CSCE 5610
Computer System Architecture

The Processor

Content

- Single-cycle implementation
  - All operations take the same amount of time—a single cycle.
- Multicycle implementation
  - Allows faster operations to take less time than slower ones, so overall performance can be increased.
- Pipelining
  - Lets a processor overlap the execution of several instructions, potentially leading to big performance gains.

Single-cycle implementation

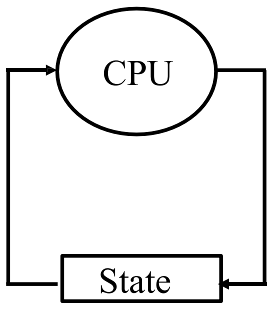
- **We will describe the implementation a simple MIPS-based instruction set supporting just the following operations.**

Arithmetic:	add	sub	and	or	slt
Data Transfer:	lw	sw			
Control:	beq				

- Today we'll build a single-cycle implementation of this instruction set.
  - All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations.
  - We'll explain the datapath first, and then make the control unit.

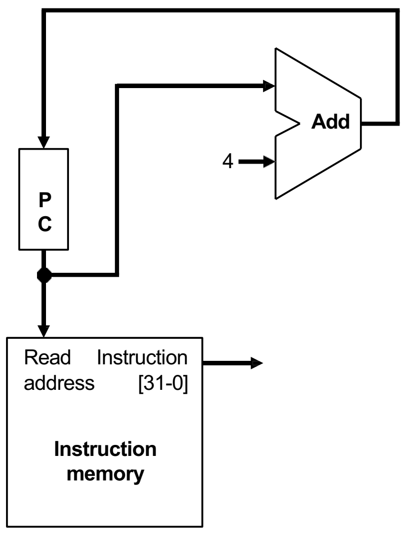
Computers are state machines

- A computer is just a big fancy state machine.
  - Registers, memory, hard disks and other storage form the state.
  - The processor keeps reading and updating the state, according to the instructions in some program.



Instruction fetching

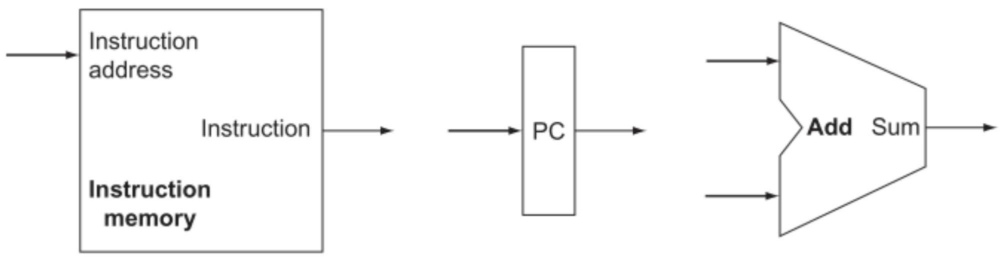
- The CPU is always in an infinite loop, fetching instructions from memory and executing them.
- The program counter or PC register holds the address of the current instruction.



Basic MIPS implementation

The first two steps for every instruction:

- Send PC to instruction memory
- Read one or two registers using fields in the machine code

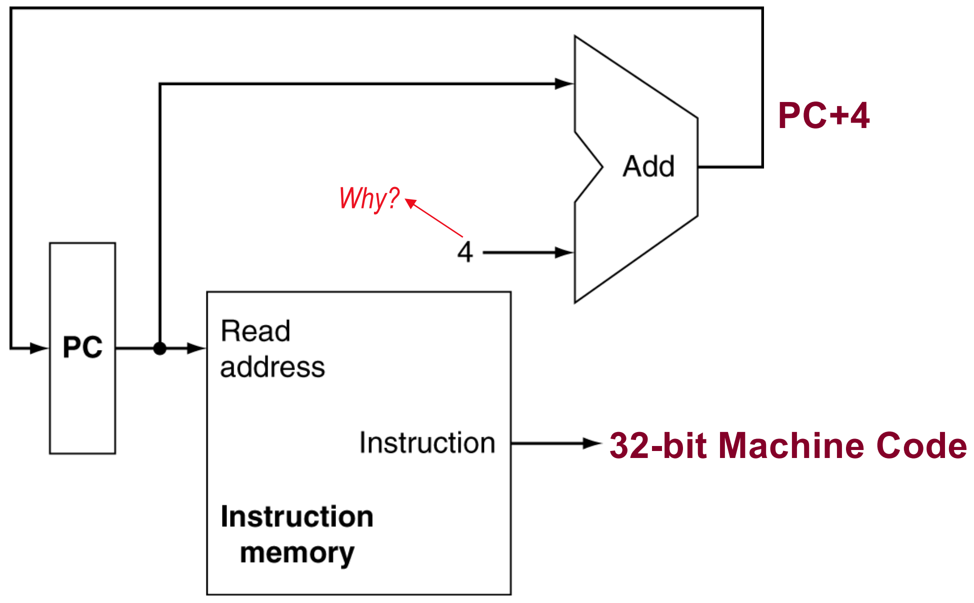


- Instruction Memory: stores the code and supply instruction given an address
- Program Counter (PC): holds the address of the current instruction
- Adder: increments the PC to the address of the next instruction

Building a datapath: Fetch

To execute any instruction:

1. ***Fetch*** the instruction from memory
2. Increment the PC to point at the next instruction



Encoding R-type instructions

- Last lecture, we saw encodings of MIPS instructions as 32-bit values.
- Register-to-register arithmetic instructions use the R-type format.
  - op is the instruction opcode, and func specifies a particular arithmetic operation.
  - rs, rt and rd are source and destination registers.

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

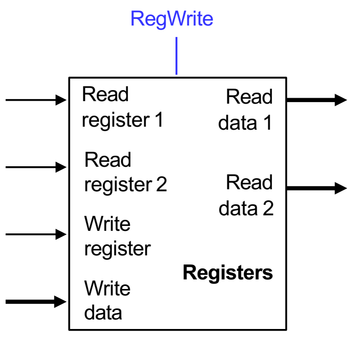
- An example instruction and its encoding:

add $20, $9, $10

000000	01001	01010	10100	00000	1000000

Registers and ALUs

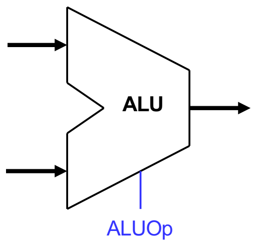
- R-type instructions must access registers and an ALU.



- Our register file stores thirty-two 32-bit values.
  - Each register specifier is 5 bits long.
  - You can read from two registers at a time.
  - RegWrite is 1 if a register should be written.

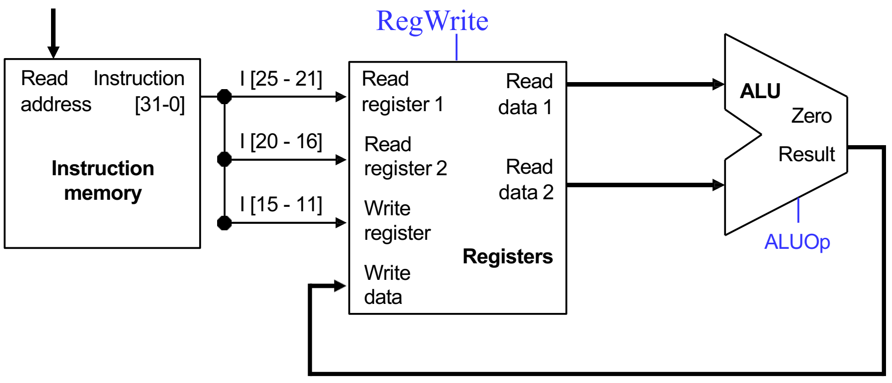
- Here's a simple ALU with five operations, selected by a 3-bit control signal ALUOp.

ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt



Executing an R-type instruction

1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields rs and rt, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field rd of the instruction word.



op	rs	rt	rd	shamt	func
31 26	25 21	20 16	15 11	10 6	5 0

Encoding I-type instructions

- The lw, sw and beq instructions all use the I-type encoding.
  - rt is the *destination* for lw, but a *source* for beq and sw.
  - address is a 16-bit signed constant.

