# UCCD1133
# Introduction to Computer Organisation and Architecture

## Chapter 6
## Processor, Memory System and Instruction Execution

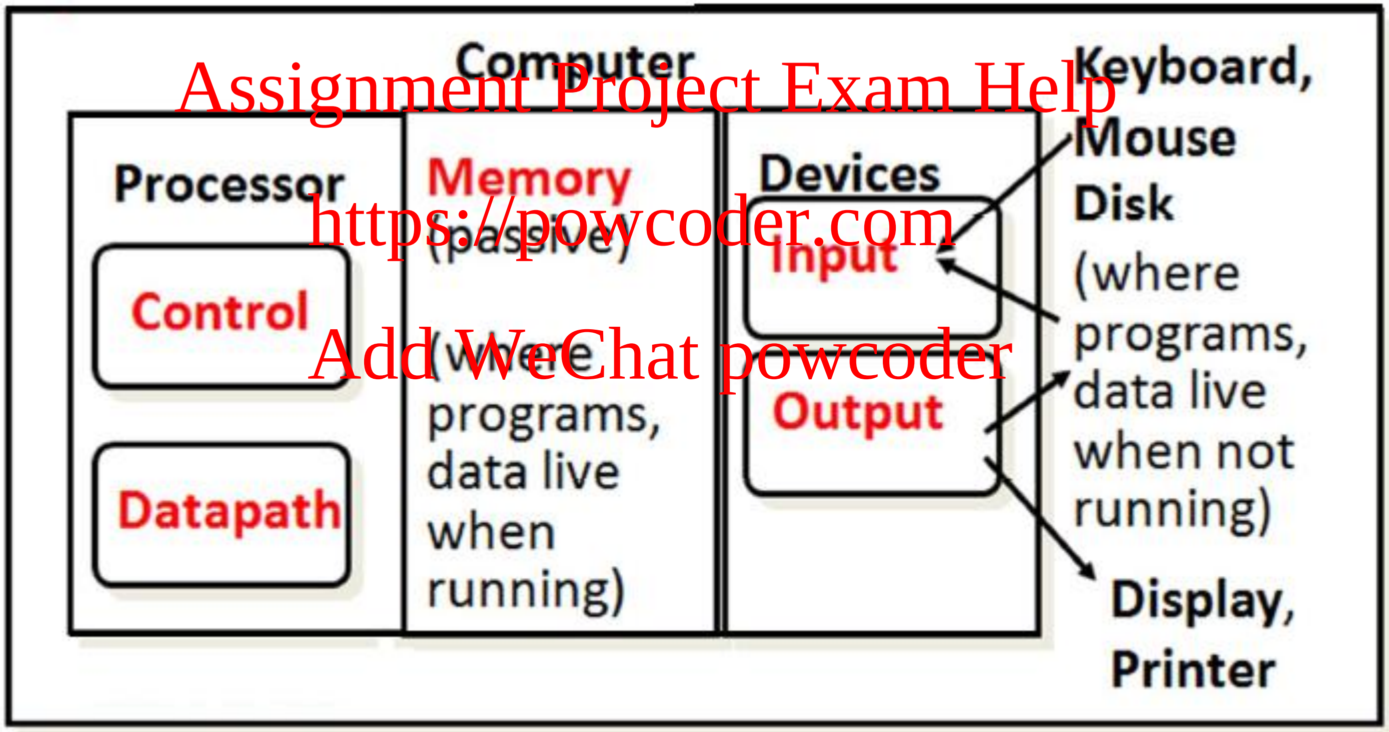
# Disclaimer

- This slide may contain copyrighted material of which has not been specifically authorized by the copyright owner. The use of copyrighted materials are solely for educational purpose. If you wish to use this copyrighted material for other purposes, you must first obtain permission from the original copyright owner.

Chapter 6-1

# Instruction execution cycle and Stages of processor

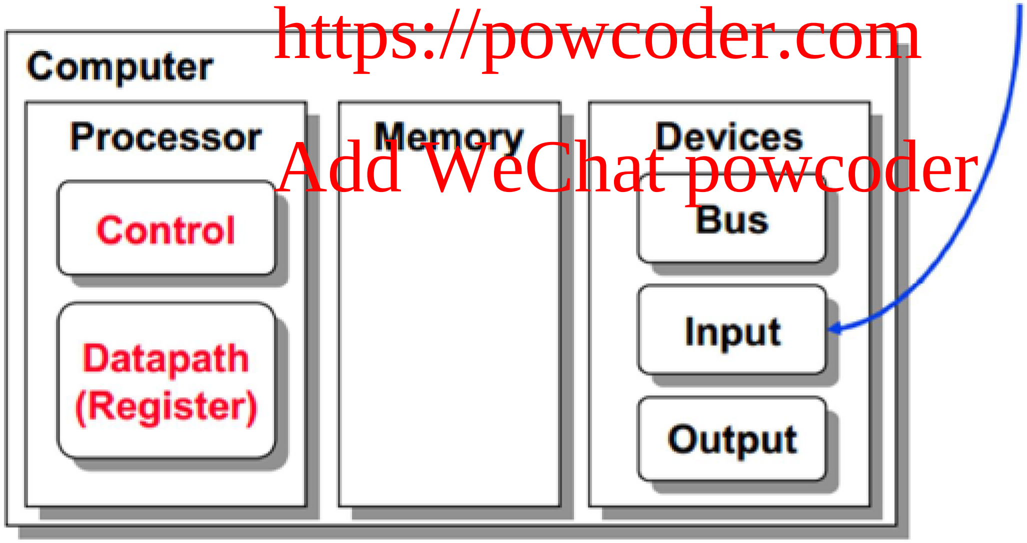
## Five Major Components of Computer Organization



# Instruction Execution Cycle

## Loading a Program via Input Device

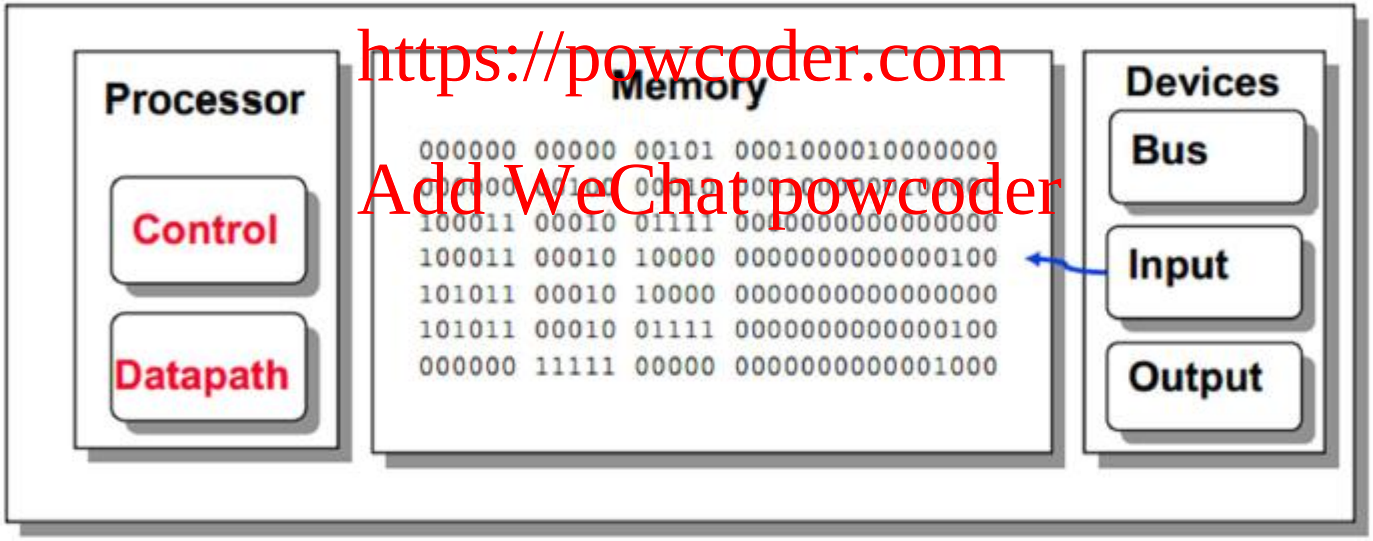
```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```



# Instruction Execution Cycle

## Storing a Program in Memory

Memory stores both instructions and data

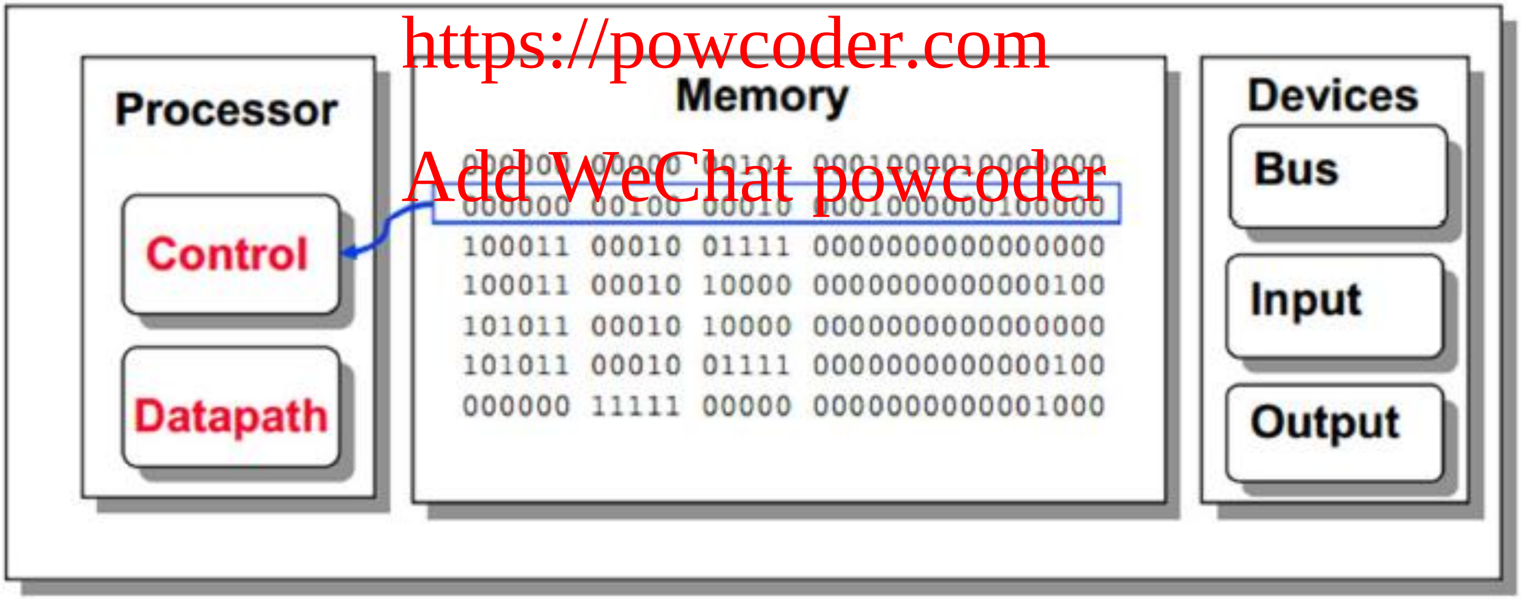


