CO20-320241

Computer Architecture and Programming Languages

CAPL

Lecture 16 & 17

Dr. Kinga Lipskoch

Fall 2019

Floating Point Addition

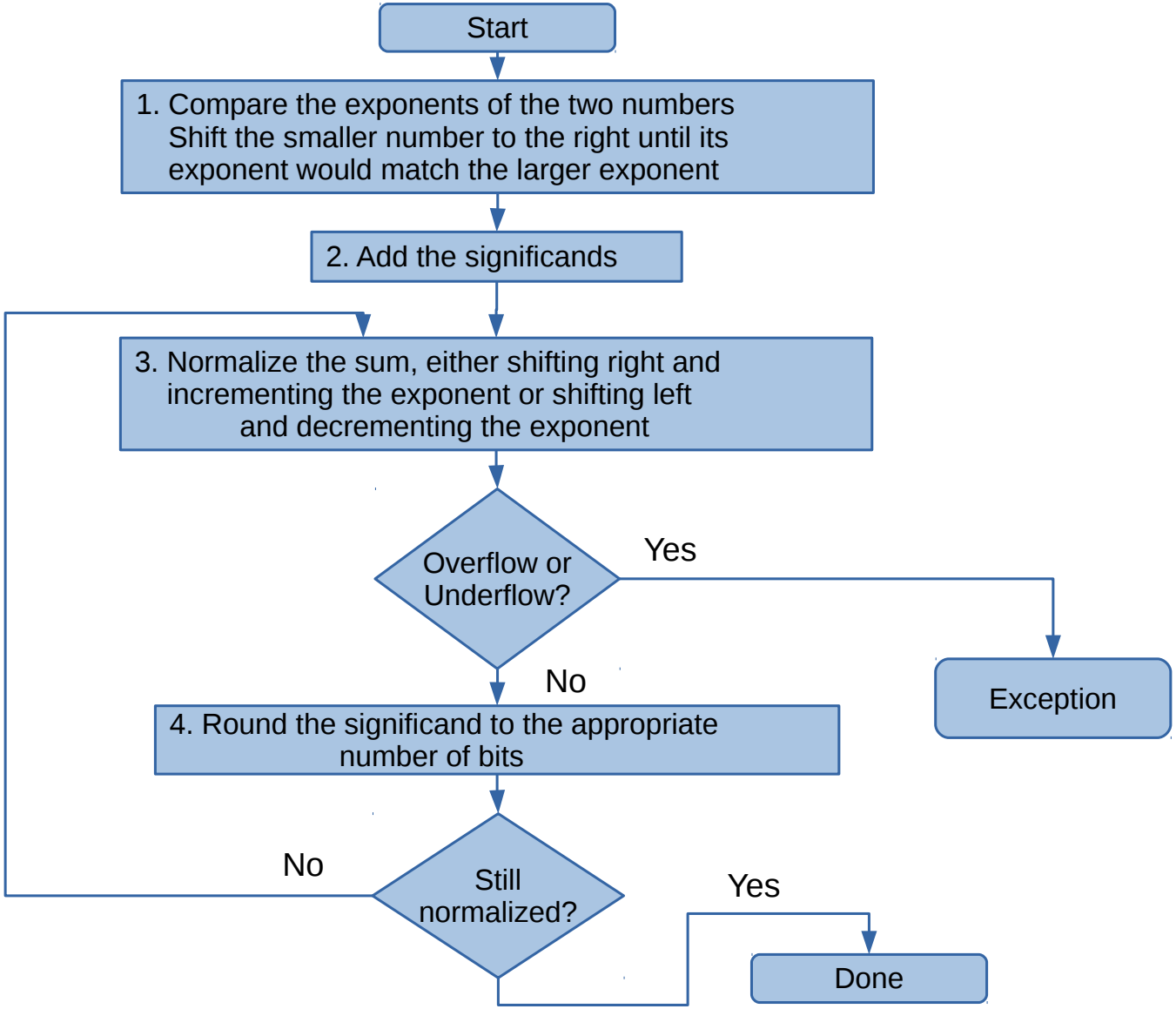
- $0.5_{ten} = 1/2_{ten} = 1/2^1_{ten}$
  $= 0.1_{two} = 0.1_{two} * 2^0 = 1.000_{two} * 2^{-1}$
  $-0.4375_{ten} = -7/16_{ten} = -7/2^4_{ten}$
  $= -0.0111_{two} = -0.0111_{two} * 2^0 = -1.110_{two} * 2^{-2}$
- Lesser exponent shifted right until exponent matches larger number
  $-1.110_{two} * 2^{-2} = -0.111_{two} * 2^{-1}$
- Add significands
  $1.000_{two} * 2^{-1} + (-0.111_{two} * 2^{-1}) = 0.001_{two} * 2^{-1}$
- Normalize the sum
  $0.001_{two} * 2^{-1} = 1.000_{two} * 2^{-4}$
- Round the sum (already fits into 4 bits)
  $1.000_{two} * 2^{-4} = 0.0001_{two} = 1/2^4_{ten} = 0.0625$

Floating Point Addition

Addition (and subtraction)

$(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$

- Step 1: Restore the hidden bit in $F1$ and in $F2$
- Step 2: Align fractions by right shifting $F2$ by $E1 - E2$ positions (assuming $E1 \geq E2$) keeping track of (three of) the bits shifted out in a round bit, a guard bit, and a sticky bit
- Step 3: Add the resulting $F2$ to $F1$ to form $F3$
- Step 4: Normalize $F3$ (so it is in the form $1.XXXXX\ldots$)
  - If $F1$ and $F2$ have the same sign $\rightarrow$ $F3 \in [1, 4) \rightarrow$ 1 bit right shift $F3$ and increment $E3$
  - If $F1$ and $F2$ have different signs $\rightarrow$ $F3$ may require many left shifts each time decrementing $E3$
- Step 5: Round $F3$ and possibly normalize $F3$ again
- Step 6: Rehide the most significant bit of $F3$ before storing the result


Start
1. Compare the exponents of the two numbers
Shift the smaller number to the right until its
exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and
incrementing the exponent or shifting left
and decrementing the exponent
Overflow or
Underflow?
Yes
Exception
No
4. Round the significand to the appropriate
number of bits
Still
normalized?
No
Yes
Done

MIPS Floating Point Instructions

- MIPS has a separate Floating Point Register File (`$f0`, `$f1`, ..., `$f31`) (whose registers are used in pairs for double precision values) with special instructions to load to and store from them

```
lwc1 $f1,54($s2)        # $f1 = Memory[$s2+54]
swc1 $f1,58($s4)        # Memory[$s4+58] = $f1
```

- And supports IEEE 754 single

```
add.s $f2,$f4,$f6       # $f2 = $f4 + $f6
```

and double precision operations

```
add.d $f2,$f4,$f6       # $f2||$f3 = $f4||$f5 + $f6||$f7
```

similarly for `sub.s`, `sub.d`, `mul.s`, `mul.d`, `div.s`, `div.d`

Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines
- Algorithm choice is important and may lead to hardware optimizations for both space and time (e.g., multiplication)

Floating Point Issues

- Integers can represent every number between smallest and largest number
- Floating point numbers only represent an approximation, for numbers they cannot really represent
  - "gap" between 0 and smallest positive number
- Infinite number of real numbers between 0 and 1, but no more than 2^{53} numbers can be represented in double precision format
  - IEEE 754 keeps two extra bits during intermediate results "guard" and "round"
  - sticky bit
  - four rounding modes

Guard and Round

- Add $2.56_{ten} * 10^0$ to $2.34_{ten} * 10^2$
- Assume we have only three significant digits
- Additionally guard and round
  - shift smaller number right to align exponents
  - able to represent the two least significant digits when we align exponents

