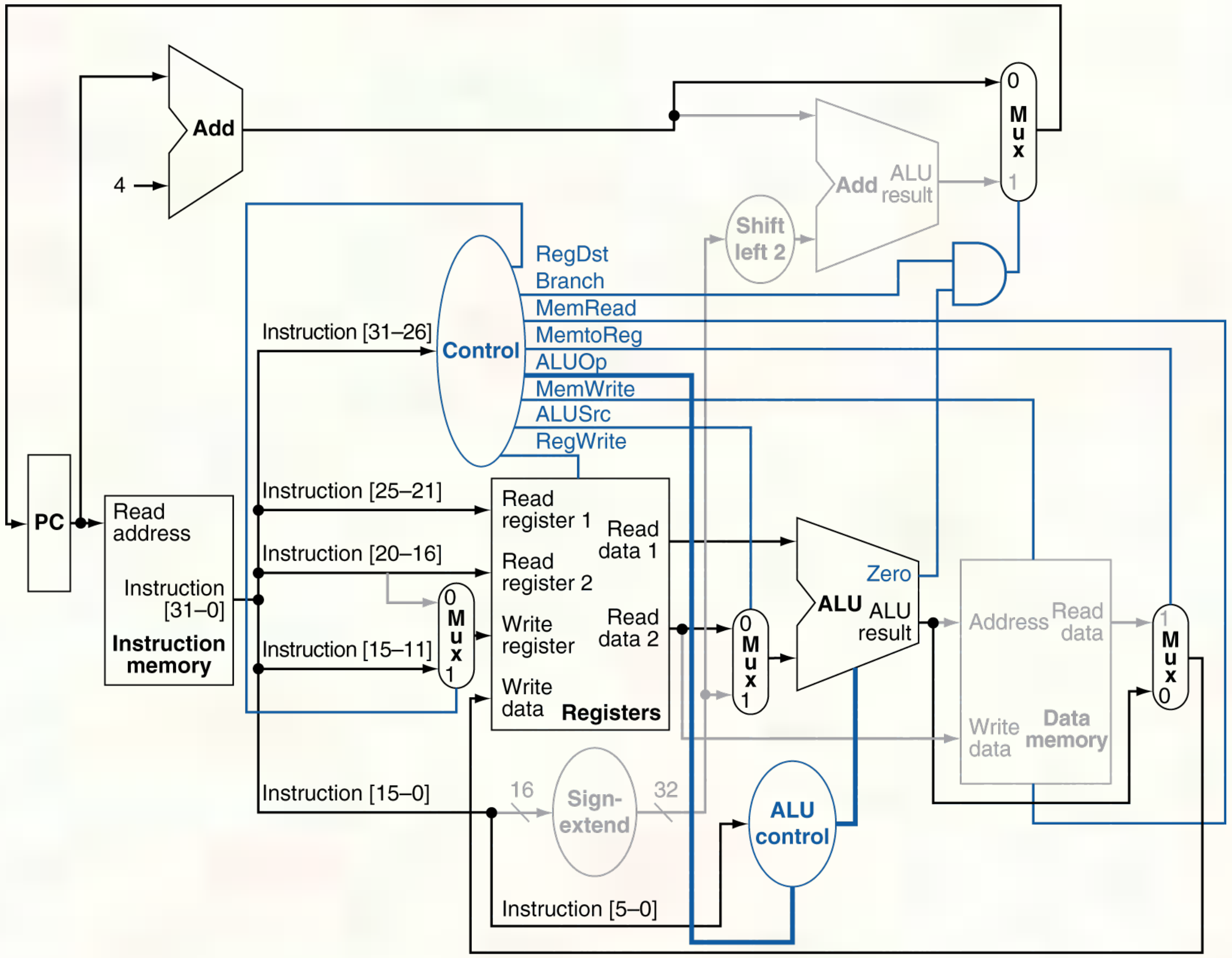
- Design the Main Control logic from the truth table