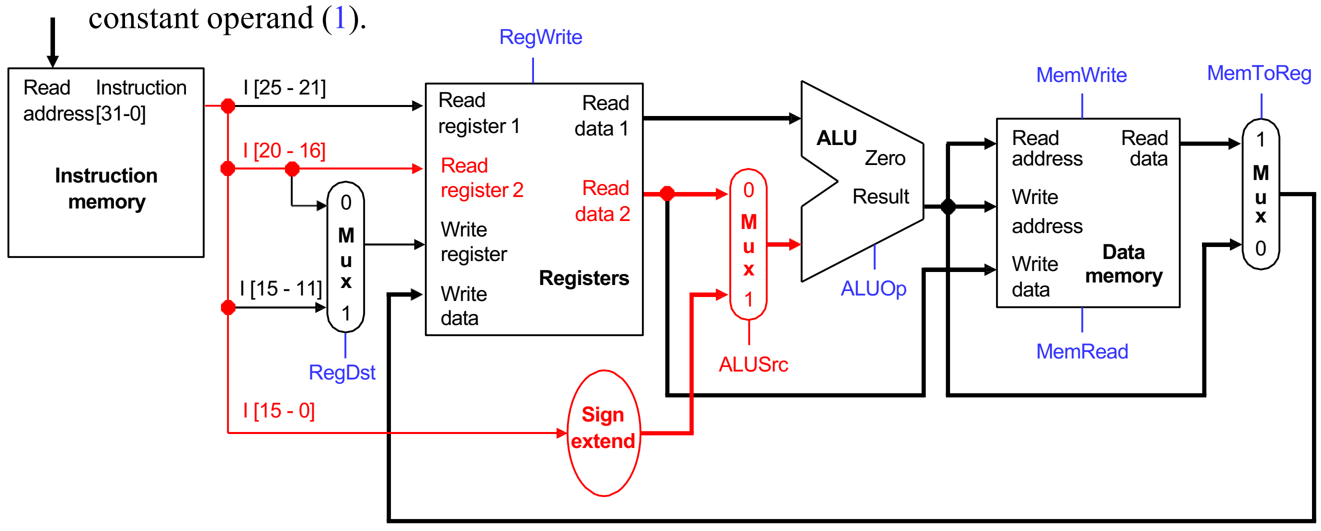
op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- Two example instructions:

lw $t0, –4($sp)	100011	11101	01000	1111 1111 1111 1100
sw $a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000

Accessing data memory

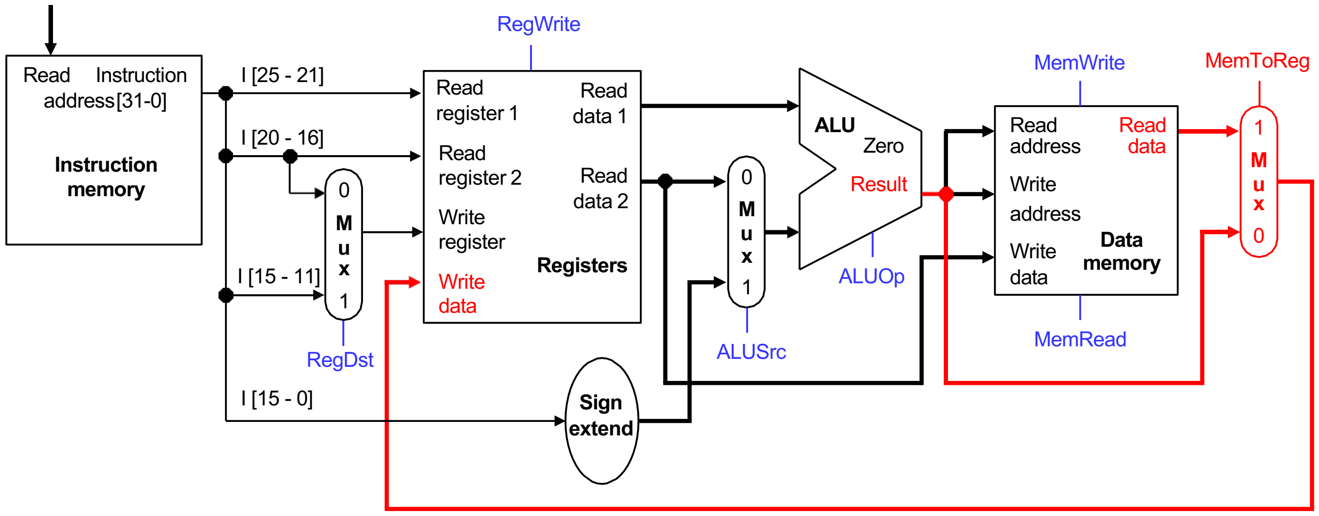
- For an instruction like lw $t0, –4($sp), the base register $sp is added to the *sign-extended* constant to get a data memory address.
- This means the ALU must accept *either* a register operand for arithmetic instructions, *or* a sign-extended immediate operand for lw and sw.
- We'll add a multiplexer, controlled by ALUSrc, to select either a register operand (0) or a constant operand (1).



lw $t0, -4($sp)	100011	11101	01000	1111 1111 1111 1100

MemToReg

- The register file's — "Write data" input has a similar problem. It must be able to store *either* the ALU output of R-type instructions, *or* the data memory output for lw.
- We add a mux, controlled by MemToReg, to select between saving the ALU result (0) or the data memory output (1) to the registers.



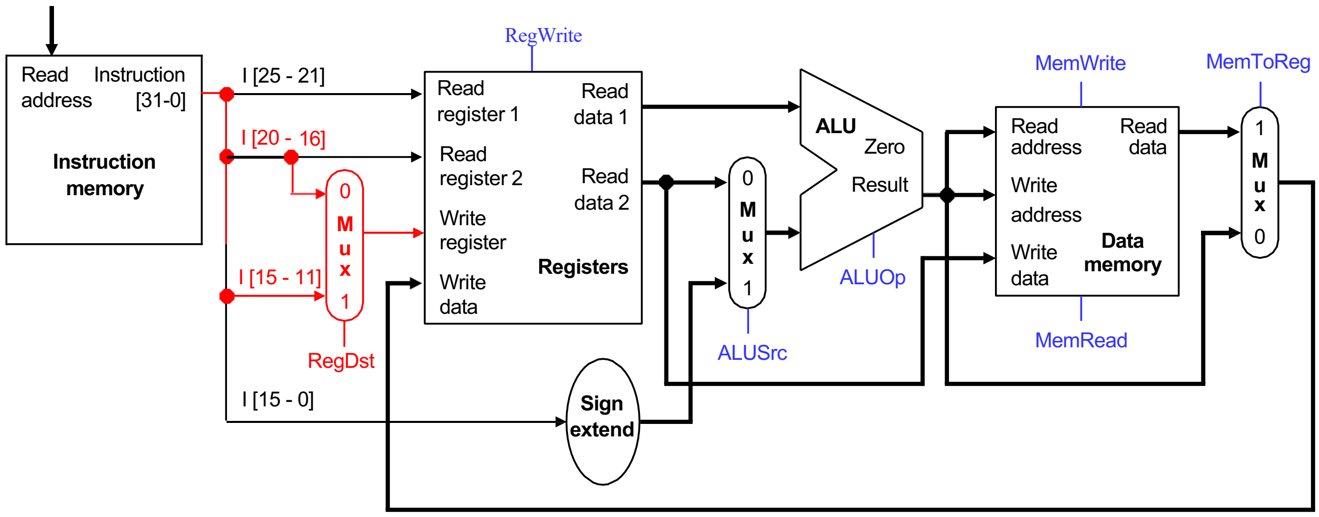
RegDst

- A final annoyance is the destination register of lw is in *rt* instead of rd.

op	rs	rt	address

lw $rt, address($rs)

- We'll add one more mux, controlled by RegDst, to select the destination register from either instruction field rt (0) or field rd (1).



Branches

- For branch instructions, the constant is not an address but an *instruction offset* from the current program counter to the desired address.

```
        beq   $at, $0, L
        add   $v1, $v0, $0
        add   $v1, $v1, $v1
        j     Somewhere
L:      add   $v1, $v0, $v0
```

- The target address L is three *instructions* past the beq, so the encoding of the branch instruction has 0000 0000 0000 0011 for the address field.