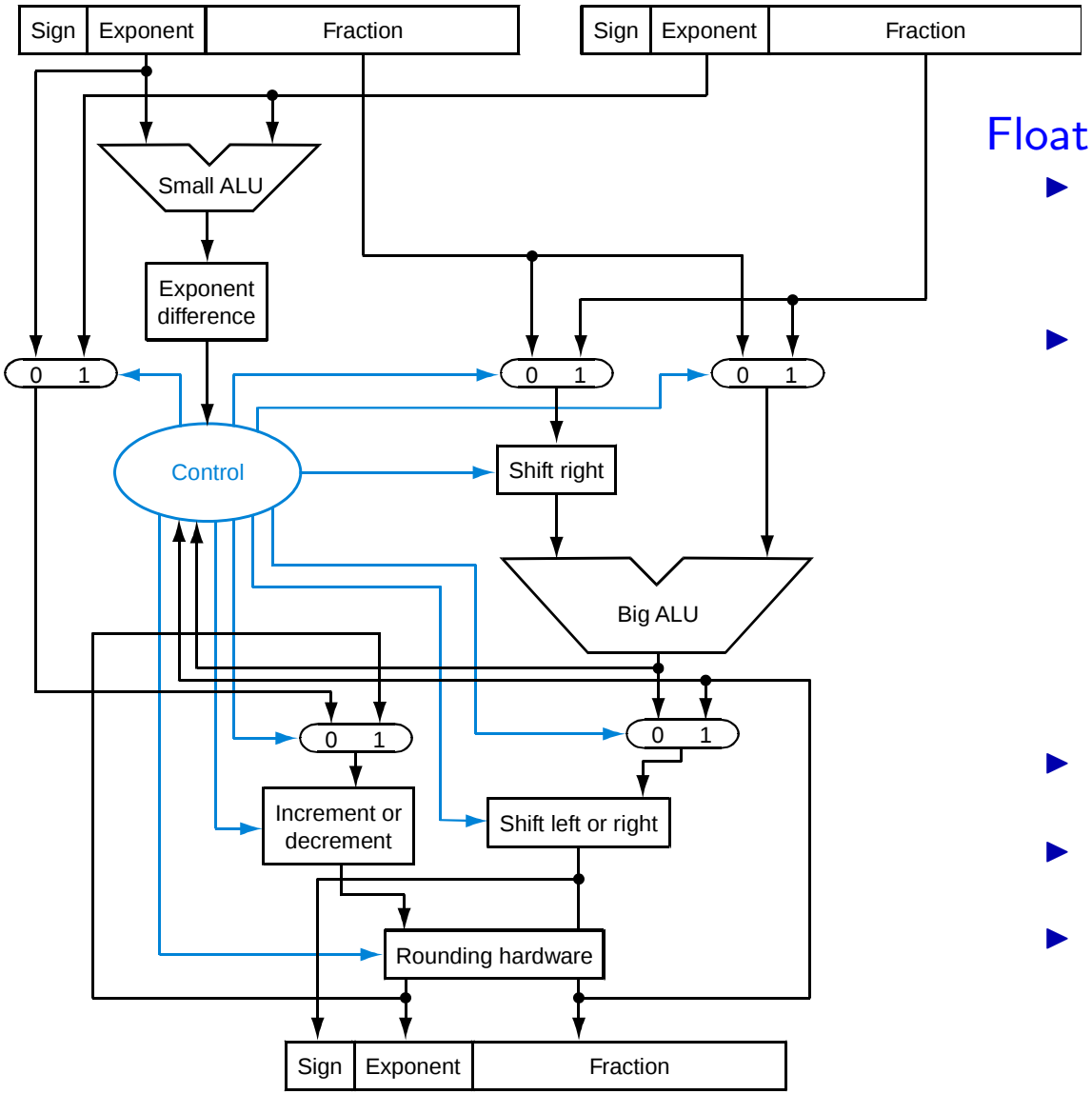
$$\begin{array}{rl} & 2.3400_{\text{ten}} \\ + & 0.0256_{\text{ten}} \\ = & 2.3656_{\text{ten}} \end{array}$$

Rounding up yields
$2.37_{ten} * 10^2$

- Without guard drop two digits from calculation, result would be $2.36_{ten} * 10^2$

Round, Guard and Sticky Bit

- Example above may only need one bit, multiply can need two
  - binary product may have leading 0 bit
  - normalization must shift product left, round bit is being used to round
- Rounding modes includes "nearest even"
- What to do with 0.50?
  - half the time up, half the time down
  - standard says if least significant bit retained in such case is odd round up, otherwise truncate
- Sticky bit is set whenever nonzero bits to the right of round bit
  - allows to see difference between $0.50...00_{ten}$ and $0.50...01_{ten}$



Floating point adder:

- Exponent of one operand is subtracted from the other to determine which is larger and by how much
- Difference controls three multiplexers: from left to right
  - Select larger exponent
  - Significand of smaller number
  - Significand of larger number
  - Smaller significand is shifted right, then significands are added using Big ALU
- Normalization step shifts sum left or right and increments or decrements the exponent
- Rounding creates final result
- May require normalizing again

Performance

- What is performance?
- Measuring performance
- Performance metrics
- Performance evaluation
- Why does some hardware perform better with different programs?
- What performance factors are related to hardware?

Performance Metrics

- Purchasing perspective
  - given a collection of machines, which has the
    - best performance?
    - least cost?
    - best cost/performance rate?
- Design perspective
  - faced with design options, which has the
    - best performance improvement?
    - least cost?
    - best cost/performance rate?
- Both require
  - basis for comparison
  - metric for evaluation
- Goal is to understand what factors in the architecture contribute to the overall system performance and the relative importance (and cost) of these factors

Speed Performance Definition

- Normally interested in reducing
  - Response time (a.k.a. execution time) – the time between the start and the completion of a task
    - Important to individual users
  - Thus, to maximize performance, need to minimize execution time

$$\text{performance}_x = 1 / \text{execution\_time}_x$$

- If X is n times faster than Y, then

$$\frac{\text{performance}_x}{\text{performance}_y} = \frac{\text{execution\_time}_y}{\text{execution\_time}_x} = n$$

Very Simple Example

- How much faster is A than B?

$$\text{Performance}_A \;/\; \text{Performance}_B = n$$
$$\text{Execution\_time}_B \;/\; \text{Execution\_time}_A = n$$

- If ratio

$$25/20 = 1.25$$

then A is by a factor of 1.25 faster than B

Computer Performance

- Response Time (latency)
  - How long does it take for my job to run?
  - How long does it take to execute a job?
  - How long must I wait for the database query?
- Throughput
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How much work is getting done?
- Decreasing response time almost always improves throughput

What is Meant by Time

- Time is the measure of computer performance
- The computer that performs the same amount of work in the least time is the fastest
  - Elapsed Time
    - counts everything (disk and memory accesses, I/O, etc.)
    - a useful number, but often not good for comparison purposes
  - CPU time
    - does not count I/O or time spent running other programs
    - can be broken up into system time and user time
- Our focus: user CPU time
  - time spent executing the lines of code that are "in" our program

Performance Factors

- Want to distinguish elapsed time and the time spent on our task
- CPU execution time (CPU time) – time the CPU spends working on a task
  - Does not include time waiting for I/O or running other programs

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

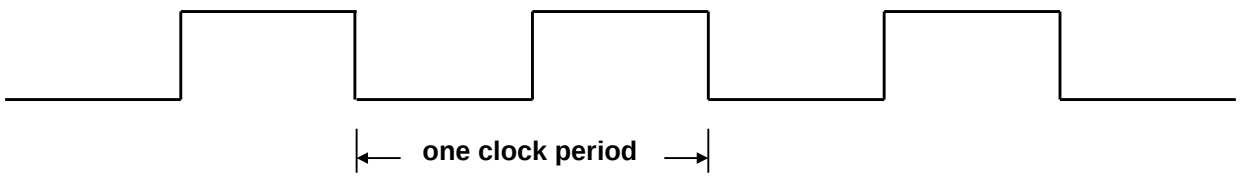
$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

- Can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program

Machine Clock Rate

Clock rate (CR) (unit: MHz, GHz) is inverse of clock cycle (CC) time (clock period)

$$CC = 1/CR$$



10 nsec clock cycle $\rightarrow$ 100 MHz clock rate
5 nsec clock cycle $\rightarrow$ 200 MHz clock rate
2 nsec clock cycle $\rightarrow$ 500 MHz clock rate
1 nsec clock cycle $\rightarrow$ 1 GHz clock rate
500 psec clock cycle $\rightarrow$ 2 GHz clock rate
250 psec clock cycle $\rightarrow$ 4 GHz clock rate
200 psec clock cycle $\rightarrow$ 5 GHz clock rate

Clock Cycles Per Instruction

- Not all instructions take the same amount of time to execute
  - One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction

# CPU clock cycles for a program	=	Instructions for a program	x	Average clock cycles per instruction

- Clock cycles per instruction (CPI) – the average number of clock cycles each instruction takes to execute
  - A way to compare two different implementations of the same ISA

Comparing Code Segments

- Compiler designer needs to decide between two code sequences for particular computer
- Knows that:
  - Sequence 1: $2 + 1 + 2 = 5$ instructions
  - Sequence 2: $4 + 1 + 1 = 6$ instructions

    CPU clock cycles $= \sum_{i=1}^{n} CPI_i \cdot IC_i$
  - Is Sequence 1 faster?
  - CPI varies by instruction mix
    - instruction count & CPI needs to be compared

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

Code seq.	Instr. counts for instr. class		
	A	B	C
1	2	1	2
2	4	1	1

Effective CPI

- Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging

  Overall effective CPI $= \sum_{i=1}^{n} \frac{CPI_i \cdot IC_i}{IC_{total}}$

  - where n is the number of instruction classes
  - CPI_i is the (average) number of clock cycles per instruction for that instruction class
  - IC_i is the overall number of instructions of class i and IC_{total} is the total number of instructions
    - weight of instructions of class i
- The overall effective CPI varies by instruction mix – a measure of the dynamic frequency of instructions across one or many programs

Performance Equation

- Our basic performance equation is then

$$\text{CPU time} = \text{Instruction count} * \text{CPI} * \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{Instruction count} * \text{CPI}}{\text{Clock rate}}$$

- These equations contain the three key factors that affect performance
  - Can measure the CPU execution time by running the program
  - The clock rate is usually given
  - Can measure overall instruction count by using profilers/simulators without knowing all of the implementation details
  - CPI varies by instruction type and ISA implementation for which we must know the implementation details

Evaluating Performance

CPU time = Instruction count x CPI x Clock cycle

- Algorithm might also effect CPI, by using floating-point rather than integer arithmetic
- Language also affects CPI, for example Java allows more data abstraction $\rightarrow$ more indirect calls $\rightarrow$ higher CPI

A Simple Example (1)

Op	Frequency	CPI_i	Freq * CPI
ALU	50%	1	
Load	20%	5	
Store	10%	3	
Branch	20%	2	
			Σ =

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
- How does this compare with using branch prediction to shave a cycle off the branch time?
- What if two ALU instructions could be executed at once?