```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

# Instruction Execution Cycle

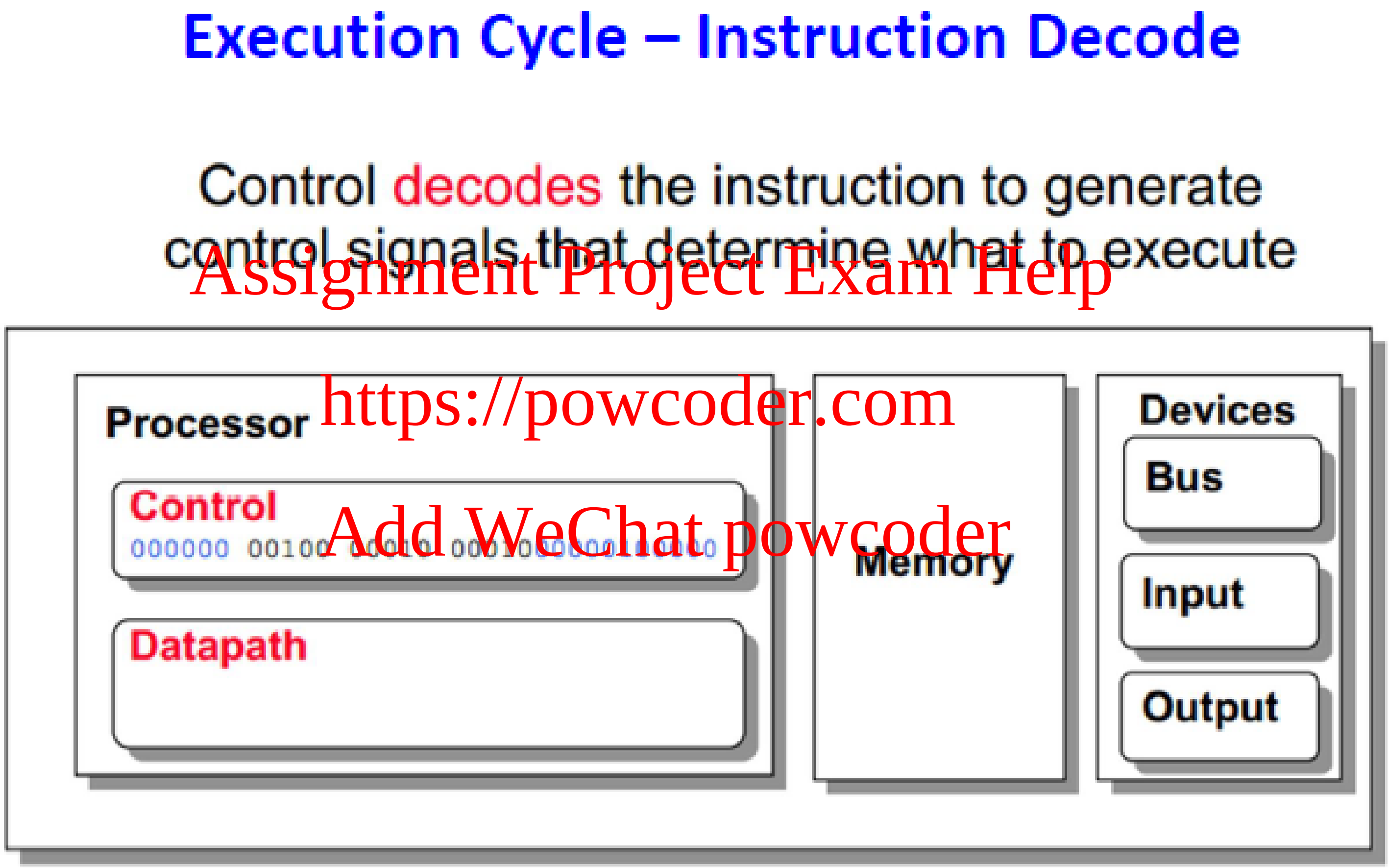
## Execution Cycle – Instruction Fetch

Processor fetches an instruction from memory



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

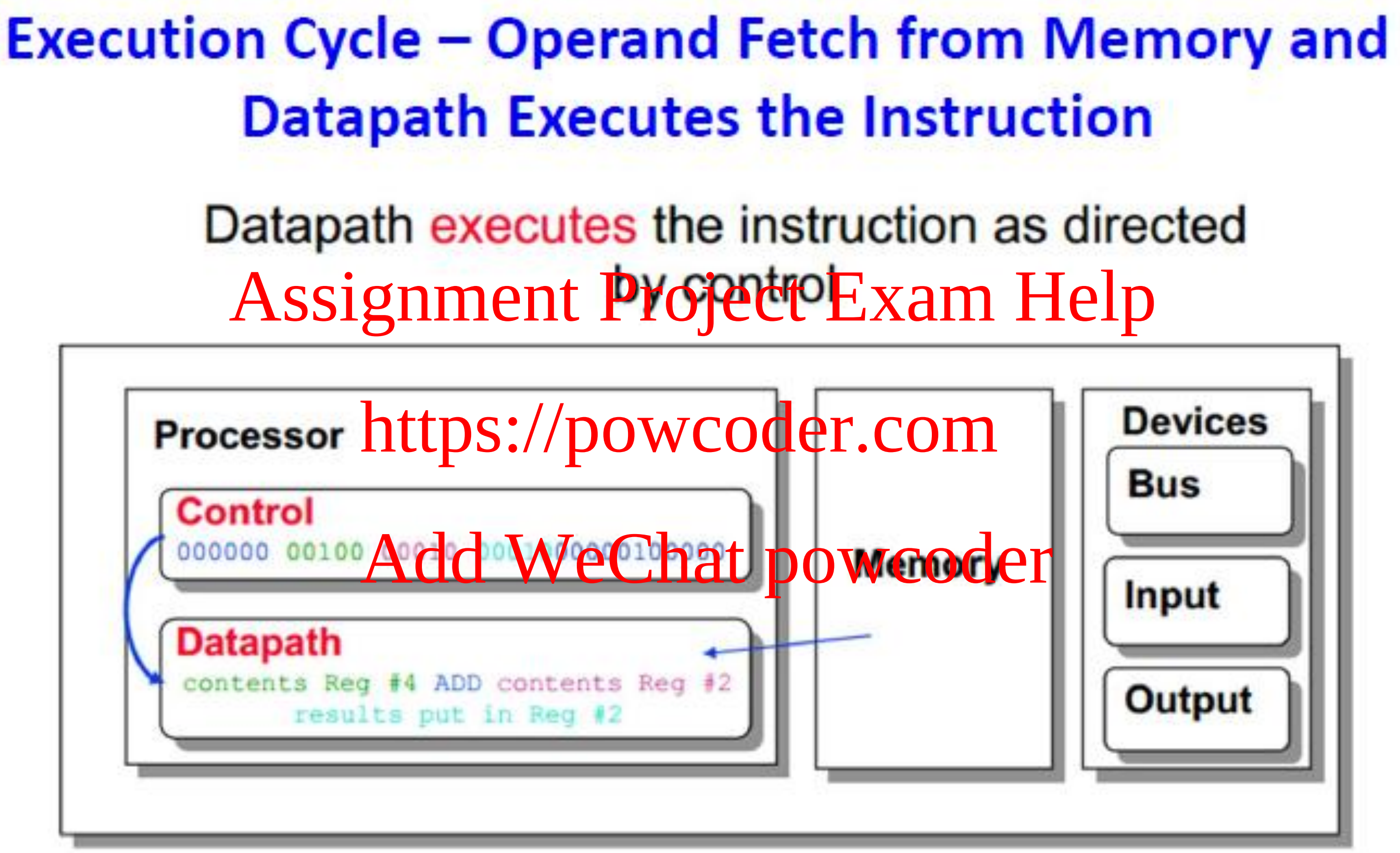
# Instruction Execution Cycle



## Execution Cycle – Instruction Decode

Control decodes the instruction to generate control signals that determine what to execute

# Instruction Execution Cycle



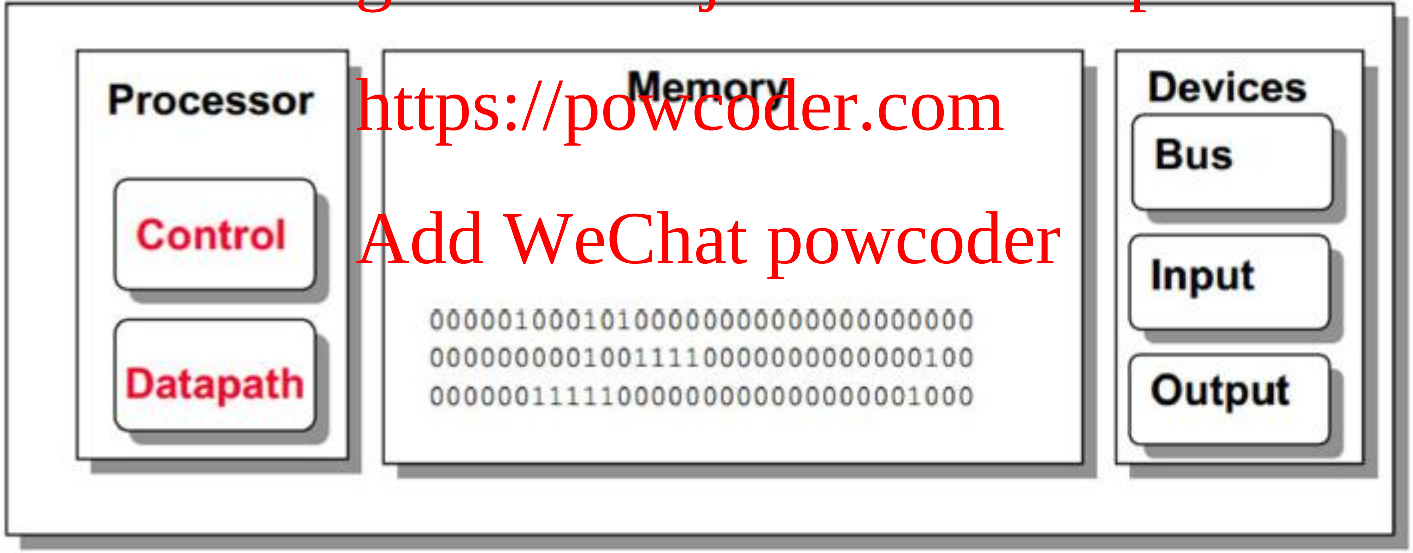