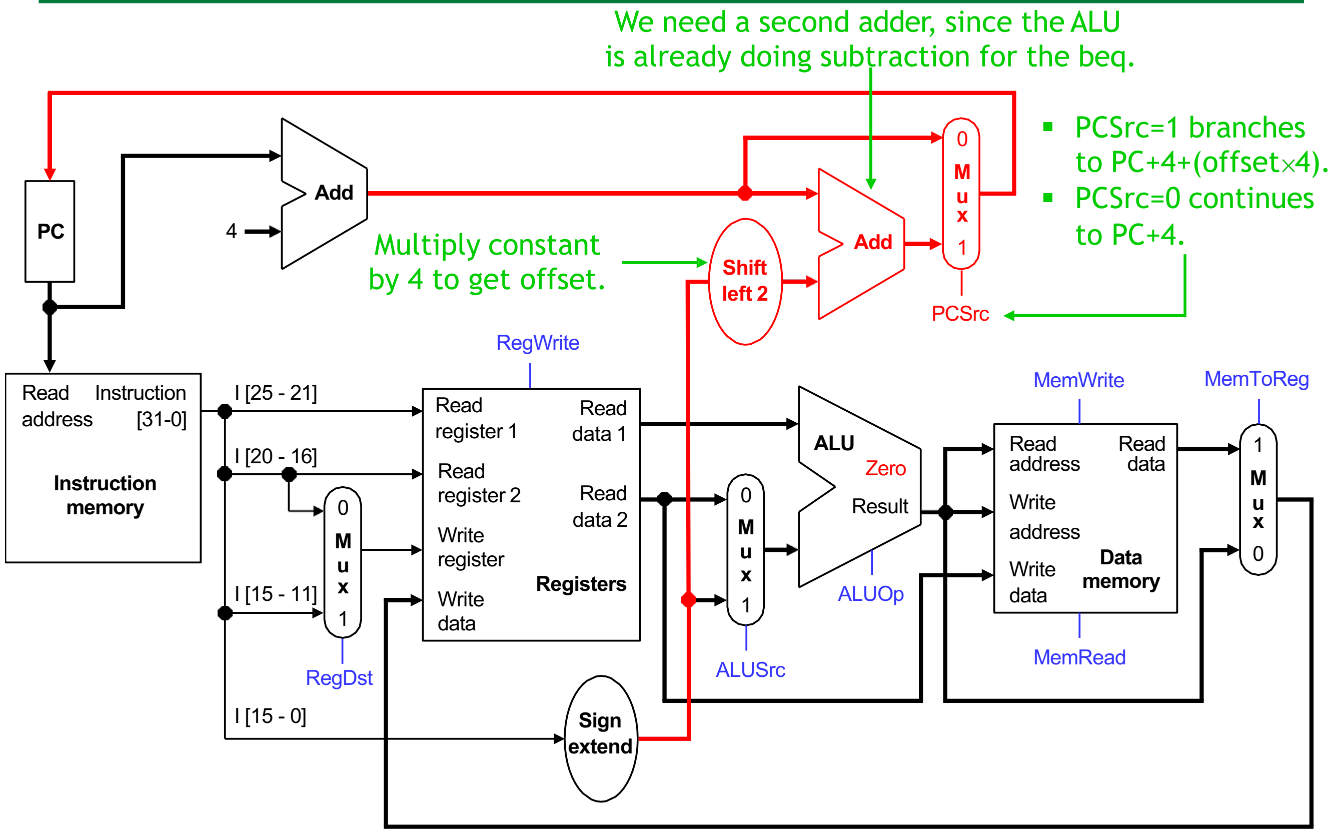
000100	00001	00000	0000 0000 0000 0011
op	rs	rt	address

- Instructions are four bytes long, so the actual memory offset is 12 bytes.

The steps in executing a beq

1. Fetch the instruction, like beq $at, $0, offset, from memory.
2. Read the source registers, $at and $0, from the register file.
3. Compare the values by subtracting them in the ALU.
4. If the subtraction result is 0, the source operands were equal and the PC should be loaded with the target address, PC + 4 + (offset x 4).
5. Otherwise the branch should not be taken, and the PC should just be incremented to PC + 4 to fetch the next instruction sequentially.

Branching hardware

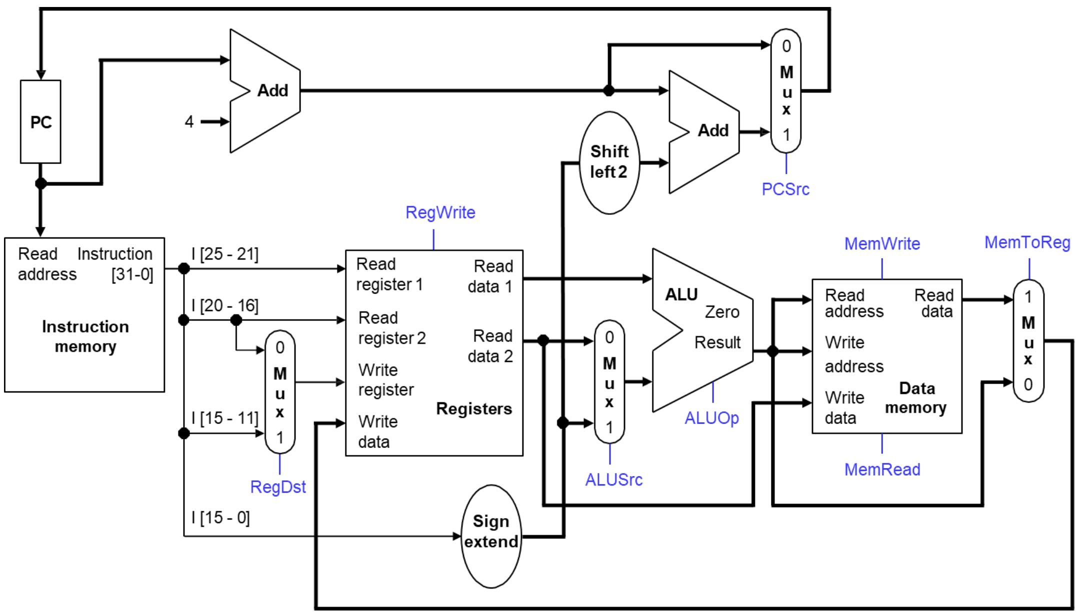


Control

- The control unit is responsible for setting all the control signals so that each instruction is executed properly.
  - The control unit's input is the 32-bit instruction word.
  - The outputs are values for the blue control signals in the datapath.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.
- To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.

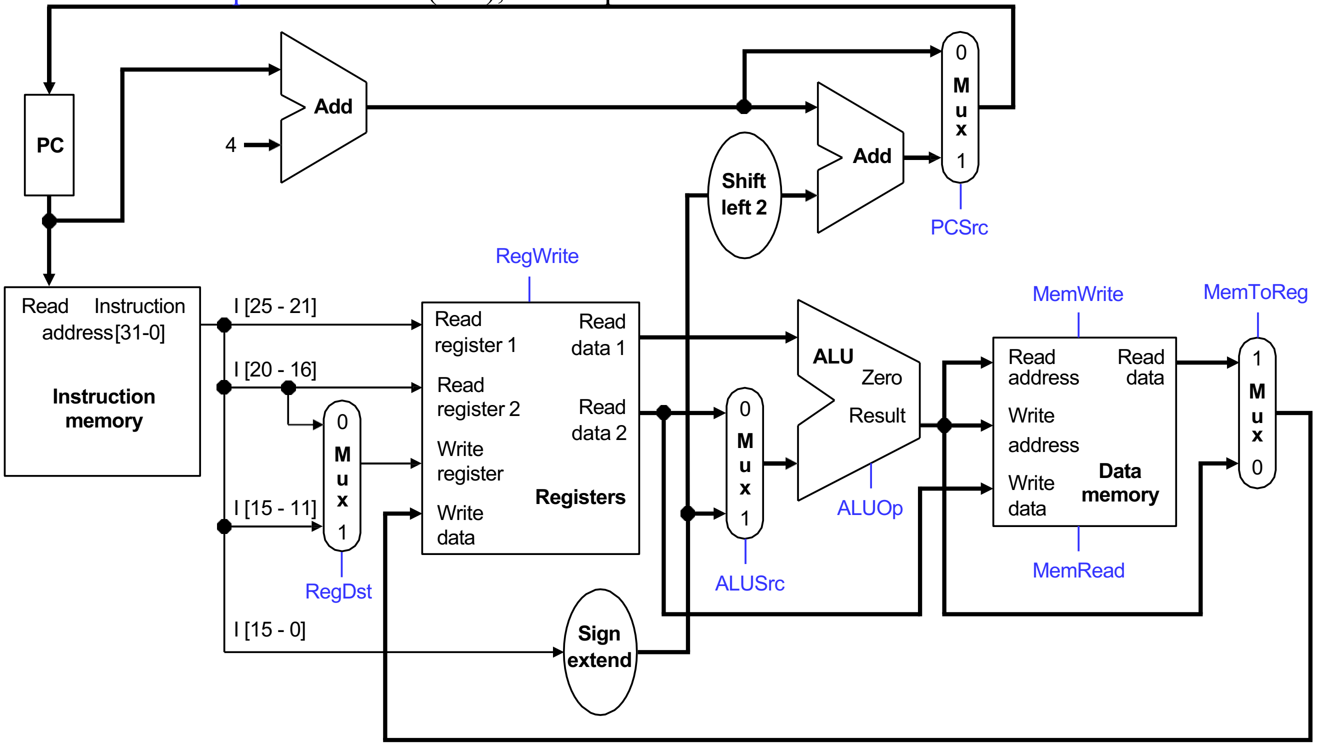
R-type instruction path

- The R-type instructions include add, sub, and, or, and slt.
- The ALUOp is determined by the instruction's —"func" field.