A Simple Example (2)

Op	Frequency	CPI_i	Freq * CPI	
ALU	50%	1	0.5	0.5
Load	20%	5	1.0	0.4
Store	10%	3	0.3	0.3
Branch	20%	2	0.4	0.4
			∑ = 2.2	∑ = 1.6

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

A Simple Example (3)

Op	Frequency	CPI_i	Freq * CPI		
ALU	50%	1	0.5	0.5	0.5
Load	20%	5	1.0	0.4	1.0
Store	10%	3	0.3	0.3	0.3
Branch	20%	2	0.4	0.4	0.2
			∑ = 2.2	∑ = 1.6	∑ = 2.0

- 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 IC 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 IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?

A Simple Example (4)

Op	Frequency	CPI_i	Freq * CPI			
ALU	50%	1	0.5	0.5	0.5	0.25
Load	20%	5	1.0	0.4	1.0	1.0
Store	10%	3	0.3	0.3	0.3	0.3
Branch	20%	2	0.4	0.4	0.2	0.4
			∑ = 2.2	∑ = 1.6	∑ = 2.0	∑ = 1.95

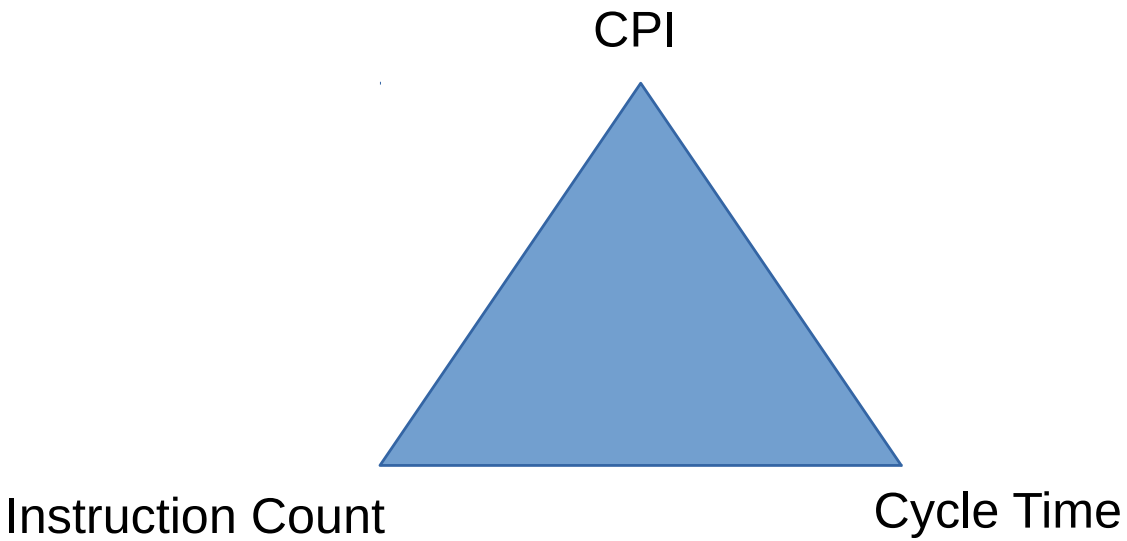
- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- How does this compare with using branch prediction to shave a cycle off the branch time?
  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?
  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

Summary: Evaluating ISA (1)

- 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 cycles are required per instruction?
- Best Metric: Time to execute the program

Summary: Evaluating ISA (2)

Depends on the instructions set, the processor organization, and compilation techniques



MIPS as a Performance Measure

- MIPS = Million Instructions Per Second

$$\text{MIPS} = \frac{\text{instruction count}}{\text{execution time} * 10^6} = \frac{\text{instruction count}}{\text{CPU clocks} * \text{cycle time} * 10^6}$$

$$= \frac{\text{instruction count} * \text{clock rate}}{\text{instruction count} * \text{CPI} * 10^6} = \frac{\text{clock rate}}{\text{CPI} * 10^6}$$

- A faster machine has a higher MIPS rate

$$\text{execution time} = \frac{\text{instruction count}}{\text{MIPS} * 10^6}$$

MIPS Metric Not Adequate

- MIPS does not take into account the instruction set (capabilities or complexity of instructions)
  - different ISAs not comparable
- The MIPS rate depends on the program and its instruction mix
- MIPS are not constant, even on a single machine
- MIPS can vary inversely with performance

MIPS Metric not Adequate: Example

- Three instruction classes and given CPI
- Two compilers

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

	Instr. counts (in billions) for instruction class		
Code from	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

- Cycles
  - CPU clock cycles $= \sum CPI_i * IC_i$
  - Compiler 1: 7 billion instructions, 10 billion cycles
  - Compiler 2: 12 billion instructions, 15 billion cycles

Execution Time

CPU execution time for a program	=	$\frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$		
Execution time $_1$	=	$\frac{10 * 10^9}{4 * 10^9}$	=	2.50 seconds
Execution time $_2$	=	$\frac{15 * 10^9}{4 * 10^9}$	=	3.75 seconds

$\rightarrow$ Compiler 1 generates a faster program

MIPS Rate (1)

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} * 10^6}$$

$$\text{MIPS}_1 = \frac{(5 + 1 + 1) * 10^9}{2.5 * 10^6} =$$

$$\text{MIPS}_2 = \frac{(10 + 1 + 1) * 10^9}{3.75 * 10^6} =$$

MIPS Rate (2)

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} * 10^6}$$

$$\text{MIPS}_1 = \frac{(5 + 1 + 1) * 10^9}{2.5 * 10^6} = 2800$$

$$\text{MIPS}_2 = \frac{(10 + 1 + 1) * 10^9}{3.75 * 10^6} = 3200$$

$\rightarrow$ Compiler 2 has a higher MIPS rating, but the code from Compiler 1 runs faster

Summary

- A given program will require
  - Some number of instructions (machine)
  - Some number of cycles
  - Some number of seconds
- Vocabulary
  - Cycle time (seconds per cycle)
  - Clock rate (cycles per second)
  - CPI (cycles per instruction)
  - MIPS (millions of instructions per second)

Amdahl's Law (1967)

General law concerning the speedup

- Version 1

$$\text{speedup} = \frac{\text{performance after improvement}}{\text{performance before improvement}}$$

$$= \frac{\text{execution time before improvement}}{\text{execution time after improvement}}$$

- Version 2

$$\text{execution time after improvement} = \frac{\text{execution time affected by improvement}}{\text{amount of improvement}} + \text{execution time unaffected}$$

Example

- Program runs 100 sec, of which multiplication operations run 80 sec
- How much do I have to improve speed of multiplication to run the program 5 times as fast?
- 80 sec / n + (100 sec - 80 sec)
- 20 sec = (80 sec / n) + 20 sec
- Not possible
- Performance enhancement is limited by the amount that the improved feature is used

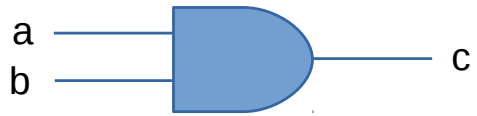
Part III: Datapath and Control

Datapath: Introduction

- A datapath is a collection of functional units, such as arithmetic logic units or multipliers, registers, and buses
- Along with the control unit it composes the central processing unit (CPU)
- Performance of machine is determined by three key factors:
  - instruction count (ISA, compiler, algorithm)
  - clock cycle time (processor implementation)
  - clock cycles per instruction (processor implementation)
- Principles and techniques to implement processor
- Highly abstract and simplified overview
- Enough to implement processor with simple instruction set like the one for MIPS

Review: Hardware Building Blocks to Build an ALU

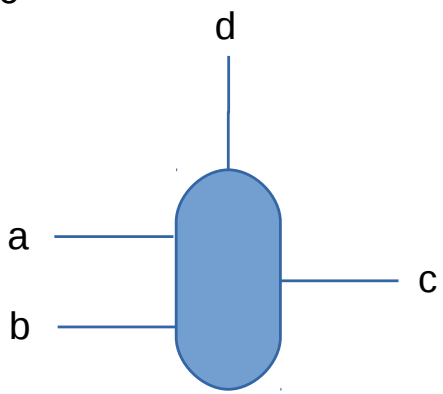
- AND gate ($c = a \cdot b$, logical product)



- OR gate ($c = a + b$, logical sum)

- Inverter ($c = \overline{a}$)

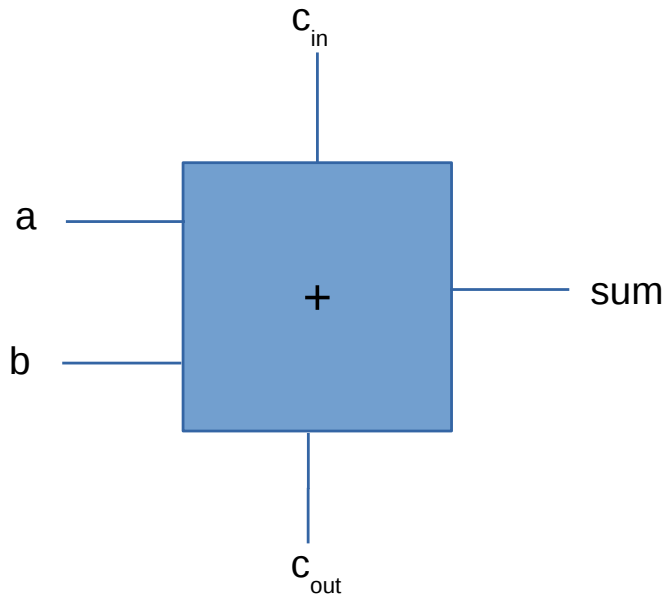
- Multiplexer (if $d == 0$, $c = a$; else $c = b$)
  - Selects one of the inputs to be the output, based on a control input