## Execution Cycle – Operand Fetch from Memory and Datapath Executes the Instruction

Datapath executes the instruction as directed by control

# Instruction Execution Cycle

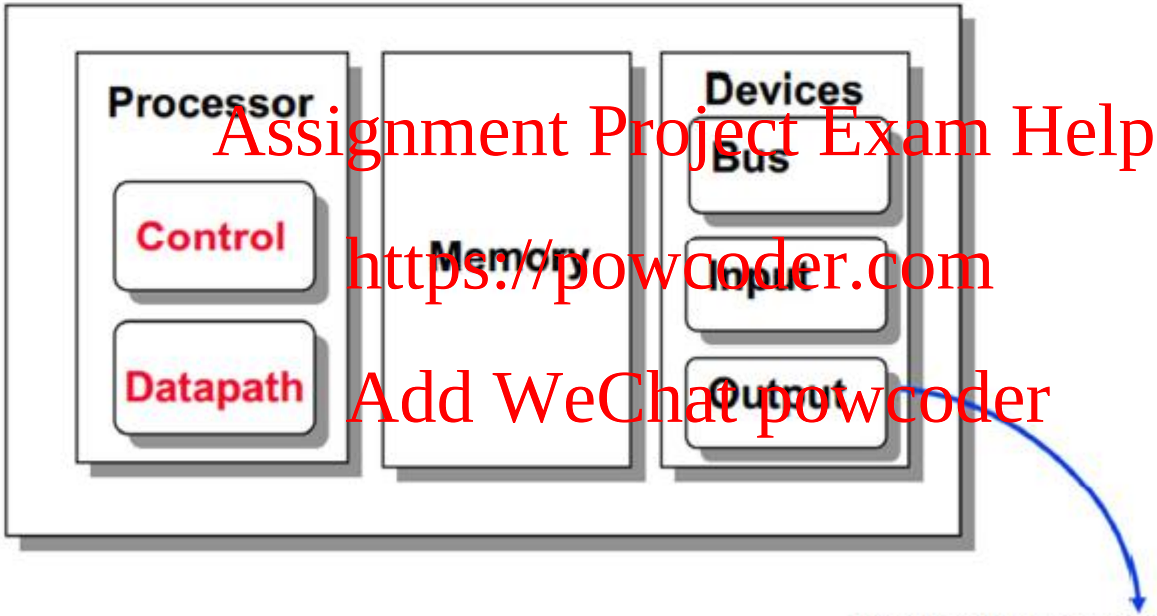
## Execution Cycle – – Result Store in Memory

At program completion the data to be output resides in memory



# Instruction Execution Cycle

## Data Output via Output Device

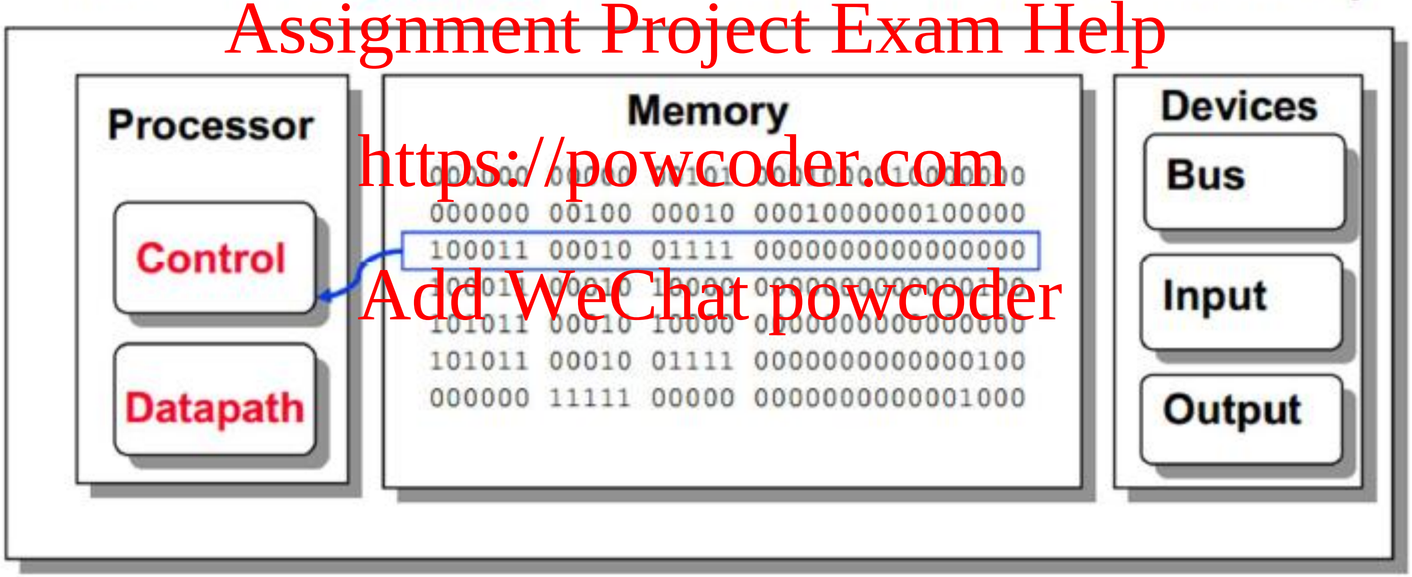


```
00000100010100000000000000000000
00000000010011110000000000000100
00000011111000000000000000001000
```