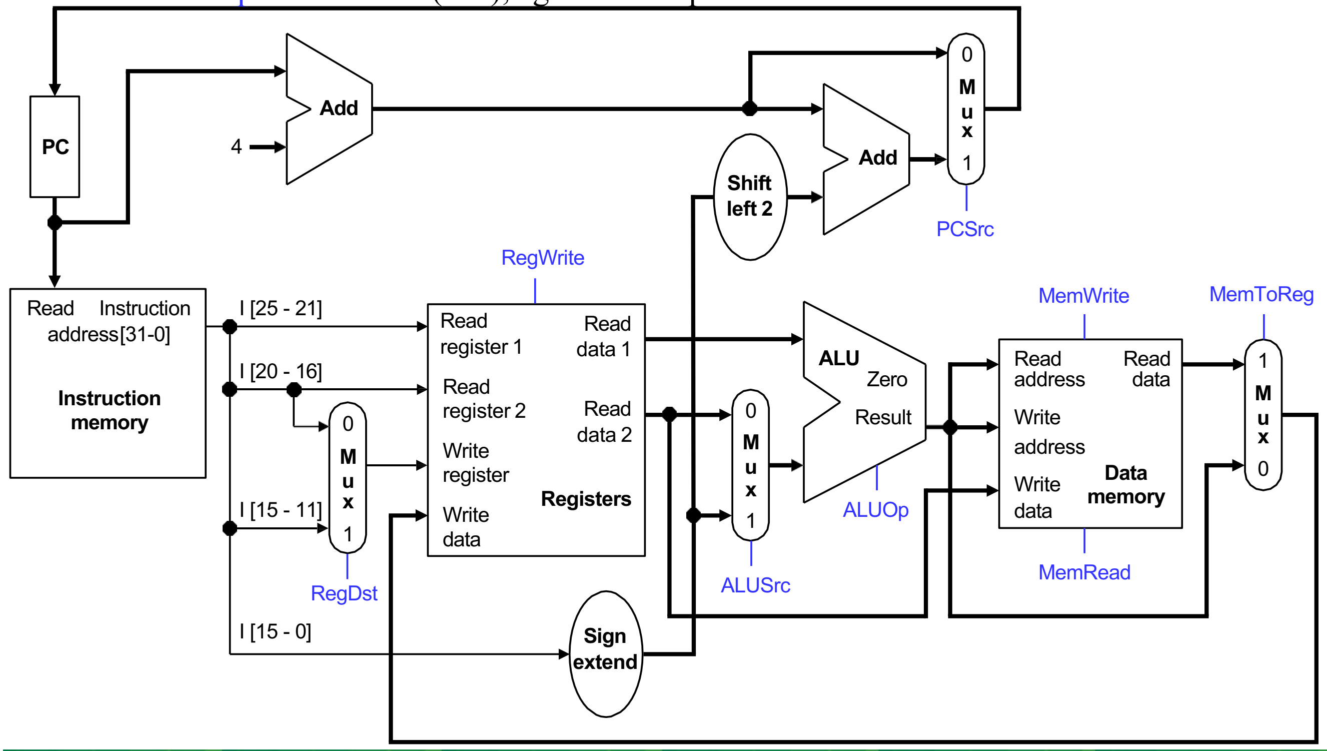
lw instruction path

- An example load instruction is lw $t0, –4($sp).
- The ALUOp must be 010 (add), to compute the effective address.



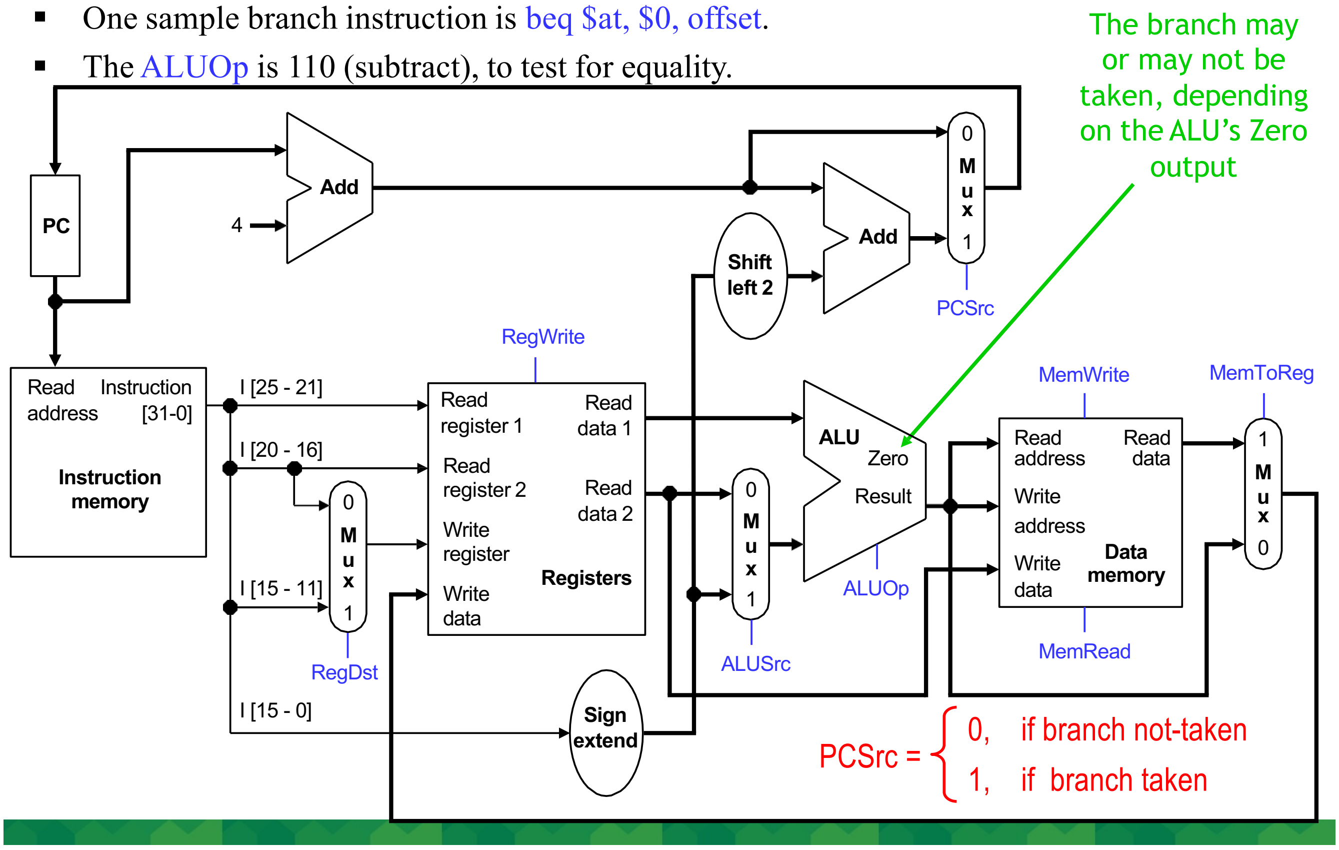
sw instruction path

- An example store instruction is sw $a0, 16($sp).
- The ALUOp must be 010 (add), again to compute the effective address.



beq instruction path

- One sample branch instruction is beq $at, $0, offset.
- The ALUOp is 110 (subtract), to test for equality.