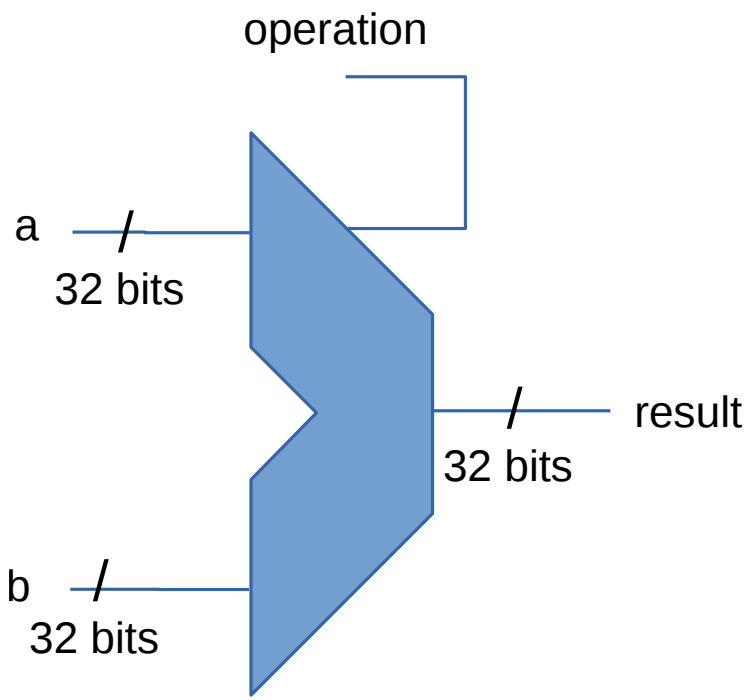
1-bit ALU

A 1-bit adder:

- Full adder
- Also called $(3, 2)$ adder
- 3 inputs, 2 outputs
- $c_{out} = ab + ac_{in} + bc_{in}$
- $sum = a \ xor \ b \ xor \ c_{in}$



ALU



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:
  1. use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  2. decode the instruction (and read registers)
  3. execute the instruction
- All instructions (except `j`) use the ALU after reading the registers
- How? memory-reference? arithmetic? control flow?

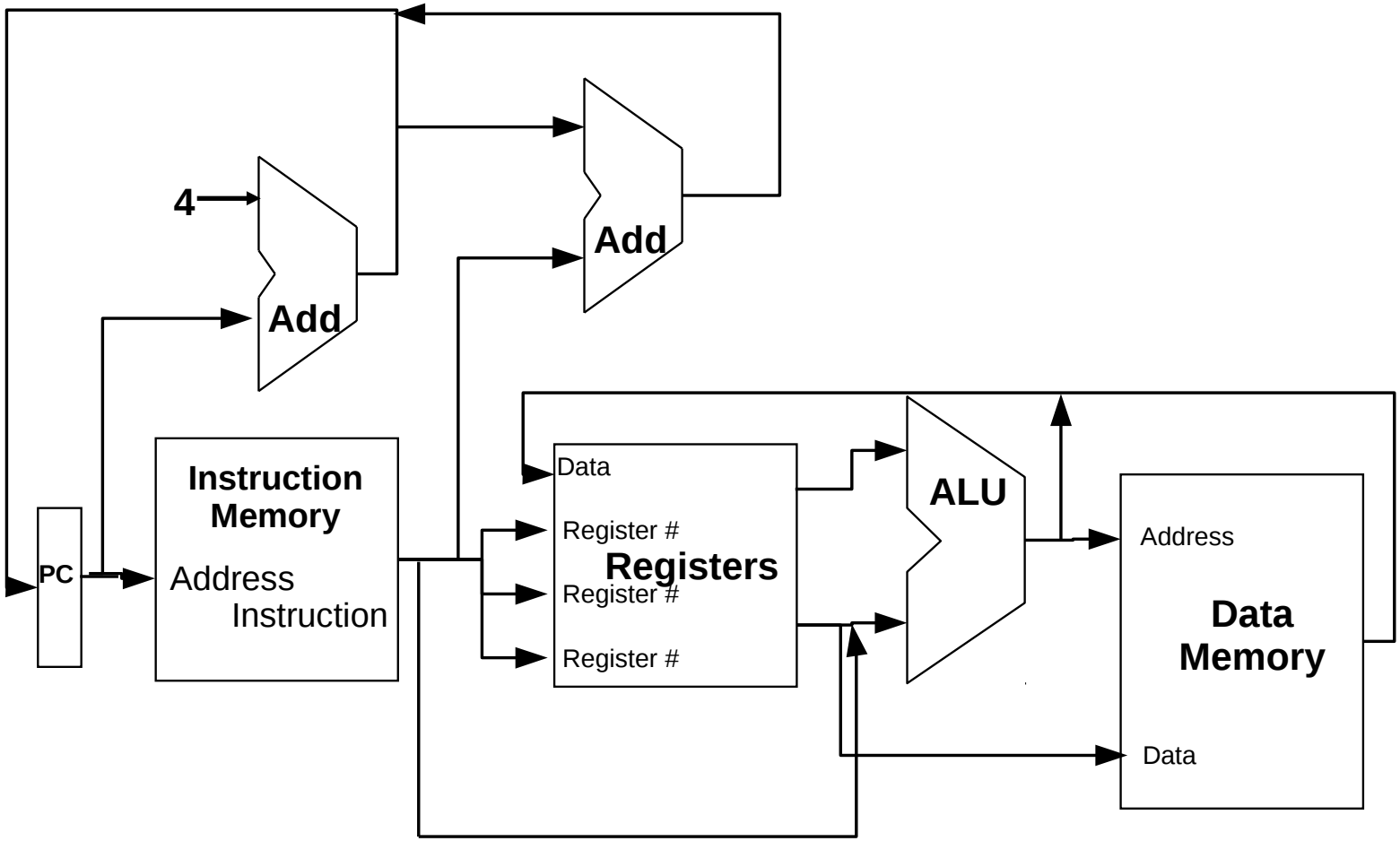
Datapath & Control (1)

- Memory reference instruction
  - use ALU for an address calculation
  -
- Arithmetic-logical instruction
  - use ALU operation execution
  -
- Branch instruction
  - use ALU for comparison
  -

Datapath & Control (2)

- Memory reference instruction
  - use ALU for an address calculation
  - *then needs to access memory to read/store data*
- Arithmetic-logical instruction
  - use ALU operation execution
  - *then needs to write data from ALU to a register*
- Branch instruction
  - use ALU for comparison
  - *based on comparison we might need to change the PC*

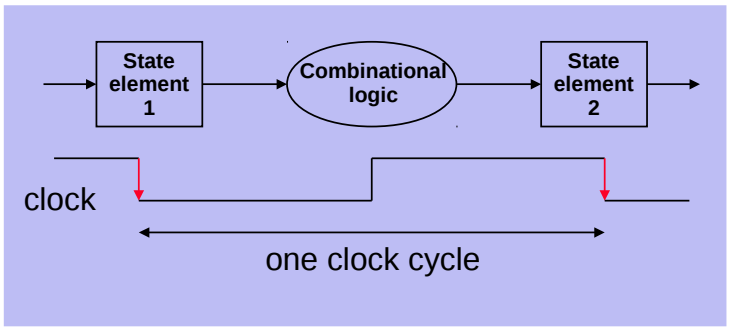
Abstract View of the Implementation of the MIPS Subset



Clocking Methodologies

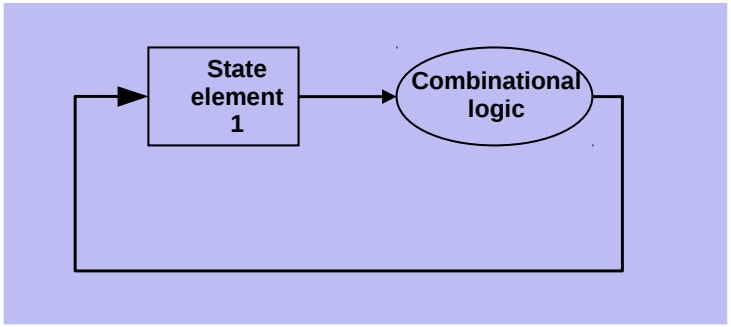
- The clocking methodology defines when signals can be read and when they can be written
  - An edge-triggered methodology
- Typical execution:

- 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

Edge-Triggered Methodology



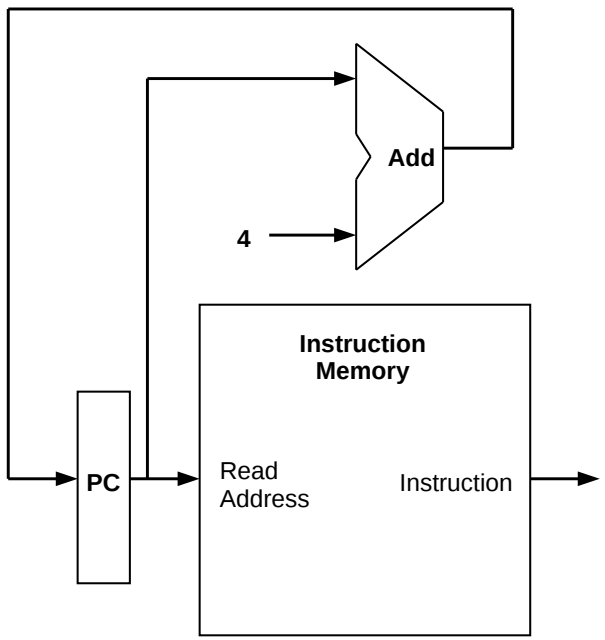
- State element can be read and written within same clock cycle without creating a race
  - cycle must be long enough so values are stable
- Feedback cannot occur within 1 clock cycle
- Typically all state and logic elements have inputs 32 bit wide