# Instruction Execution Cycle

## Execution Cycle Begins Again – Next Instruction Fetch

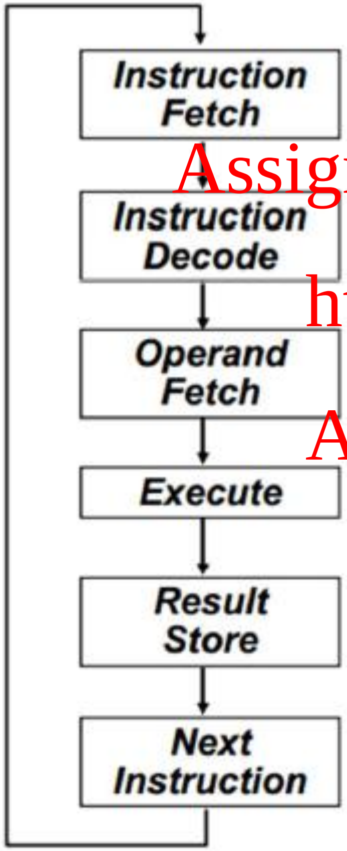
Processor **fetches** the *next* instruction from memory



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

# Instruction Execution Cycle

## Machine Interpretation: Instruction Execution Cycle in Computer Organisation



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

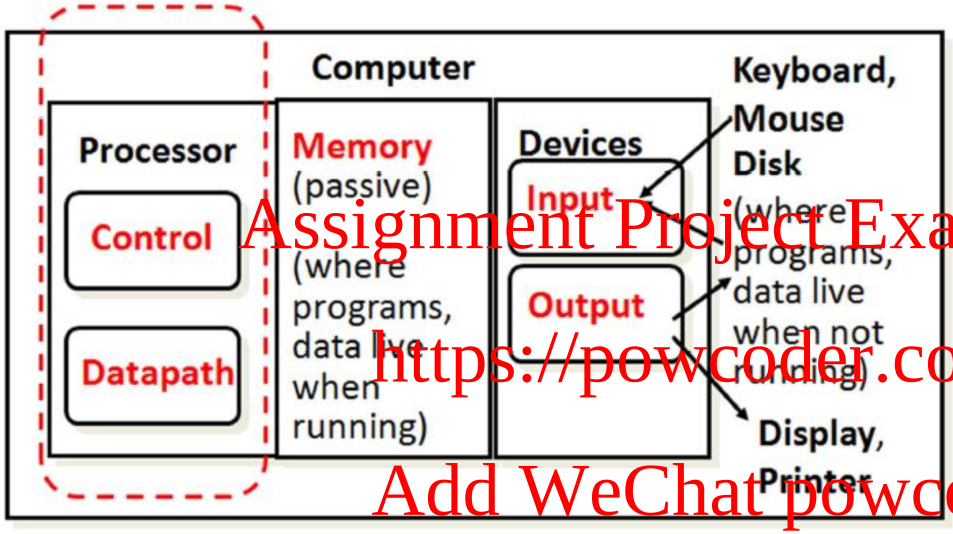
Compute result value or status

Deposit results in storage for later use

Determine successor instruction

# Instruction Execution Cycle

## Five Major Components of Computer Organization

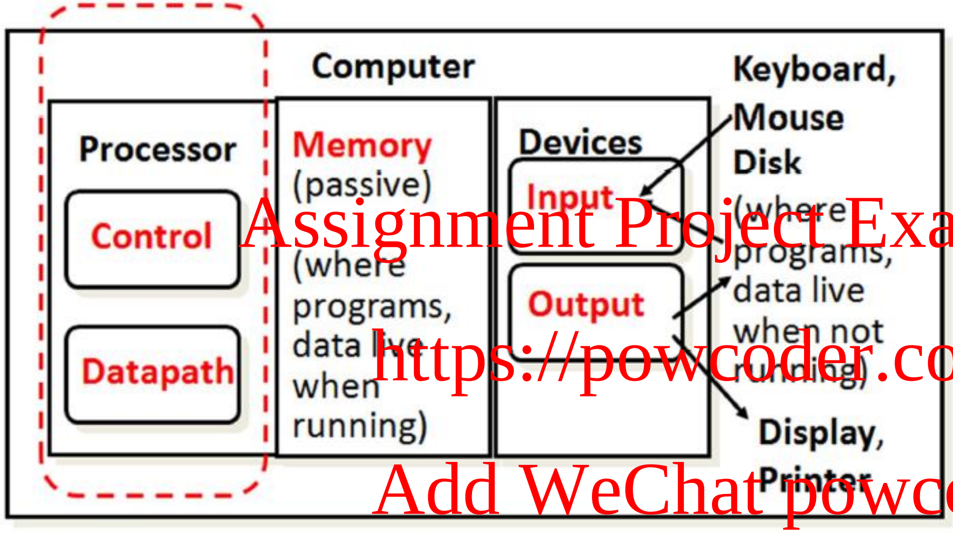


Control needs to have circuitry to:

- Decode the instruction
- Issue signals that control the way information flows between datapath components
- Control what operations the datapath's functional units perform
- Decide which is the next instruction and obtain it from memory

# Instruction Execution Cycle

## Five Major Components of Computer Organization



Datapath needs to have circuitry to:

- Execute instructions – functional units (e.g., adder) and storage locations (e.g., register file)
- Interconnect the functional units accordingly
- Load data from and store data to memory

# Performance of Computer Systems

- Basic performance measure:
  - **Response time/ Execution time:**
    The time between the start and the completion of a task
  - **Throughput:**
    The total amount of tasks done in a given time period

- Main factors influencing the performance
  - Processor and memory
  - Input/output controllers and peripherals
  - Compilers
  - Operating system

- Challenge is to satisfy constraints of:
  - Cost
  - Power
  - Performance

# Performance of CPU

- To measure CPU performance

**Execution Time / CPU Time**
= Instruction count × CPU × Clock cycle time
= (instructions/program)(cycles/instruction)(seconds/clock cycle)

- CPI (Cycles/instruction): Average number of clock cycles per instruction for a program
- clock cycle (seconds/cycle): The time for one clock period
    - Note: clock cycle = 1/clock rate

- How to improve CPU time:
    - Instruction count: ISA and compiler technology
    - CPI: organisation, ISA and compiler technology
    - Clock rate: hardware technology and organisation

# Performance of CPU

- Example

Consider the following performance measurements for a program:

| Measurement | Computer A | Computer B |
|---|---|---|
| Instruction count | 10 billion | 8 billion |
| Clock rate | 4 GHz | 4 GHz |
| CPI | 1.0 | 1.1 |

Which computer is faster?

- Solution:

Computer A execution time = $(10 \times 10^{9}) \times 1.0 \times (2.5 \times 10^{-10}\ \text{sec})$

$= 2.5$ sec

Computer B execution time = $(8 \times 10^{9}) \times 1.1 \times (2.5 \times 10^{-10}\ \text{sec})$

$= 2.2$ sec

Computer B is faster.