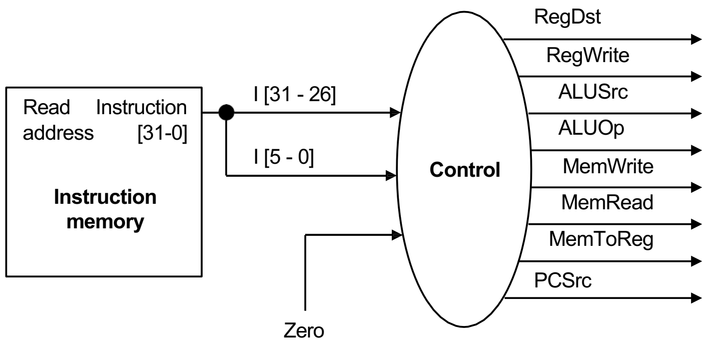
Control Signal Table

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUOp for R-type instructions depends on the instructions' func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq *and* the ALU's Zero output is true.

Generating Control Signals

- The control unit needs 13 bits of inputs.
  - — Six bits make up the instruction's opcode.
  - — Six bits come from the instruction's func field.
  - — It also needs the Zero output of the ALU.
- The control unit generates 10 bits of output, corresponding to the signals mentioned on the previous page.

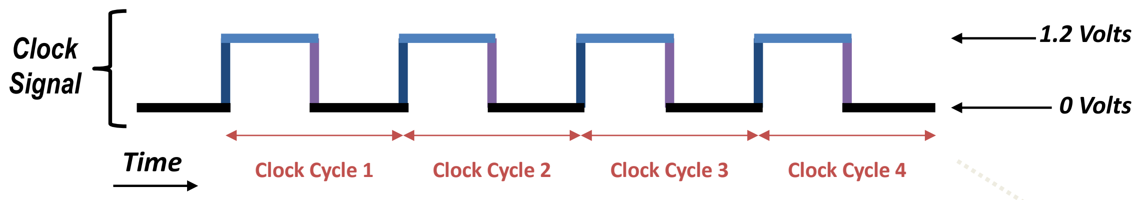


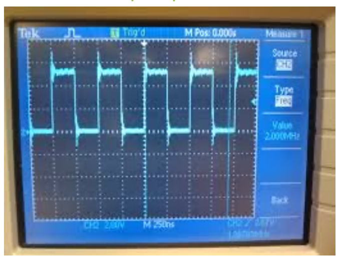
Single-Cycle Performance

- We saw a MIPS single-cycle datapath and control unit.
- Then, we'll explore factors that contribute to a processor's execution time, and specifically at the performance of the single-cycle machine.
- Next, we'll explore how to improve on the single cycle machine's performance using pipelining.

Processor clock period

the processor clock oscillates between **high** and **low** signal levels:





the clock synchronizes the processor's operations on **rising** & **falling** edges:

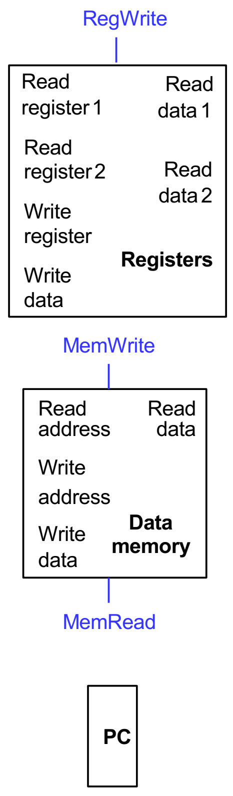
- a register updates contents upon arrival of clock edge
- a processor **starts instruction execution** at clock edge

Clock Period = **Clock Cycle Time**

- interval of time between 2 adjacent rising edges
- duration of 1 cycle of the clock signal
- clock period has units of seconds (nanoseconds, microseconds, or picoseconds)

Edge-triggered State Elements

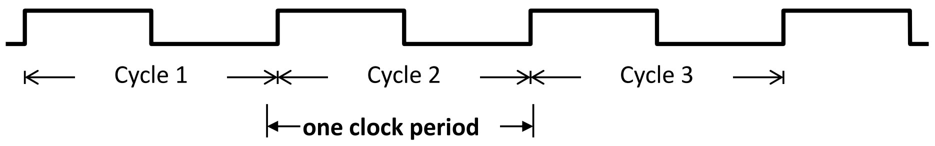
- In an instruction like add \$t1, \$t1, \$t2, how do we know \$t1 is not updated until after its original value is read?
- We'll assume that our state elements are positive edge triggered, and are updated only on the positive edge of a clock signal.
  - The register file and data memory have explicit write control signals, RegWrite and MemWrite. These units can be written to only if the control signal is asserted and there is a positive clock edge.
  - In a single-cycle machine the PC is updated on each clock cycle, so we don't bother to give it an explicit write control signal.



Processor Frequency

Clock Frequency = 1 / (Clock Period)

a *CPU's Clock Frequency is the* ***inverse*** *of it's Clock Cycle time*



- the number of clock cycles occurring in 1 second
- measured in units of **Hertz (Hz)**
- also called the **Clock Rate**

$\boxed{F = 1 / \text{Period}}$

10 nsec clock cycle period → 100 MHz clock rate
5 nsec clock cycle period → 200 MHz clock rate
2 nsec clock cycle period → 500 MHz clock rate
1 nsec (10^{-9}) clock cycle period → 1 GHz (10^{9}) clock rate
500 psec clock cycle period → 2 GHz clock rate
250 psec clock cycle period → 4 GHz clock rate
200 psec clock cycle period → 5 GHz clock rate

When designing processors, we work in units of:
Nanoseconds
or
GHz

CPU Time

$$CPU\ time = CPU\ Clock\ Cycles * Cycle\ Time = \frac{Clock\ Cycles}{Clock\ Rate}$$

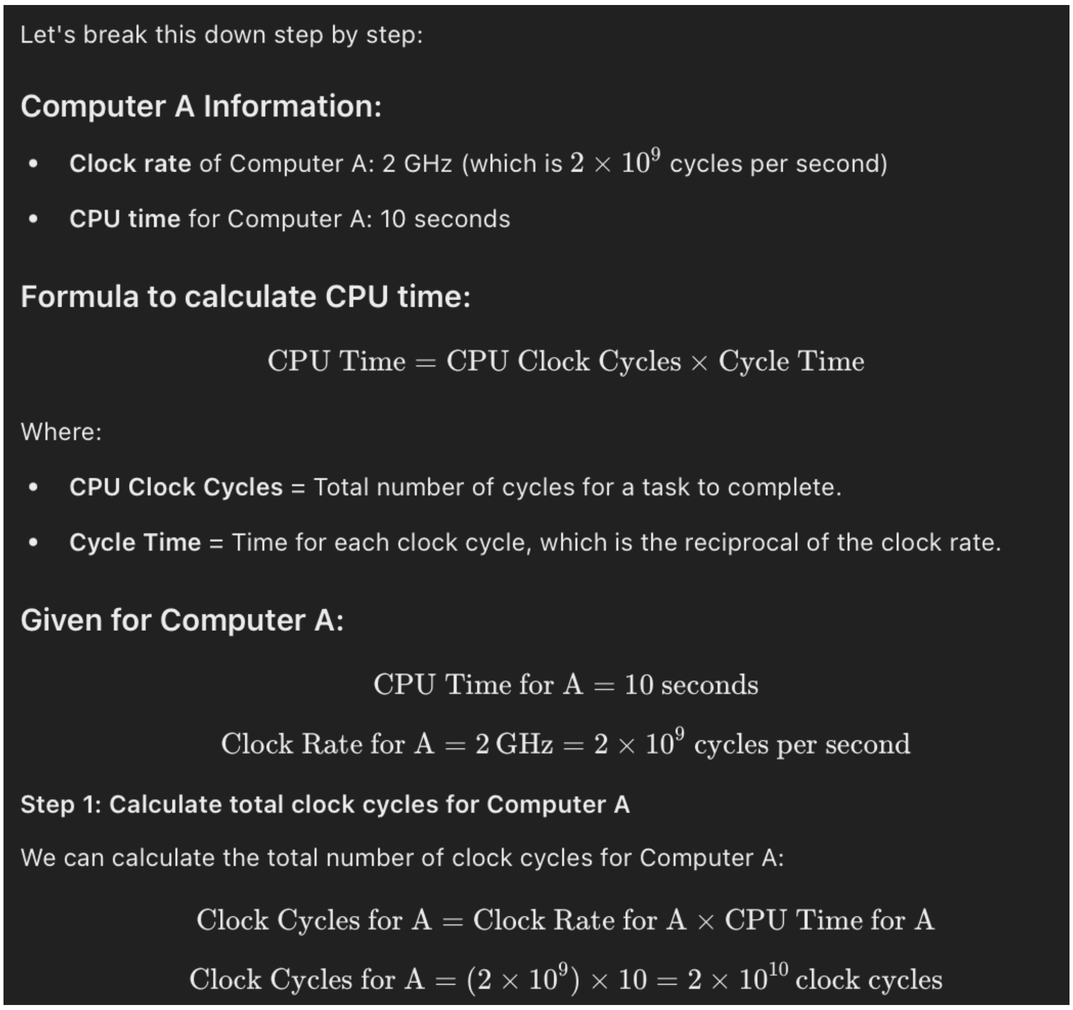
- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Exercise

$$CPU\ time = CPU\ Clock\ Cycles * Cycle\ Time = \frac{Clock\ Cycles}{Clock\ Rate}$$