Fetching Instructions

Fetching instructions involves:

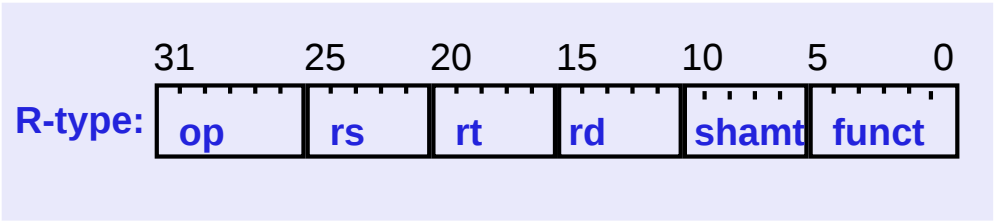
- reading the instruction from the Instruction Memory
- updating the PC to hold the address of the next instruction

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

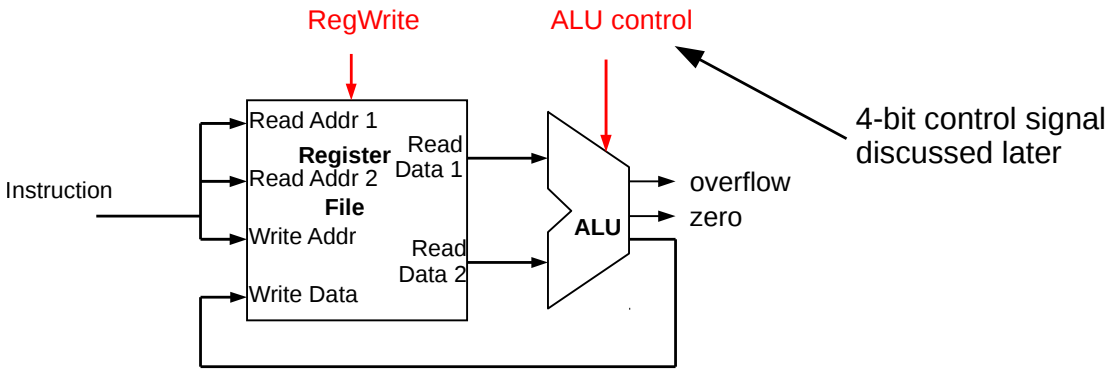


Executing R-format Instructions

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



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

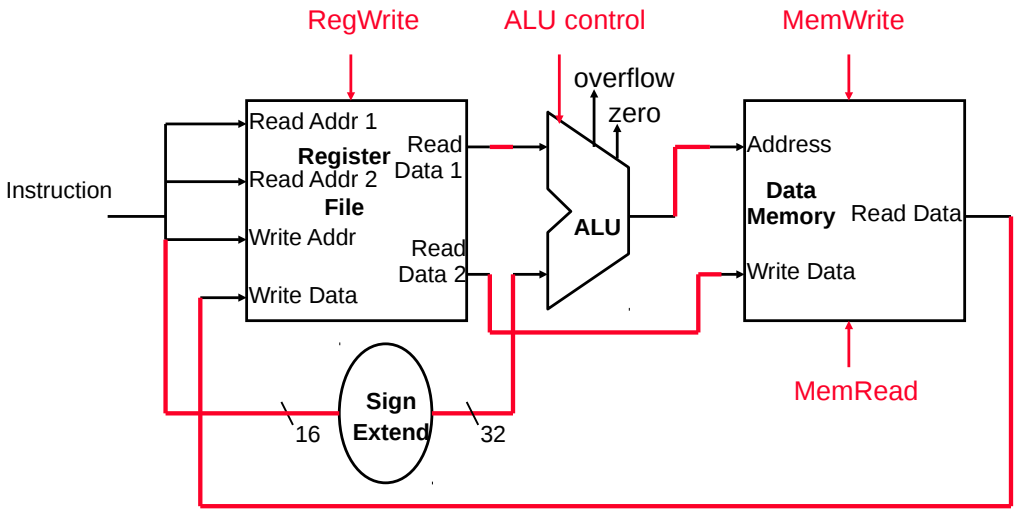


- the Register File is not written every cycle (e.g., `sw`), so we need an explicit write control signal for the Register File

Executing Load and Store Operations

Load and store operations involve:

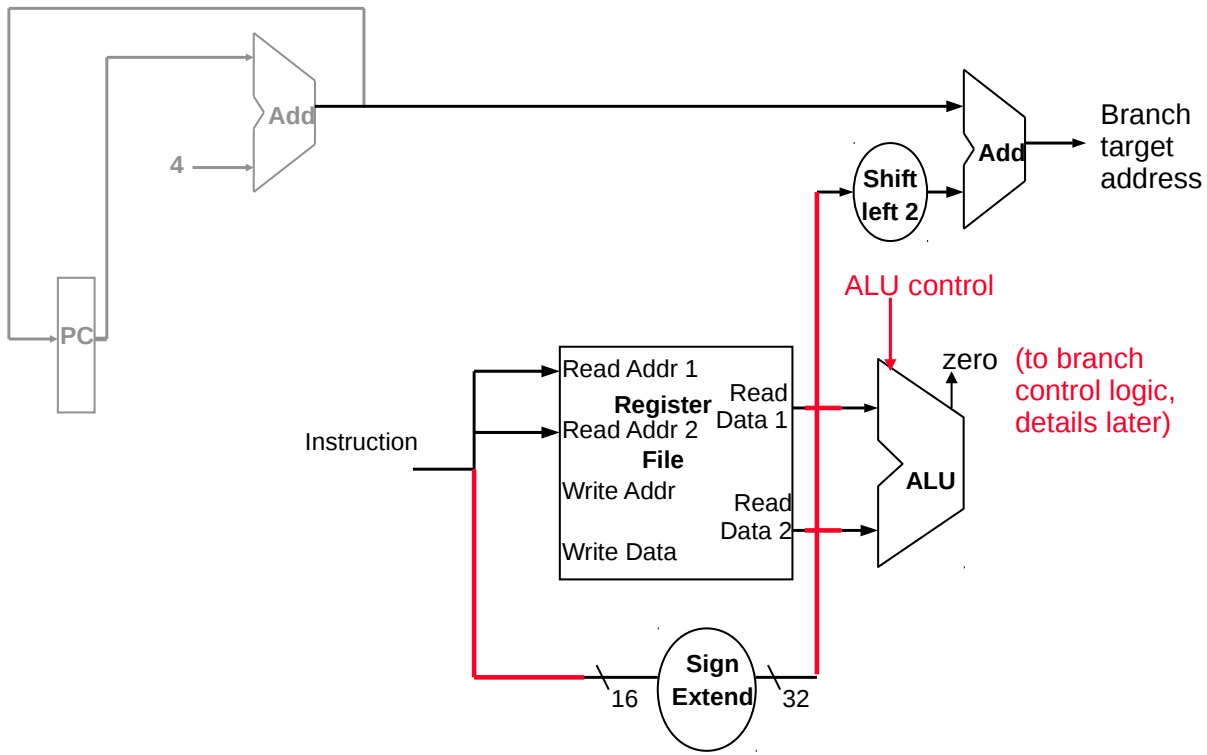
- compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
- store value (read from the Register File during decode) written to the Data Memory
- load value, read from the Data Memory, written to the Register File X



Executing Branch Operations

Branch operations involve:

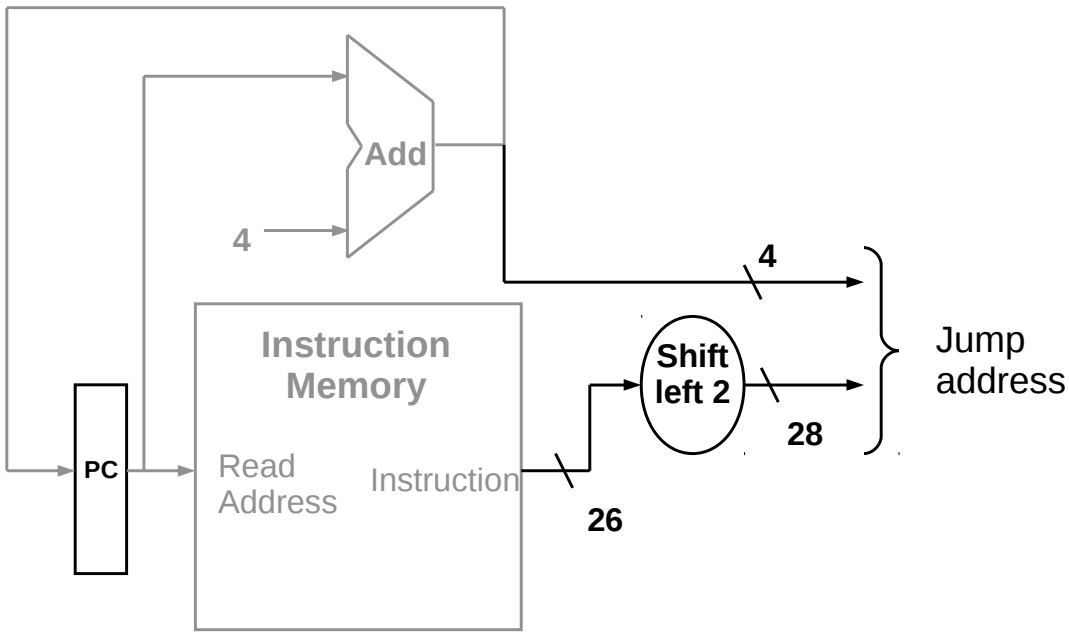
- compare the operands read from the Register File during decode for equality (zero ALU output)
- compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instruction