Chapter 6-2

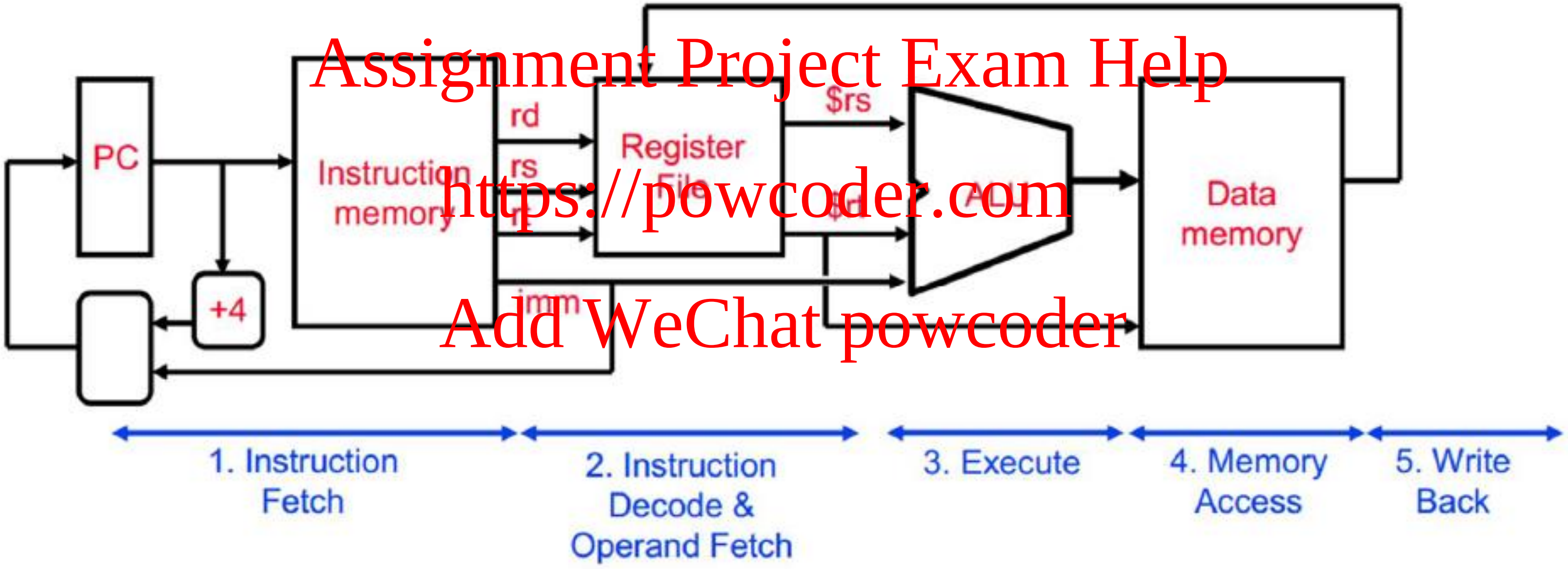
# Processors: Single-cycle, Multi-cycle and Pipeline

# Microarchitecture

- **Microarchitecture:** how to implement an architecture in hardware
- Processor:
  - **Datapath:** functional blocks
  - **Control:** control signals

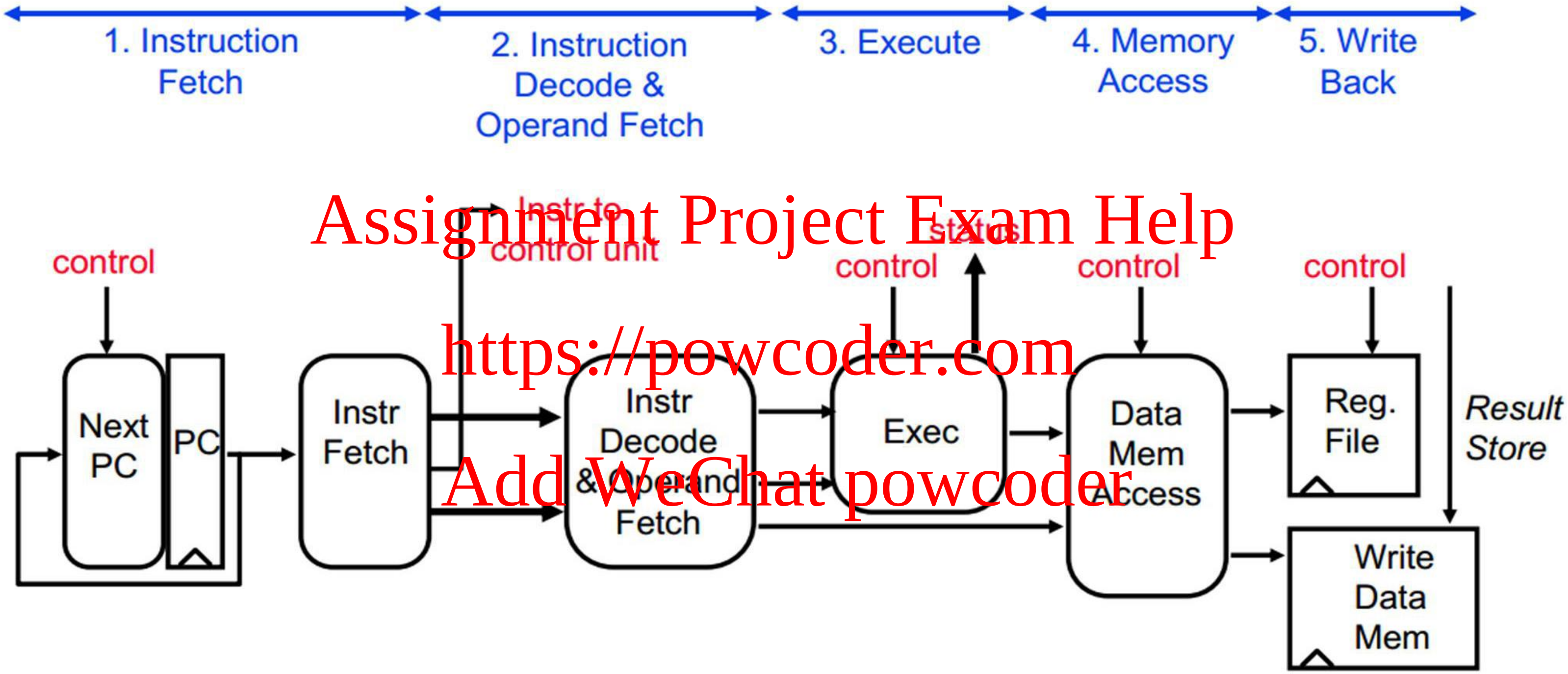
# Datapath Abstract Implementation View

- Assemble the components
  - Based on stages of “instruction execution cycle”
  - Obtain an abstract datapath implementation view.
  - Note : one could have different number of stages for different CPU architecture



# Datapath Abstract Implementation View

- Datapath with control signals



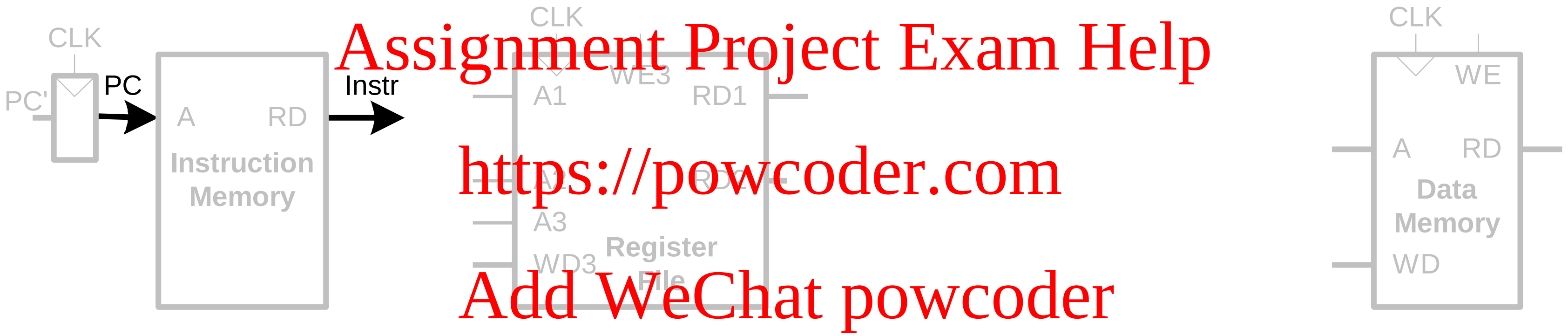
# Microarchitecture

- Multiple implementations for a single architecture:
  - Single-cycle: Each instruction executes in a single cycle
  - Multicycle: Each instruction is broken into series of shorter steps
  - Pipelined: Each instruction broken up into series of steps & multiple instructions execute at once