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2× clock cycles
- How fast must Computer B clock be?

$$\frac{Clock\ Rate_B}{Clock\ Rate_A} = ?$$



Let's break this down step by step:

Computer A Information:

- **Clock rate** of Computer A: 2 GHz (which is 2×10^9 cycles per second)
- **CPU time** for Computer A: 10 seconds

Formula to calculate CPU time:

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Cycle Time}$$

Where:

- **CPU Clock Cycles** = Total number of cycles for a task to complete.
- **Cycle Time** = Time for each clock cycle, which is the reciprocal of the clock rate.

Given for Computer A:

$$\text{CPU Time for A} = 10 \text{ seconds}$$

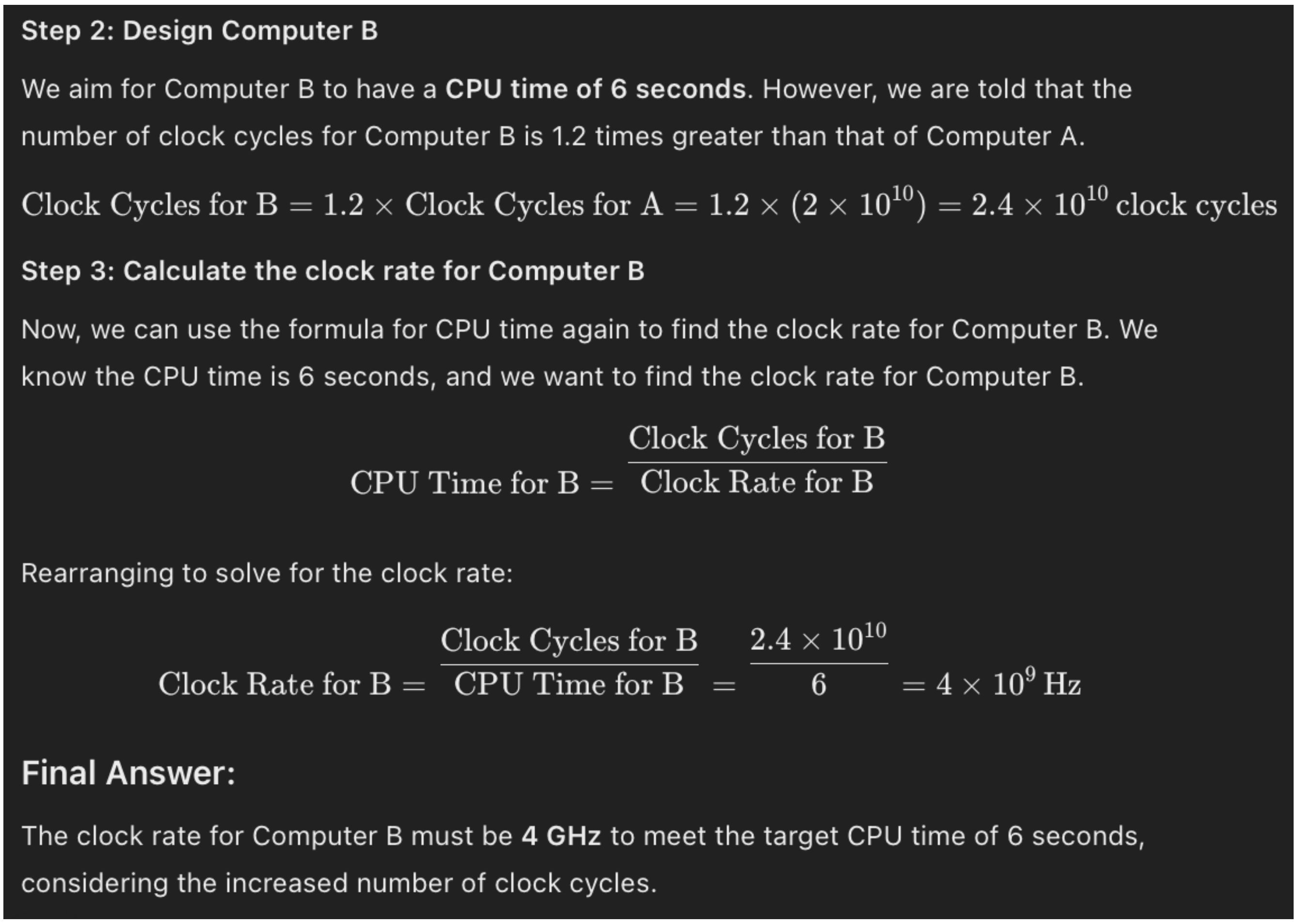
$$\text{Clock Rate for A} = 2\,\text{GHz} = 2 \times 10^9 \text{ cycles per second}$$

Step 1: Calculate total clock cycles for Computer A

We can calculate the total number of clock cycles for Computer A:

$$\text{Clock Cycles for A} = \text{Clock Rate for A} \times \text{CPU Time for A}$$

$$\text{Clock Cycles for A} = (2 \times 10^9) \times 10 = 2 \times 10^{10} \text{ clock cycles}$$



Step 2: Design Computer B

We aim for Computer B to have a **CPU time of 6 seconds**. However, we are told that the number of clock cycles for Computer B is 1.2 times greater than that of Computer A.

$$\text{Clock Cycles for B} = 1.2 \times \text{Clock Cycles for A} = 1.2 \times (2 \times 10^{10}) = 2.4 \times 10^{10} \text{ clock cycles}$$

Step 3: Calculate the clock rate for Computer B

Now, we can use the formula for CPU time again to find the clock rate for Computer B. We know the CPU time is 6 seconds, and we want to find the clock rate for Computer B.

$$\text{CPU Time for B} = \frac{\text{Clock Cycles for B}}{\text{Clock Rate for B}}$$

Rearranging to solve for the clock rate:

$$\text{Clock Rate for B} = \frac{\text{Clock Cycles for B}}{\text{CPU Time for B}} = \frac{2.4 \times 10^{10}}{6} = 4 \times 10^9 \,\text{Hz}$$

Final Answer:

The clock rate for Computer B must be **4 GHz** to meet the target CPU time of 6 seconds, considering the increased number of clock cycles.

CPI-Cycles per Instruction

- The average number of clock cycles per instruction, or CPI, is a function of the machine and program.
  - The CPI depends on the actual instructions appearing in the program— a floating-point intensive application might have a higher CPI than an integer-based program.
  - It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster.
- We assumed each instruction took one cycle, so we had CPI = 1.
  - The CPI can be >1 due to memory stalls and slow instructions.
  - The CPI can be <1 on machines that execute more than 1 instruction per cycle (superscalar).