Executing Jump Instructions

Jump operations involve:

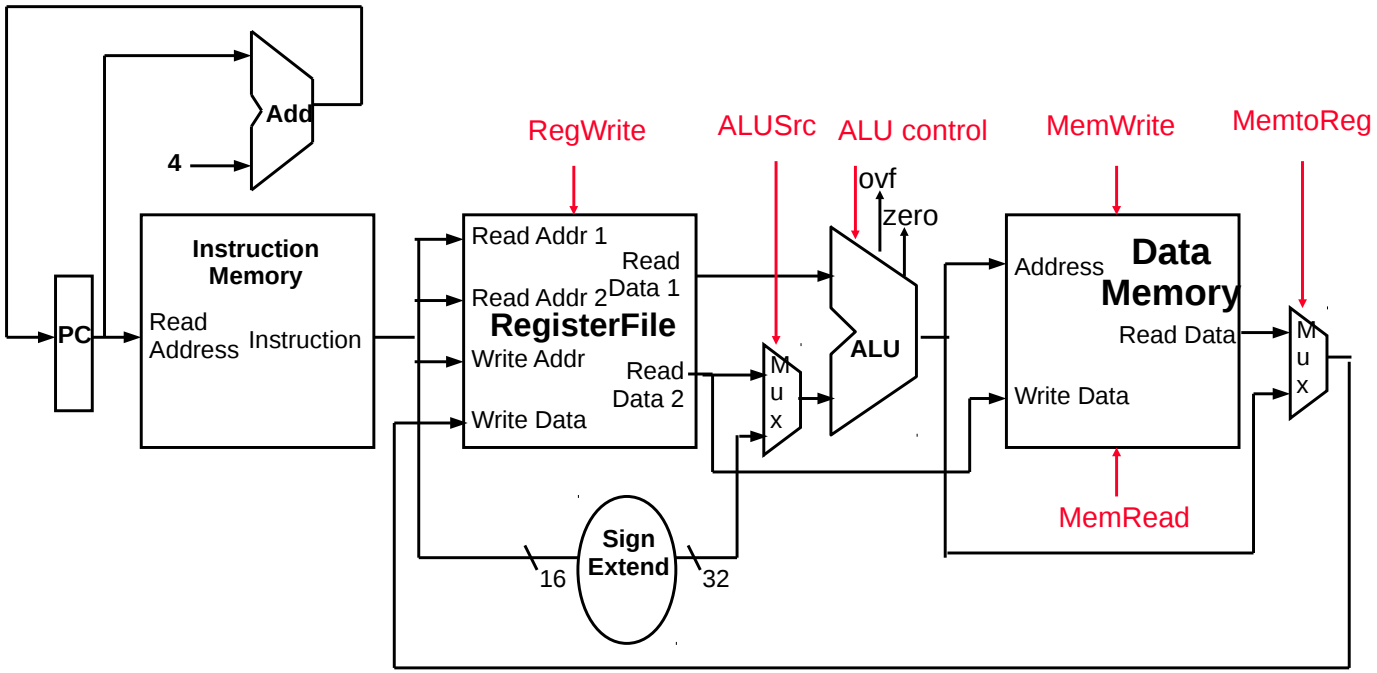
- replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



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 instruction 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)
  - multiplexers 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



- arithmetic-logical instr. use ALU with input from two registers
- memory instr., second input is sign-extended 16-bit offset field
- value in destination register comes either from ALU (R-type) of memory (load)

ALU Control

Four bits used for control inputs:

- Only 6 of 16 bit combinations are used

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

- R-type: AND, OR, subtract or set on less than
- Memory: add
- Branch equal: subtraction

ALU Control Bits

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	save word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

- Opcode determines setting of ALUOp bits
- When ALUOp is 00 or 01, ALU action does not depend on function field
- When ALUOp is 10, function code is used to set ALU control input

Truth Table for the Four ALU Control Bits

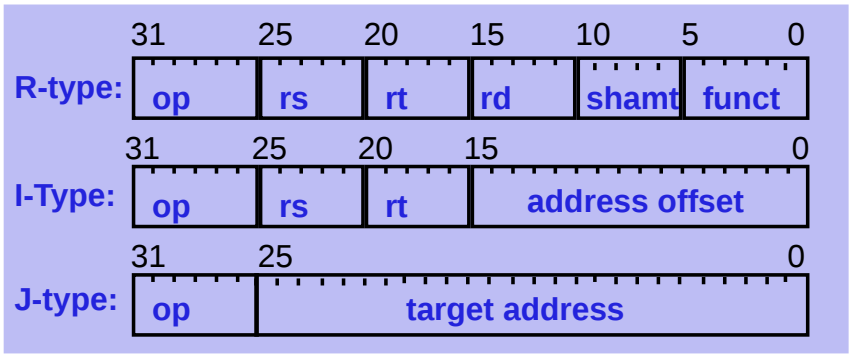
ALUOp		Funct field						Operation
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

- X are "don't care items"
- Such a truth table can be easily turned into gates (therefore encoded in hardware)

Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)
- Observations:

- op field always in bits $31-26$
- addr. of registers to be read are always specified by the `rs` field (bits $25-21$) and `rt` field (bits $20-16$); for `lw` and `sw`, `rs` is the base register
  - addr. of register to be written is in one of two places – in `rt` (bits $20-16$) for `lw`; in `rd` (bits $15-11$) for R-type instructions
  - offset for `beq`, `lw`, and `sw` always in bits $15-0$



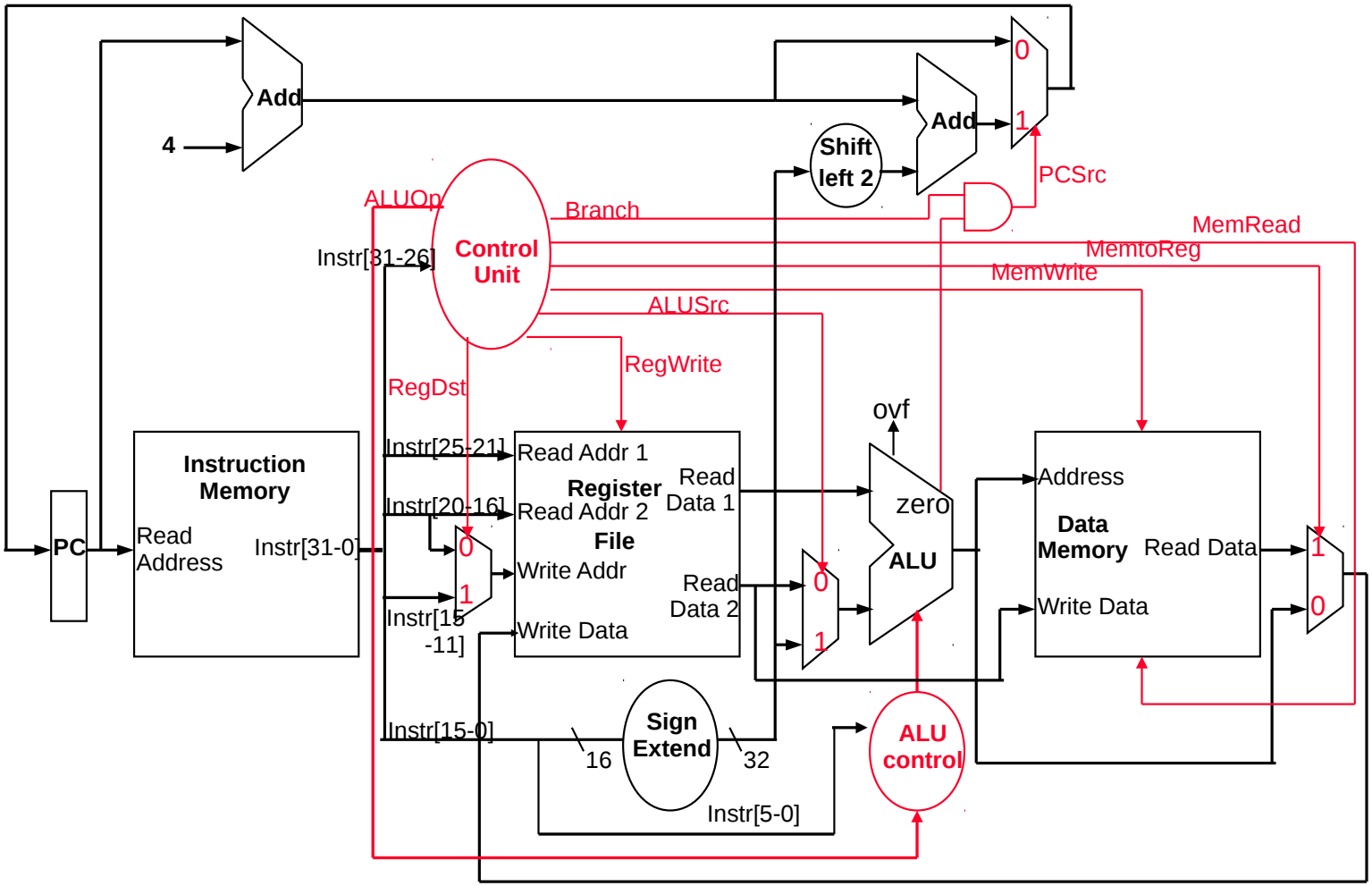
Summary of Control Lines

- RegDest
  Source of the destination register for the operation
- RegWrite
  Enables writing a register in the register file
- ALUsrc
  Source of second ALU operand, can be a register or part of the instruction
- PCsrc
  Source of the PC (increment [PC + 4] or branch)
- MemRead/MemWrite
  Reading/Writing from memory
- MemtoReg
  Source of write register contents