# Single-Cycle Datapath: lw

Example: `lw rt, imm(rs)`

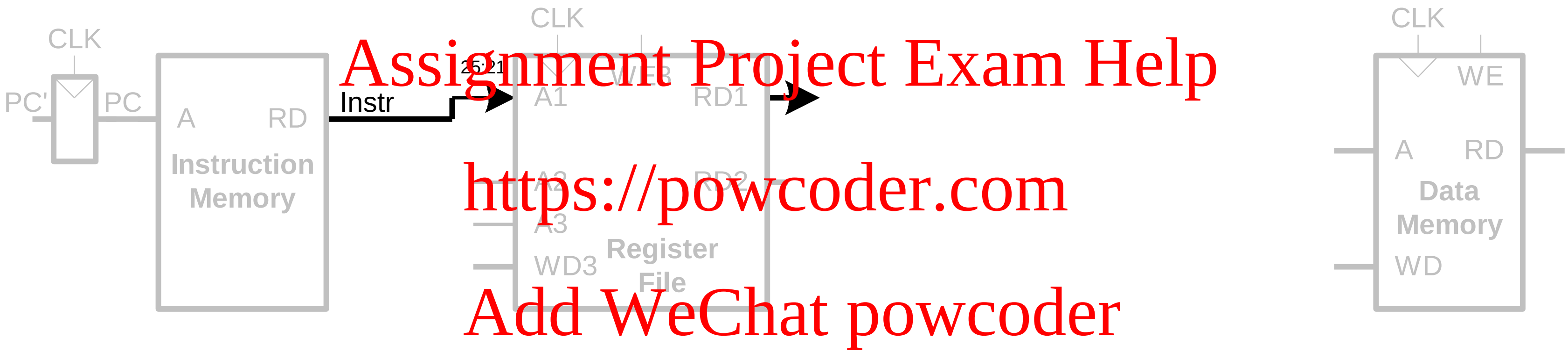
**STEP 1:** Fetch instruction



# Single-Cycle Datapath: lw Register

Example: **`lw rt, imm(rs)`**

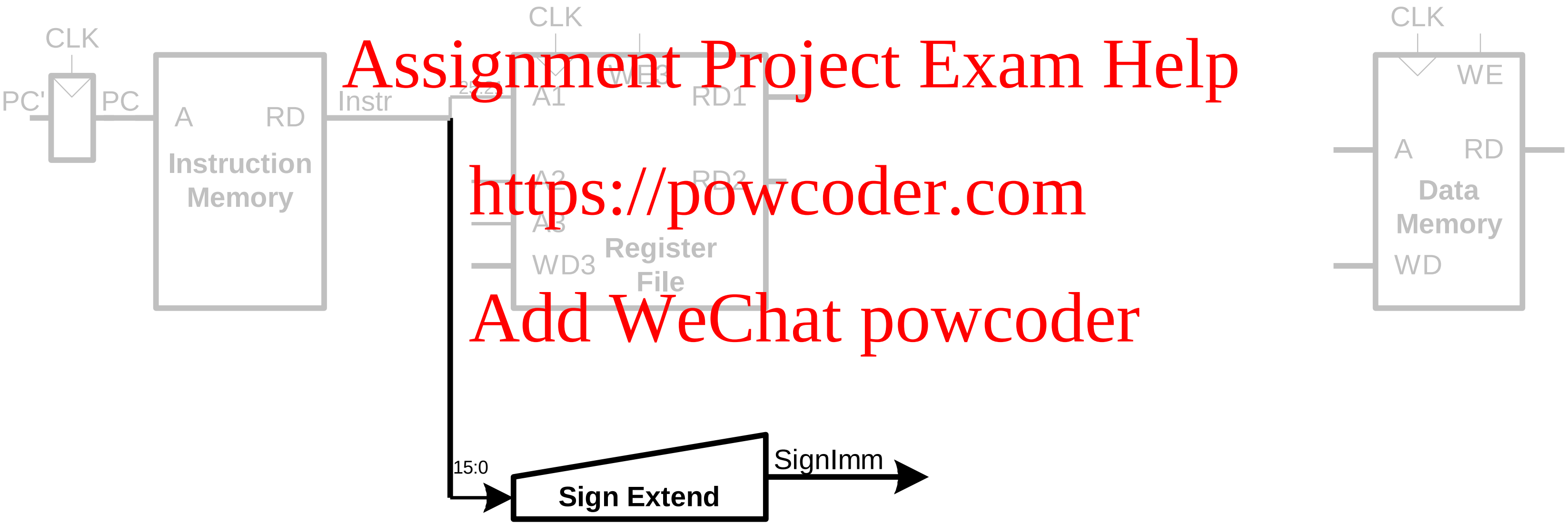
**STEP 2:** Read source operands from RF