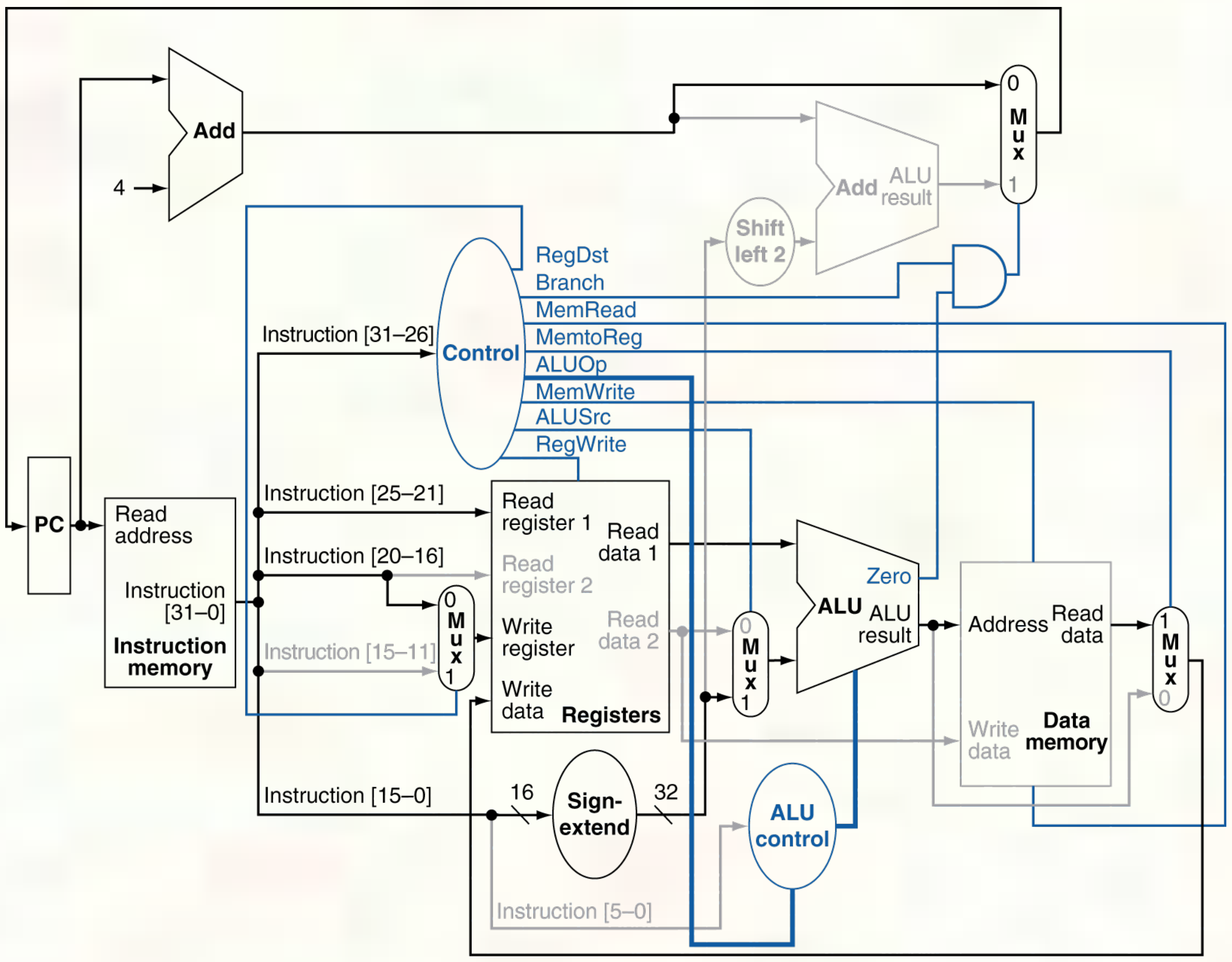
Datapath With Control



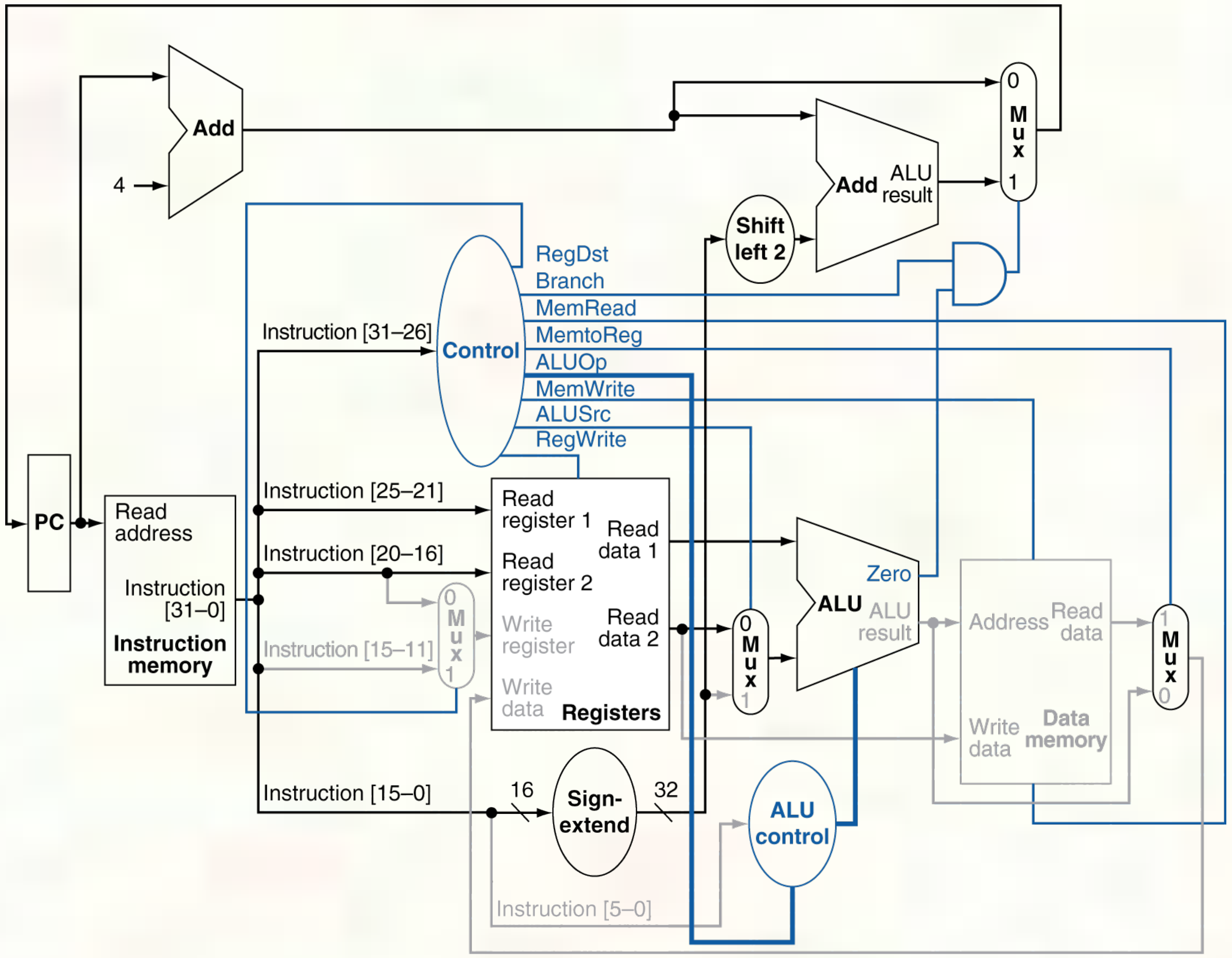
R-Type Instruction



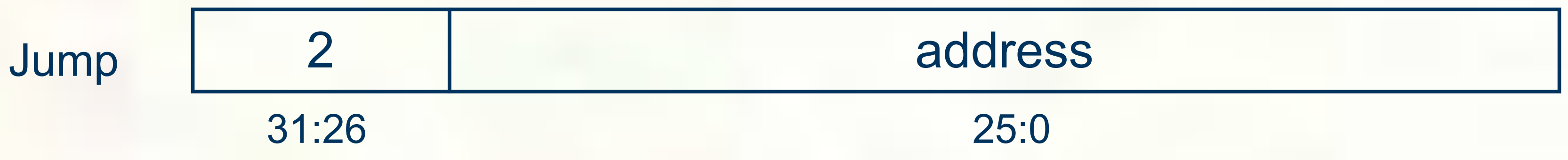
Load Instruction



Branch-on-Equal Instruction

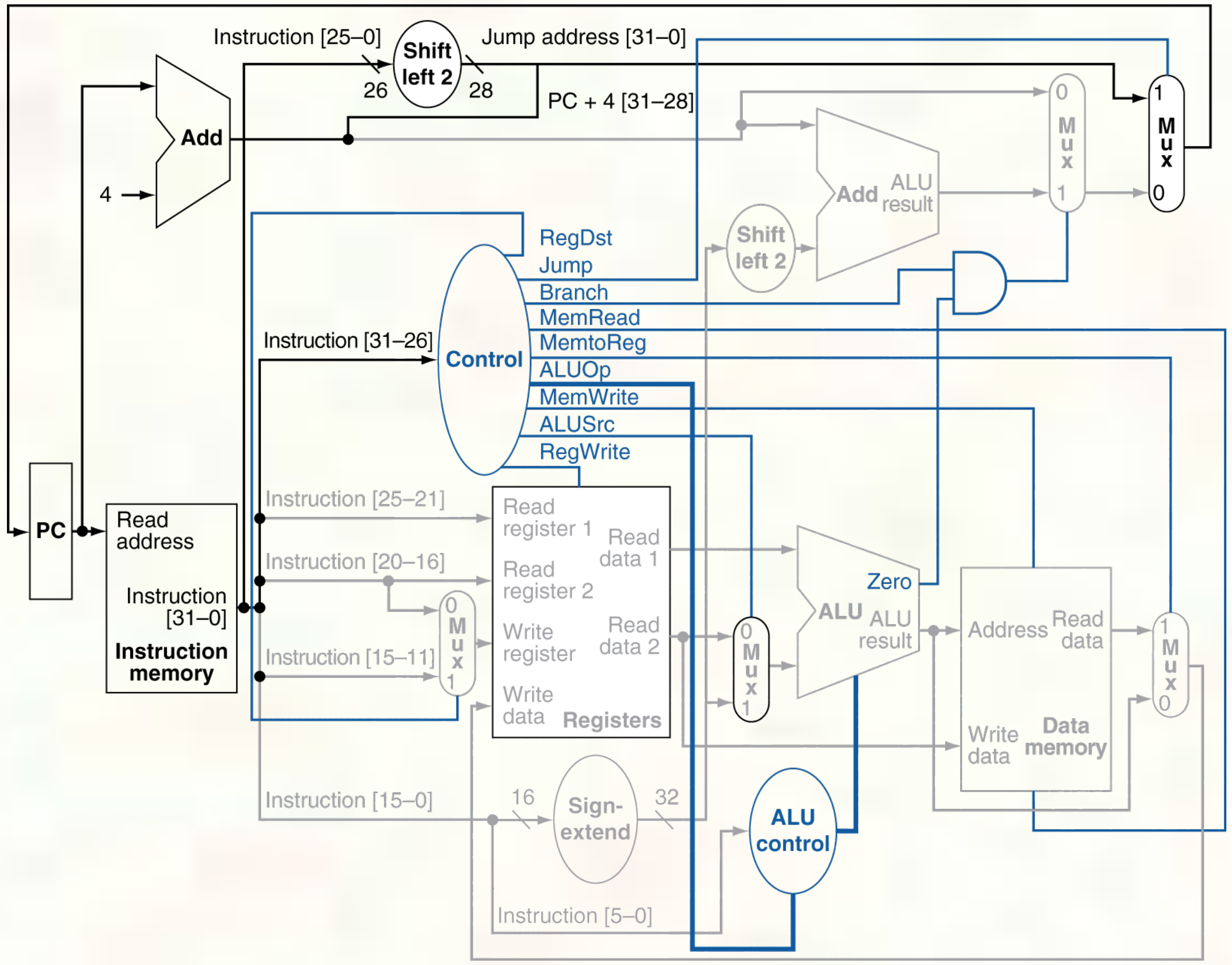


Implementing Jumps



- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added



Main Control Unit

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp	Jump