The Seven Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16)	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Single Cycle Datapath with Control Unit

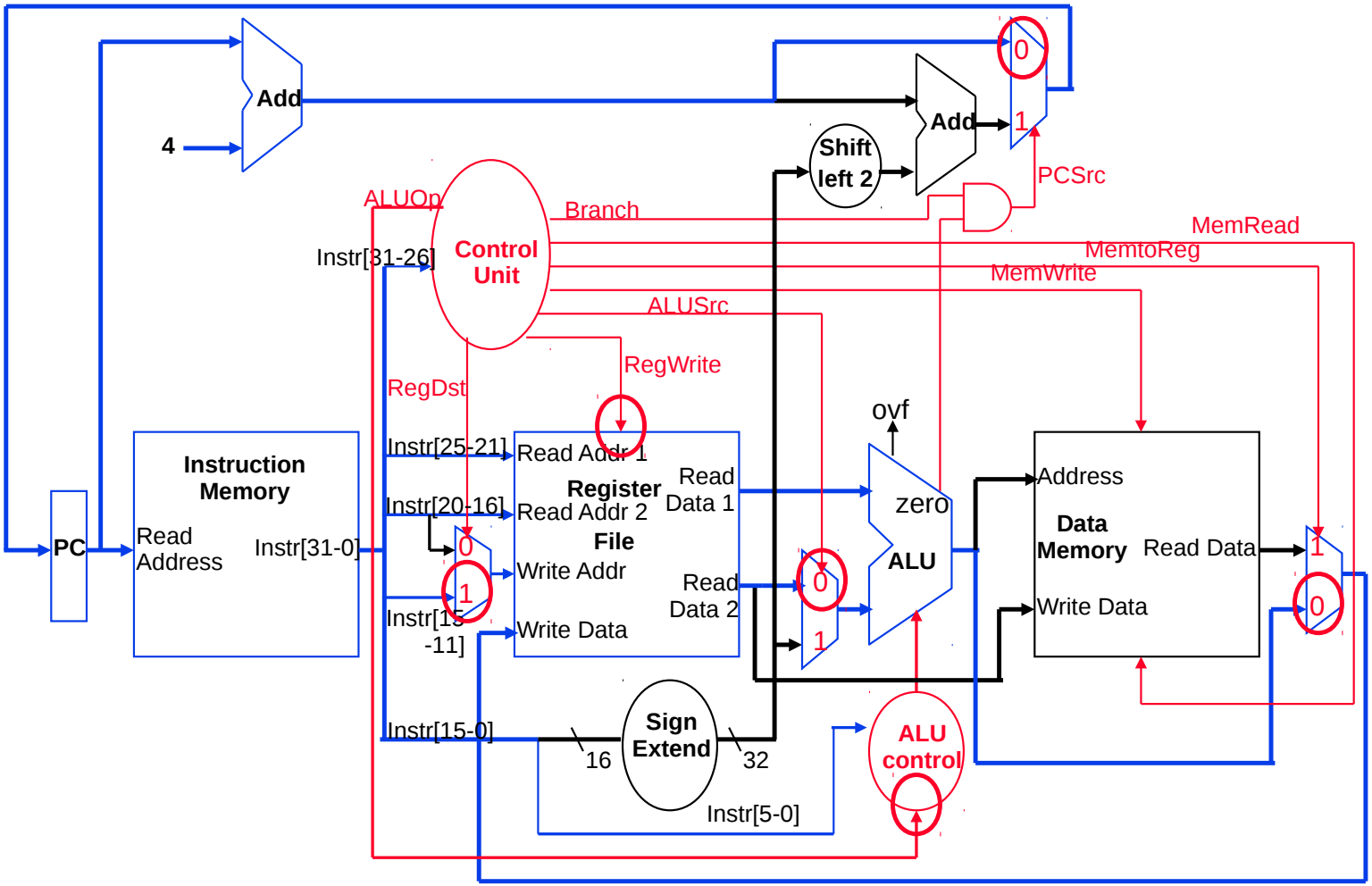


Control Lines

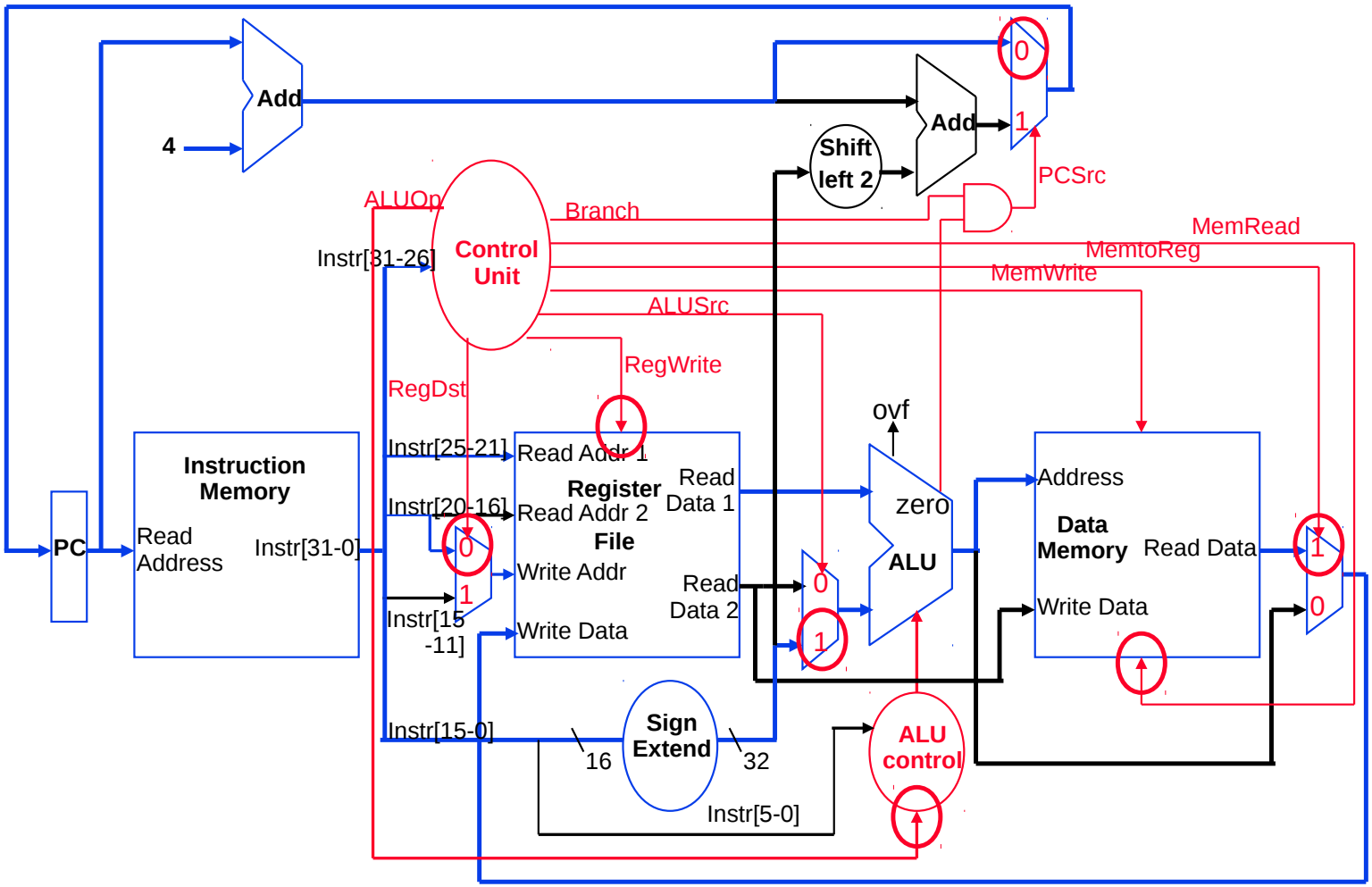
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp2
R-Format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

The setting of the control lines is completely determined by the opcode fields of the instruction