# Single-Cycle Datapath: lw Immediate

Example: **`lw rt, imm(rs)`**

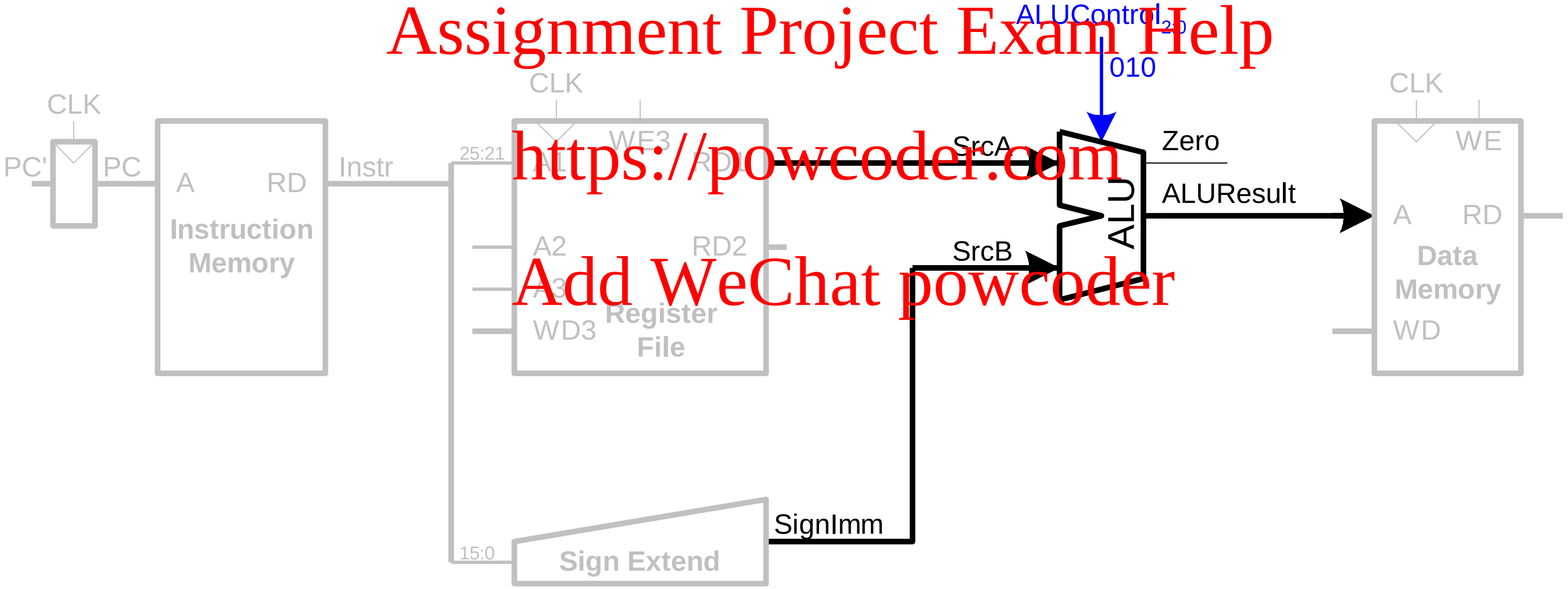
**STEP 3:** Sign-extend the immediate



# Single-Cycle Datapath: lw address

Example: `lw rt, imm(rs)`

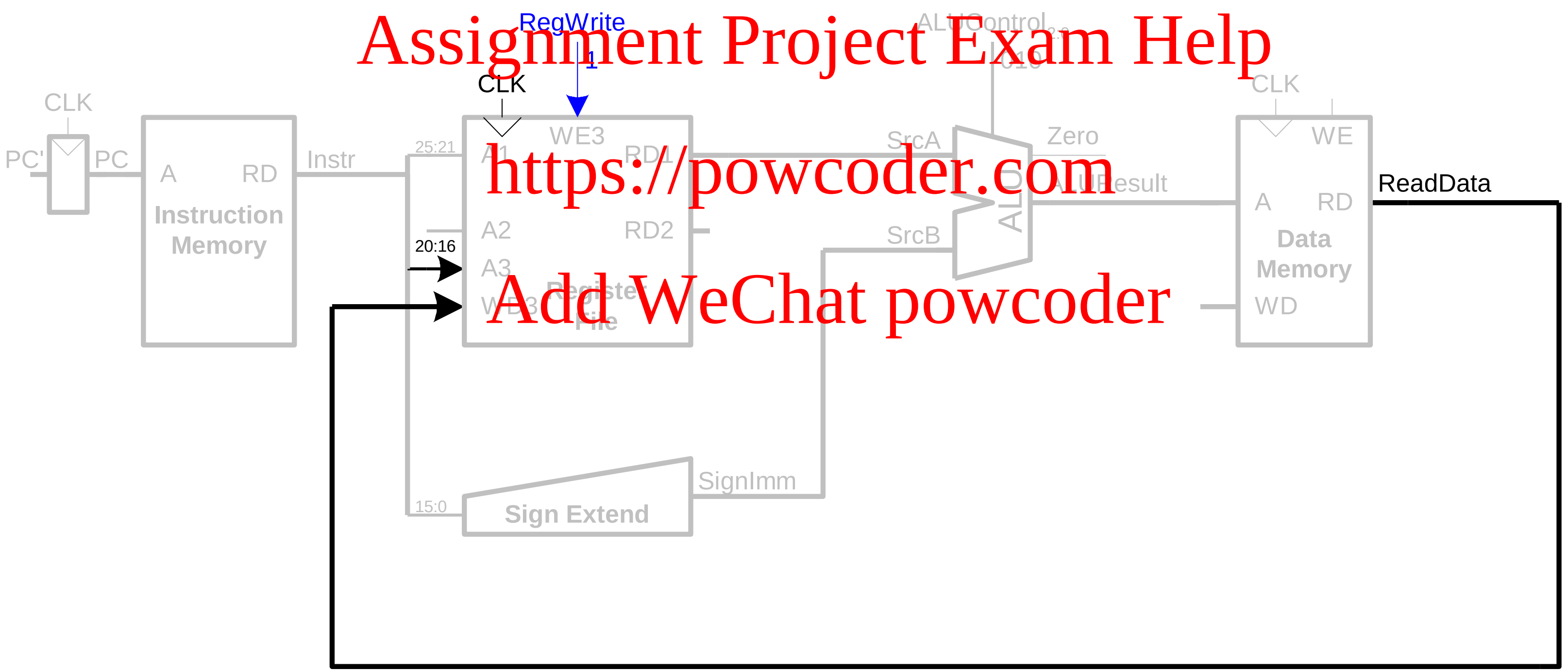
**STEP 4:** Compute the memory address



# Single-Cycle Datapath: lw Memory

Example: `lw rt, imm(rs)`

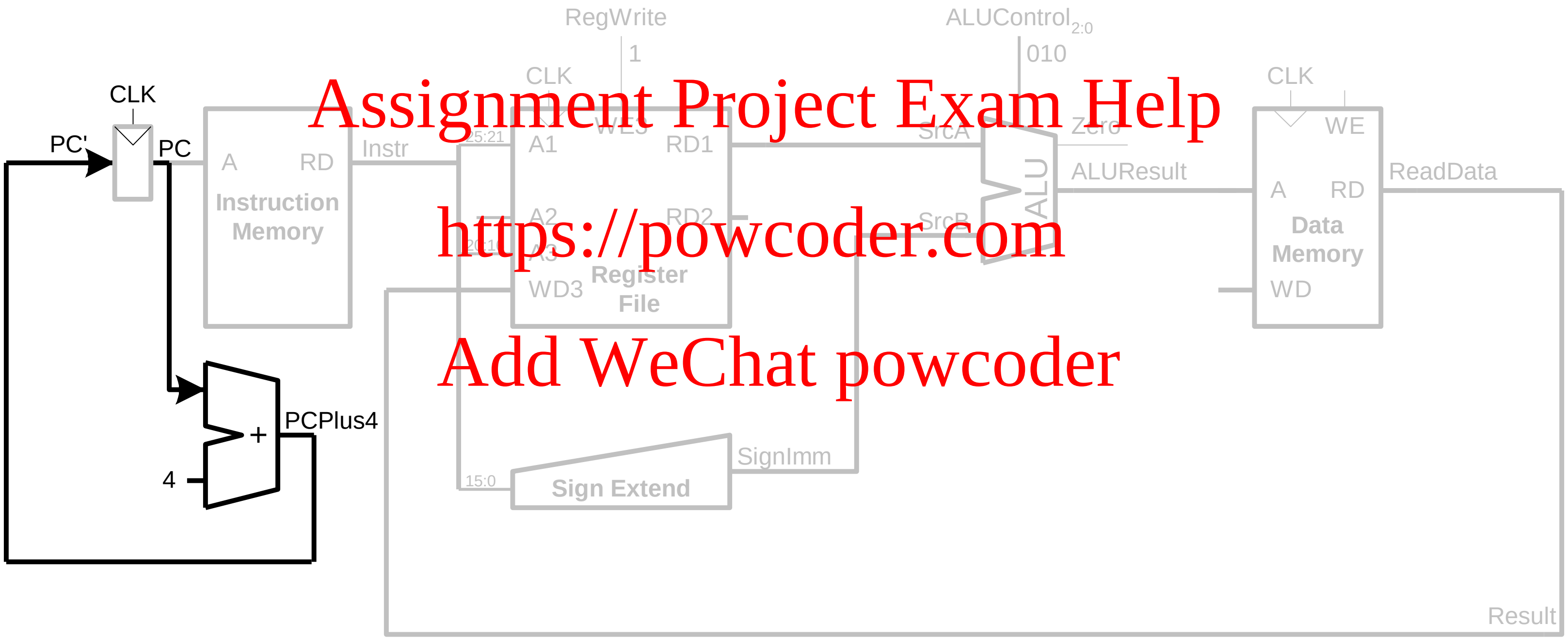
**STEP 5:** Read data from memory and write it back to register file



# Single-Cycle Datapath: lw PC

Example: **`lw rt, imm(rs)`**

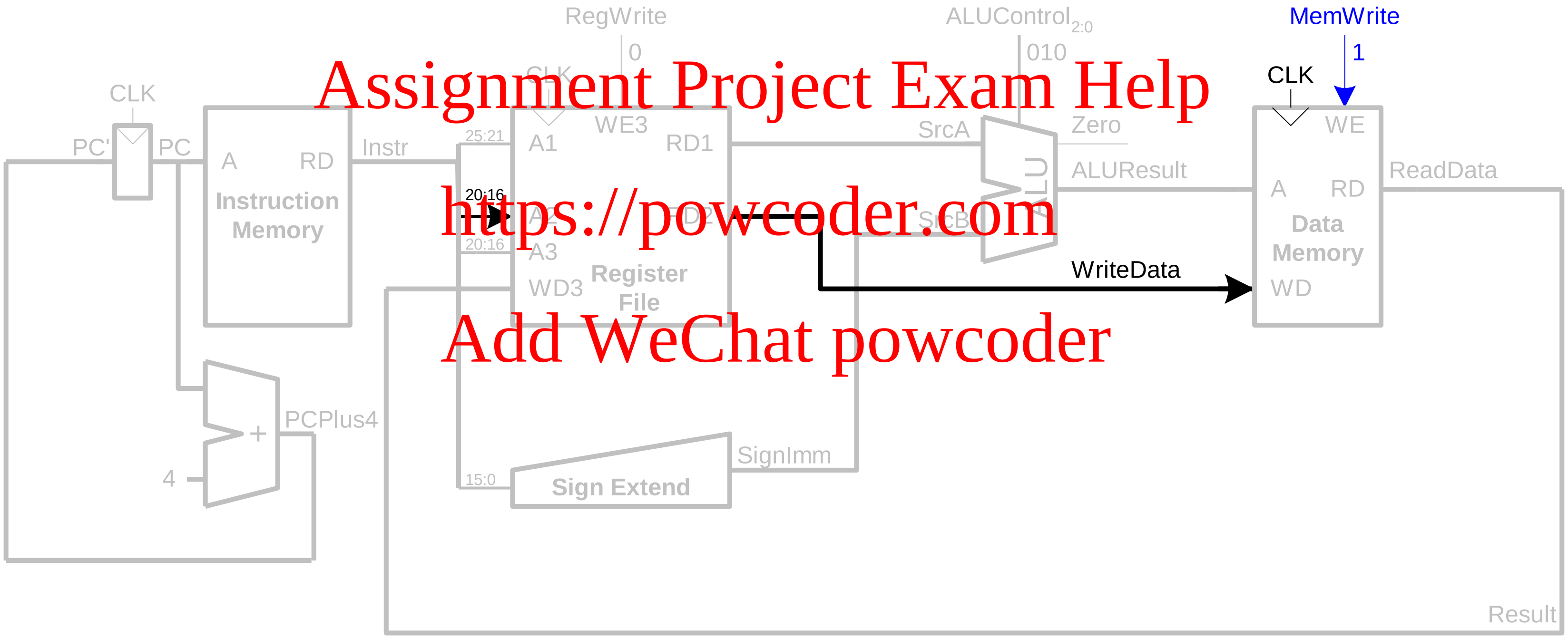
**STEP 6:** Determine address of next instruction