CPU Time

$$\boxed{CPU\ time = Instructions\ executed * CPI * Cycle\ Time}$$

Instructions Executed

- Instructions executed:
  - We are not interested in the static instruction count, or how many lines of code are in a program.
  - Instead we care about the **dynamic instruction count**, or how many instructions are actually executed when the program runs.

```
            li    $4,   1000
Ostrich:    subi  $4,   $4, 1
            bne   $4,   $0, Ostrich
```

Instructions Executed

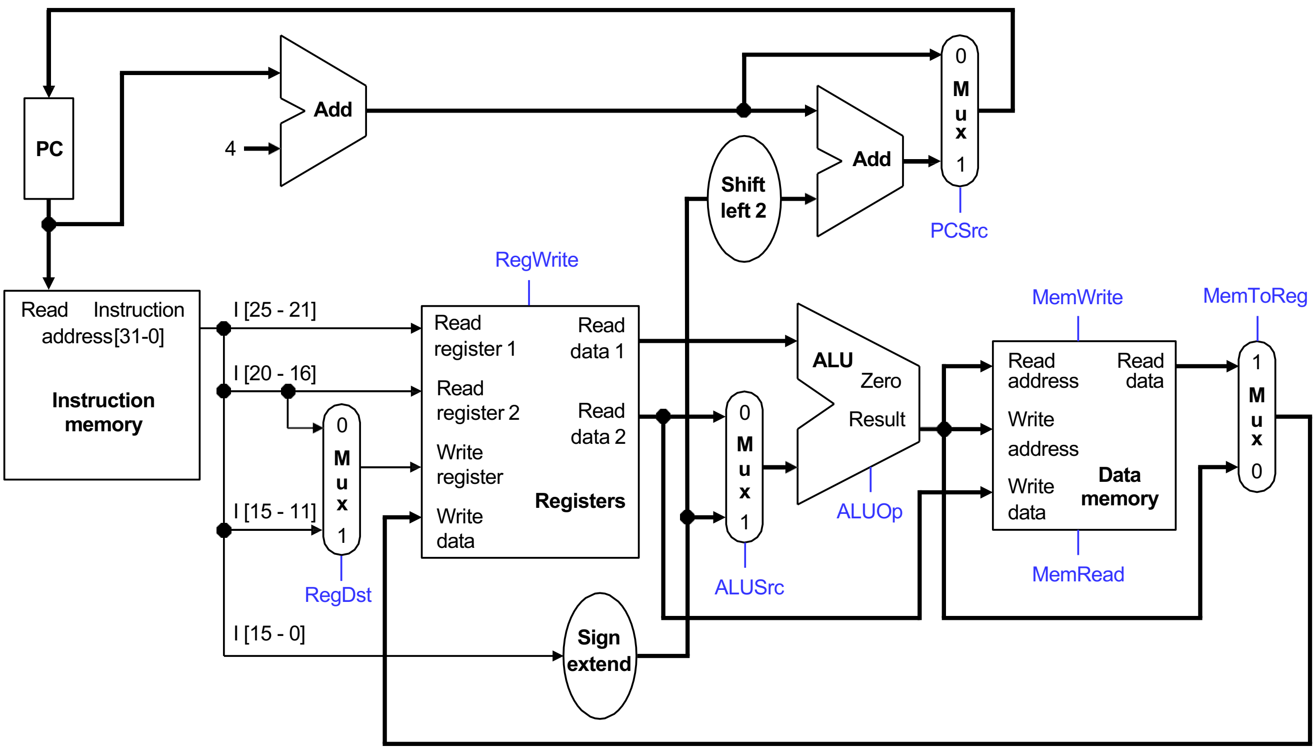
- Instructions executed:
  - We are not interested in the static instruction count, or how many lines of code are in a program.
  - Instead we care about the **dynamic instruction count**, or how many instructions are actually executed when the program runs.
  - There are three lines of code below, but the number of instructions executed would be 2001.

```
            li    $4,   1000
Ostrich:    subi  $4,   $4, 1
            bne   $4,   $0, Ostrich
```

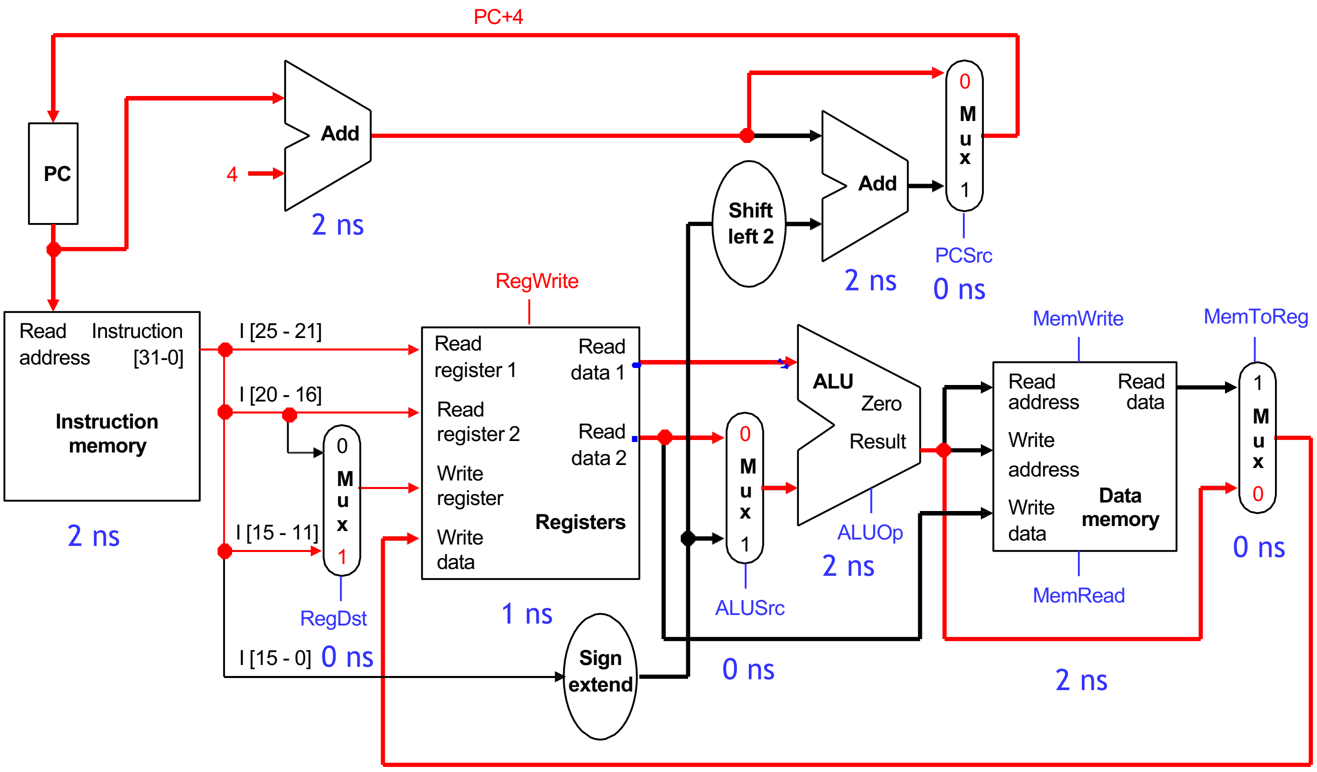
Clock Cycle Time

- One "cycle" is the minimum time it takes the CPU to do any work.
  - — The clock cycle time or clock period is just the length of a cycle.
  - — The clock rate, or frequency, is the reciprocal of the cycle time.
- Generally, a higher frequency is better.
- Some examples illustrate some typical frequencies.
  - — A 500MHz processor has a cycle time of 2ns.
  - — A 2GHz (2000MHz) CPU has a cycle time of just 0.5ns.

How does add go through the datapath

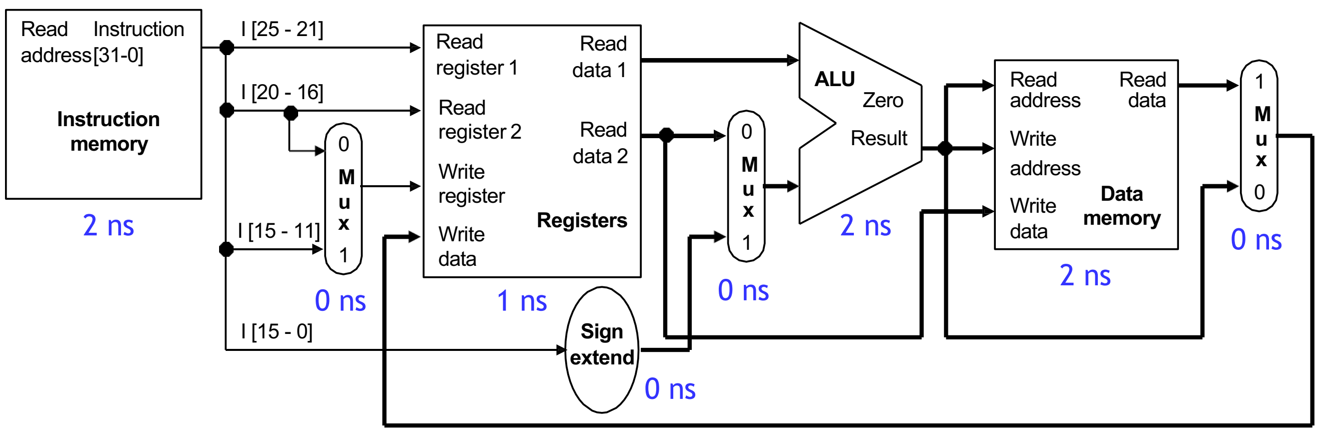


Compute the longest path in the add instruction



The Slowest Instruction...