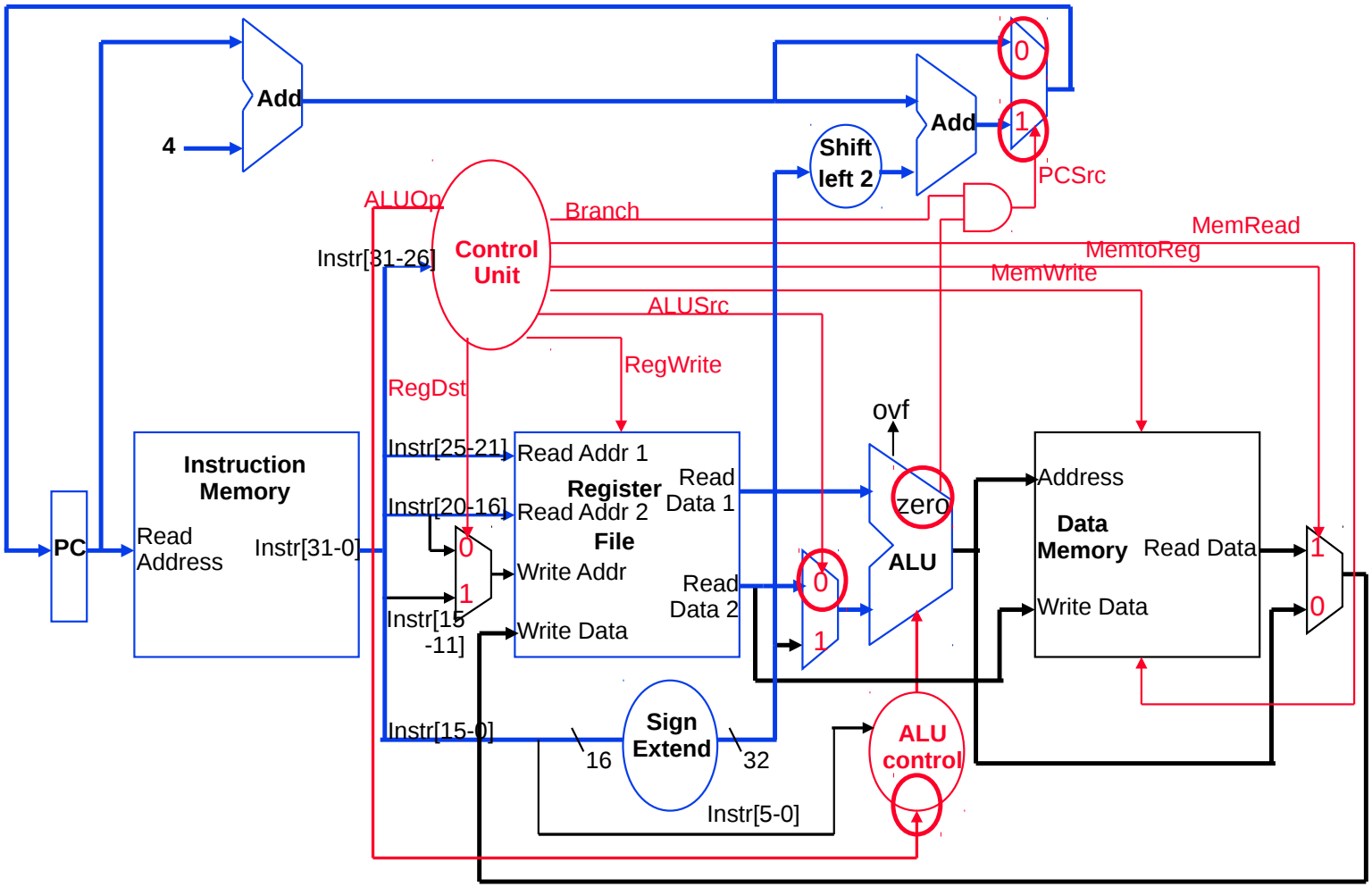
R-type Instruction Data/Control Flow



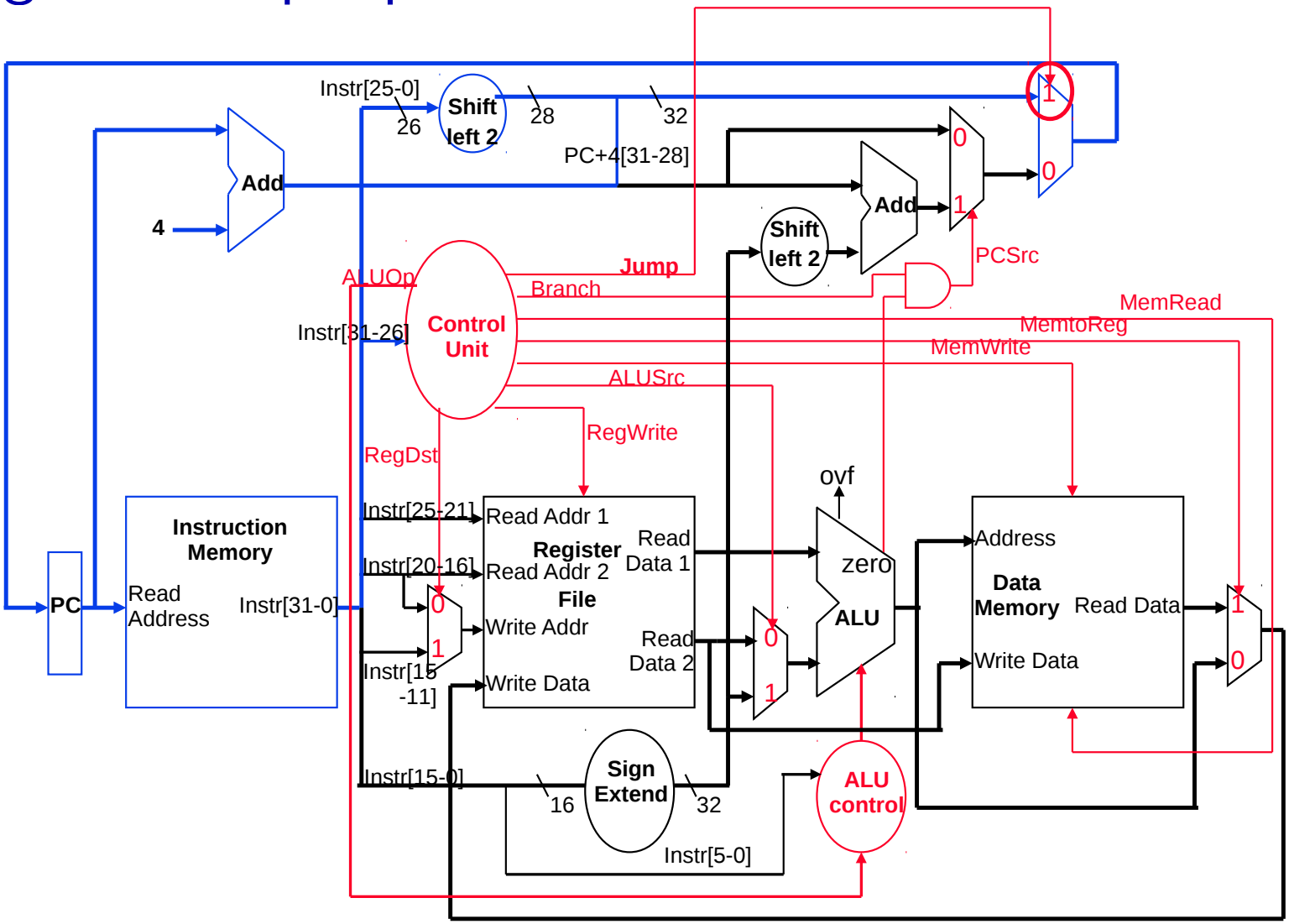
Load Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow



Adding the Jump Operation



Latencies in the Datapath (1)

Latency = a measure of the time delay experienced by a system
First a few notations:

- I-Mem = Instruction Memory
- D-Mem = Data Memory
- ALU = Arithmetic Logic Unit
- Add = Adder
- Mux1, Mux2, Mux3, Mux4 = 4 different multiplexors (which are in the datapath figure)
- Sign-Extent
- Shift-left-2
- RegF = Register File
- WBack = Write Back to the Register File

Latencies in the Datapath (2)

All possible paths for different instruction classes:

- ALU operations like `add`, etc.
- Possible paths for ALU operations:
  1. I-Mem $\rightarrow$ RegF $\rightarrow$ ALU $\rightarrow$ Mux3 $\rightarrow$ WBack to RegF
  2. I-Mem $\rightarrow$ RegF $\rightarrow$ Mux2 $\rightarrow$ ALU $\rightarrow$ Mux3 $\rightarrow$ WBack to RegF
  3. I-Mem $\rightarrow$ Mux1 $\rightarrow$ RegF (Write address)
  4. Add $\rightarrow$ Mux4 $\rightarrow$ PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time

Latencies in the Datapath (3)

- The `sw` instruction:
- Possible paths for `sw`:
  1. I-Mem → RegF → ALU → D-Mem
  2. I-Mem → RegF → D-Mem
  3. I-Mem → Sign-Extend → Mux2 → ALU → D-Mem
  4. Add → Mux4 → PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time

Latencies in the Datapath (4)

- The `lw` instruction:
- Possible paths for `lw`:
  1. I-Mem → RegF → ALU → D-Mem → Mux3 → WBack to RegF
  2. I-Mem → Mux1 → RegF (Write address)
  3. I-Mem → Sign-Extend → Mux2 → ALU → D-Mem → Mux3 → WBack to RegF
  4. Add → Mux4 → PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time

Latencies in the Datapath (5)

- Branching instructions, for example `beq`:
- Possible paths for `beq`:
  1. I-Mem → RegF → ALU → Mux4 → PCWrite (update PC)
  2. I-Mem → RegF → Mux2 → ALU → Mux4 → PCWrite (update PC)
  3. I-Mem → Sign-Extend → Shift-left-2 → Add → Mux4 → PCWrite (update PC)
  4. Add → Mux4 → PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time