R-type 000000	1	0	0	1	0	0	0	10	0
lw 100011	0	1	1	1	1	0	0	00	0
sw 101011	X	1	X	0	0	1	0	00	0
beq 000100	X	0	X	0	0	0	1	01	0
j 000010	X	X	X	0	0	0	X	XX	1

Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - 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 will improve performance by pipelining

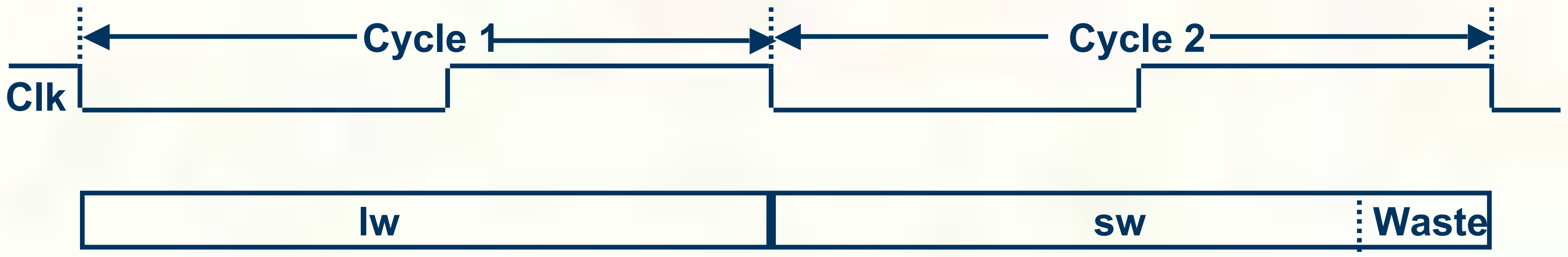
Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires) except:
  - Instruction and Data Memory (4 ns)
  - ALU and adders (2 ns)
  - Register File access (reads or writes) (1 ns)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	4	1	2		1	8
load	4	1	2	4	1	12
store	4	1	2	4		11
beq	4	1	2			7
jump	4					4

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the **slowest** instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle but
- Is simple and easy to understand

Next Lecture

- MIPS pipelined implementation
  - Rest of chapter 4