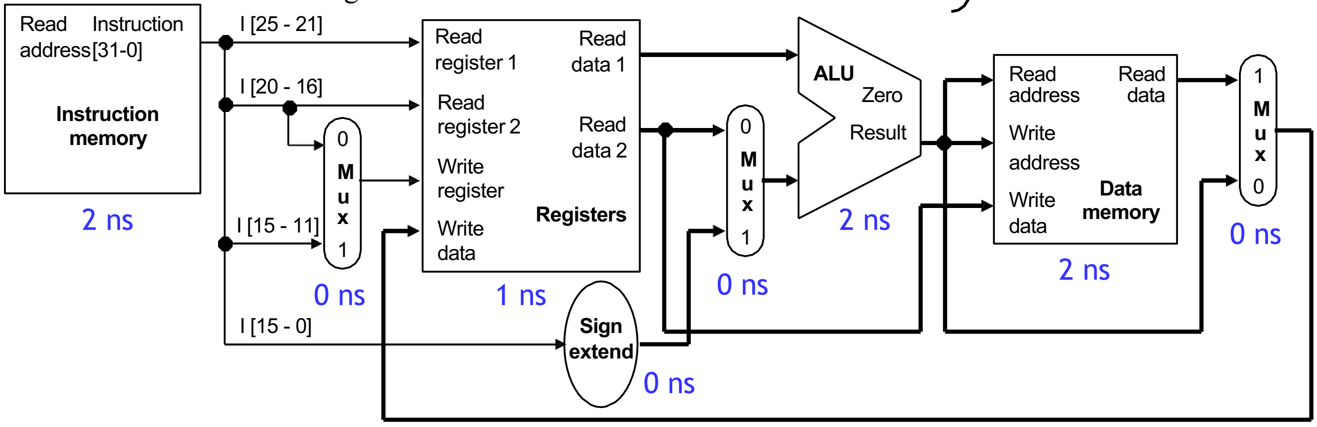
- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, lw $t0, –4($sp) is the slowest instruction needing ___ns.
  - — Assuming the circuit latencies below.



The Slowest Instruction...

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, lw $t0, –4($sp) is the slowest instruction needing ___ns.
  - — Assuming the circuit latencies below.

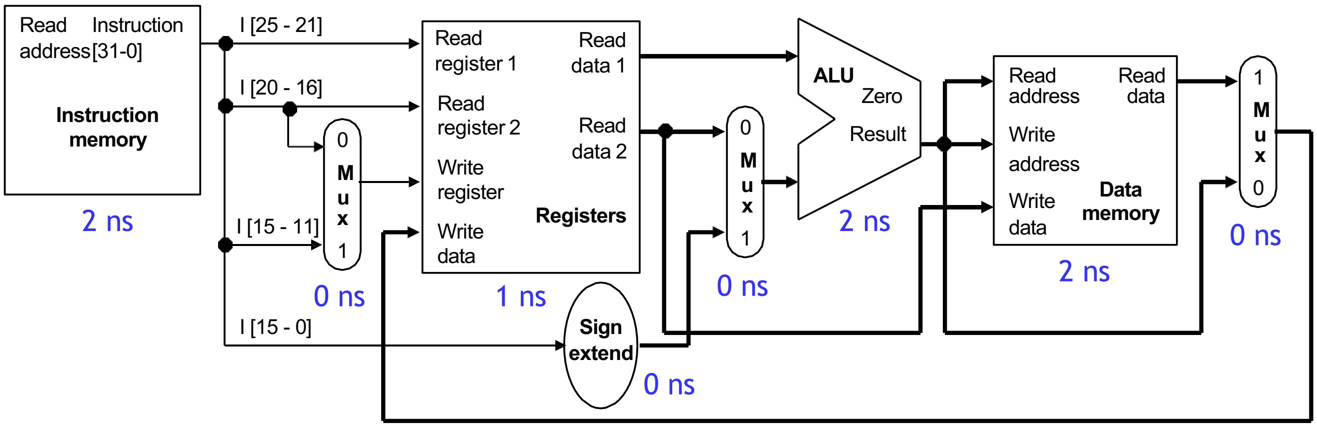
reading the instruction memory	2ns	8ns
reading the base register $sp	1ns	
computing memory address $sp-4	2ns	
reading the data memory	2ns	
storing data back to $t0	1ns	



...Determines the Clock Cycle Time

- If we make the cycle time 8ns then *every* instruction will take 8ns, even if they don't need that much time.
- For example, the instruction add $s4, $t1, $t2 really needs just 6ns.

reading the instruction memory	2 ns	6 ns
reading registers $t1 and $t2	1 ns	
computing $t1 + $t2	2 ns	
storing the result into $s0	1 ns	



How bad is this?

- With these same component delays, a sw instruction would need 7ns, and beq would need just 5ns.
- Let's consider the **gcc** instruction mix.

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be?

How bad is this?

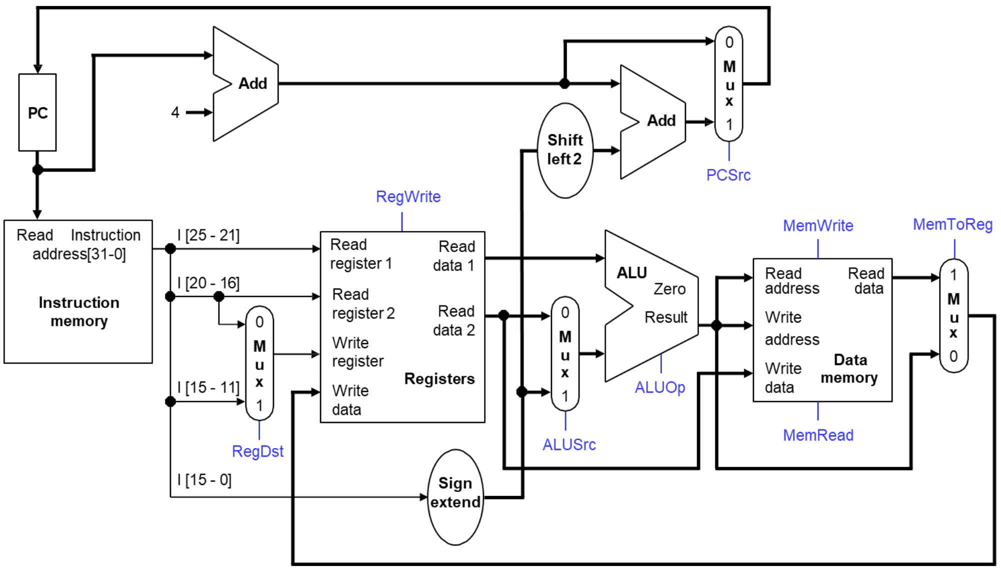
- With these same component delays, a sw instruction would need 7ns, and beq would need just 5ns.
- Let's consider the **gcc** instruction mix.

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be?

$$(48\% \text{ x } 6\text{ns}) + (22\% \text{ x } 8\text{ns}) + (11\% \text{ x } 7\text{ns}) + (19\% \text{ x } 5\text{ns}) = 6.36\text{ns}$$

- The single-cycle datapath is about 1.26 times slower!


PC
Add
4
Add
Shift left 2
M u x
0
1
PCSrc
RegWrite
Read address[31-0]
Instruction
Instruction memory
I [25 - 21]
I [20 - 16]
I [15 - 11]
I [15 - 0]
M u x
RegDst
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Sign extend
ALU
Zero
Result
ALUOp
ALUSrc
MemWrite
Read address
Write address
Write data
Read data
Data memory
MemRead
MemToReg