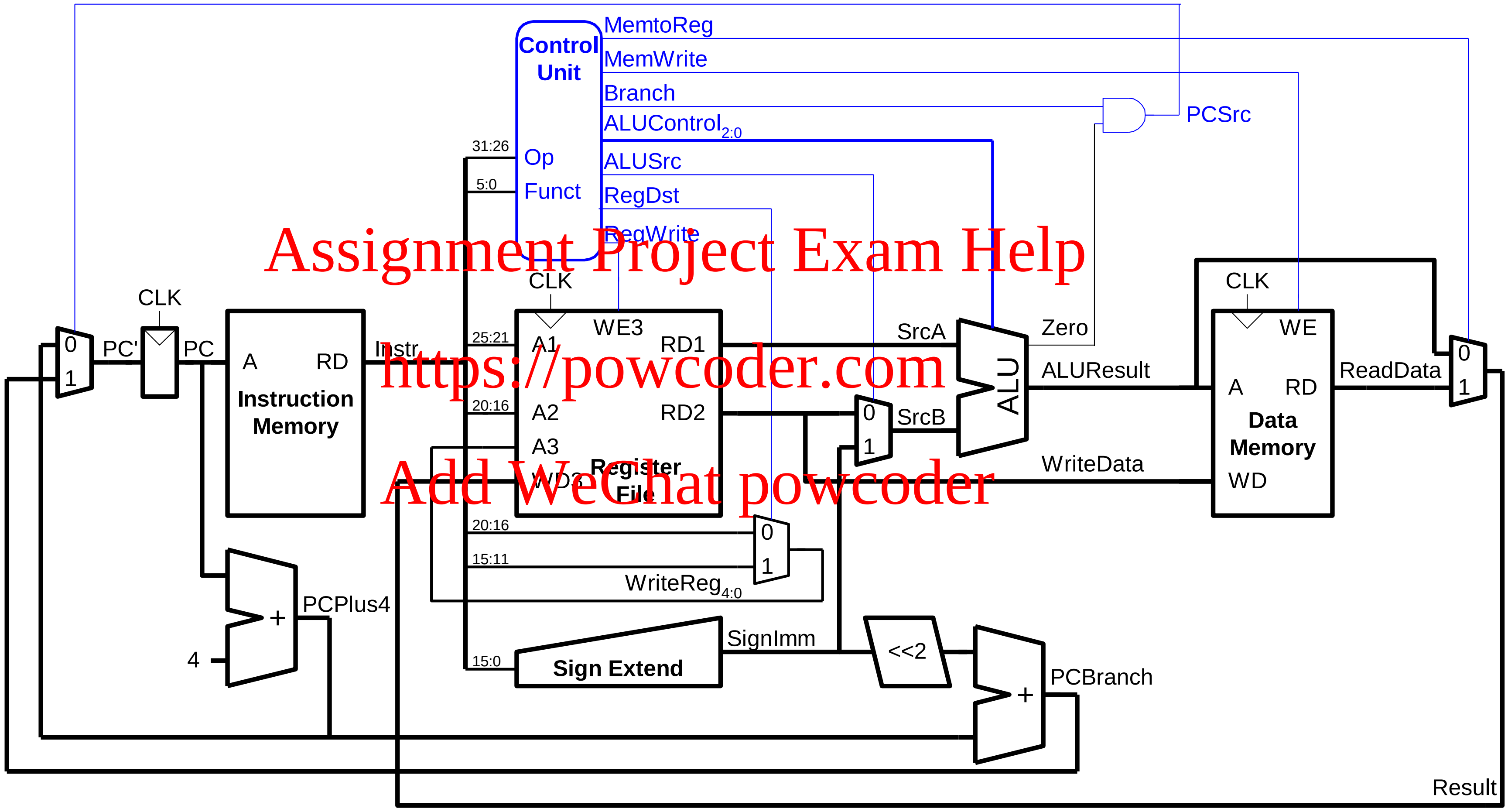
# Single-Cycle Datapath: sw

Example: `sw rt, imm(rs)`

Write data in `rt` to memory

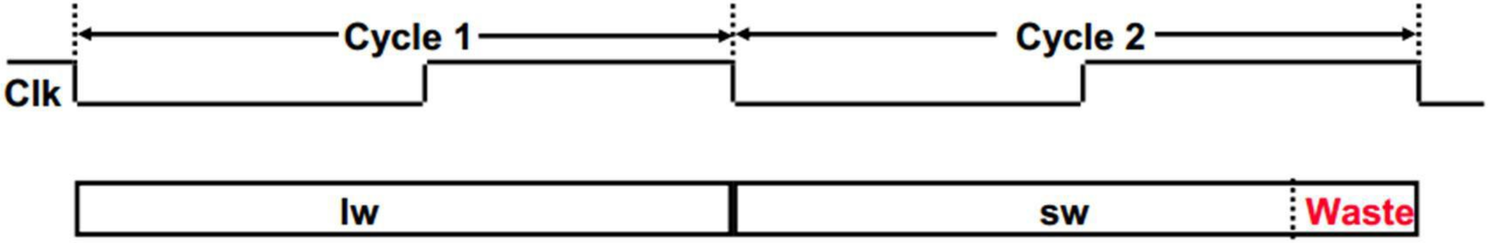


# Single-Cycle Processor



# Drawback of Single-Cycle

- Uses the clock cycle inefficiency – the clock cycle is set to accommodate the slowest instruction, and must be the same length for all instructions
  - Problem:
    - all instructions take as much time as the slowest instruction than needed
    - Especially problematic for more complex instructions like floating point multiply
- Need to duplicate resources that are used more than once per cycle – waste area and power.
  - Some functional units (eg. adders) must be duplicate since they can not be shared during a clock cycle
- Real memory is slower than idealized memory
  - Cannot always get the job done in one (short) cycle
  - Need to further stretch the clock period – more slower.



# Multicycle MIPS Processor

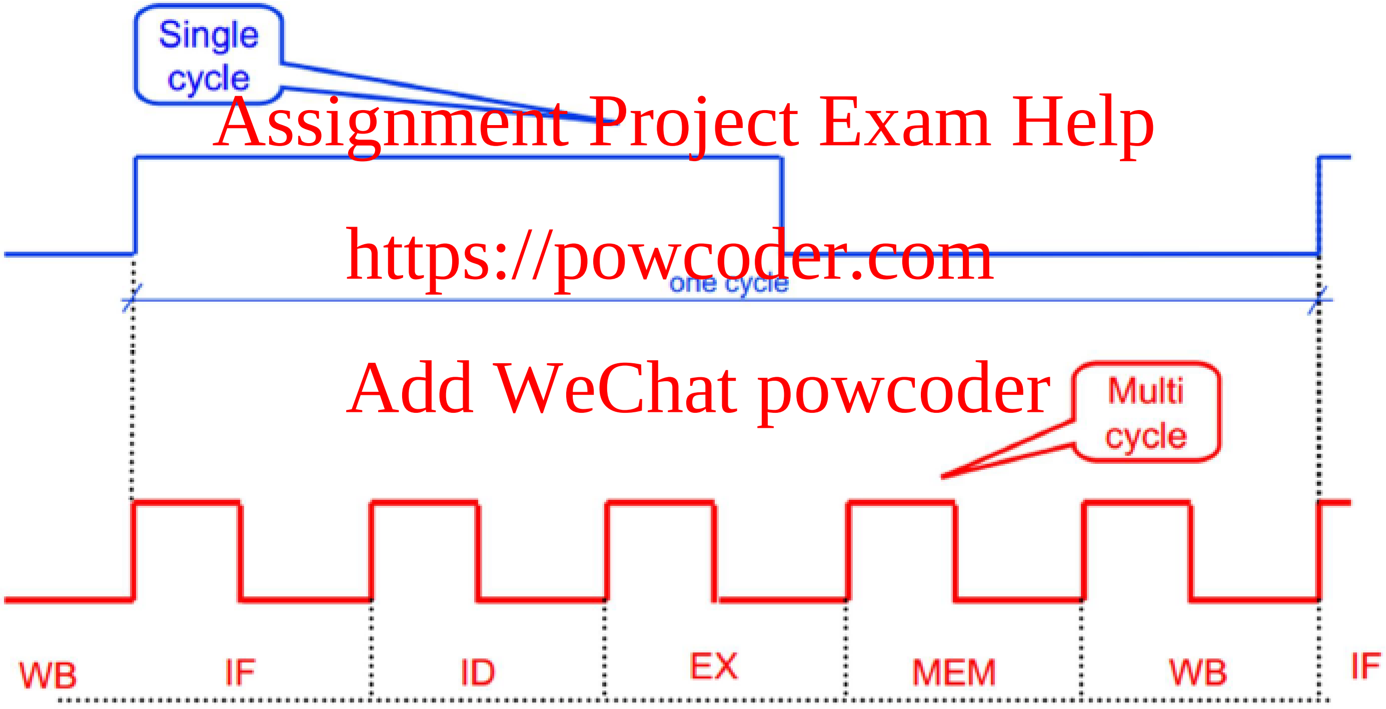
- Single-cycle datapath => CPI=1, CCT => long
- Multi-cycle processors
  - Require more complex control – control is the hard part.
  - Avoid idling – different instructions to take a different number of clock cycles.
  - Faster clock rates.
  - Data path allows greater sharing of hardware.

- MIPS makes control easier
  - Instructions same size
  - Source registers always in same place
  - Immediate same size, location
  - Operations always on registers/immediates

- Functional units can be used more than once per instruction as long as they are used on different clock cycles, as a result
  - Only one memory – holding both instructions and data
  - Only one ALU (used almost every cycle) – saves two adders

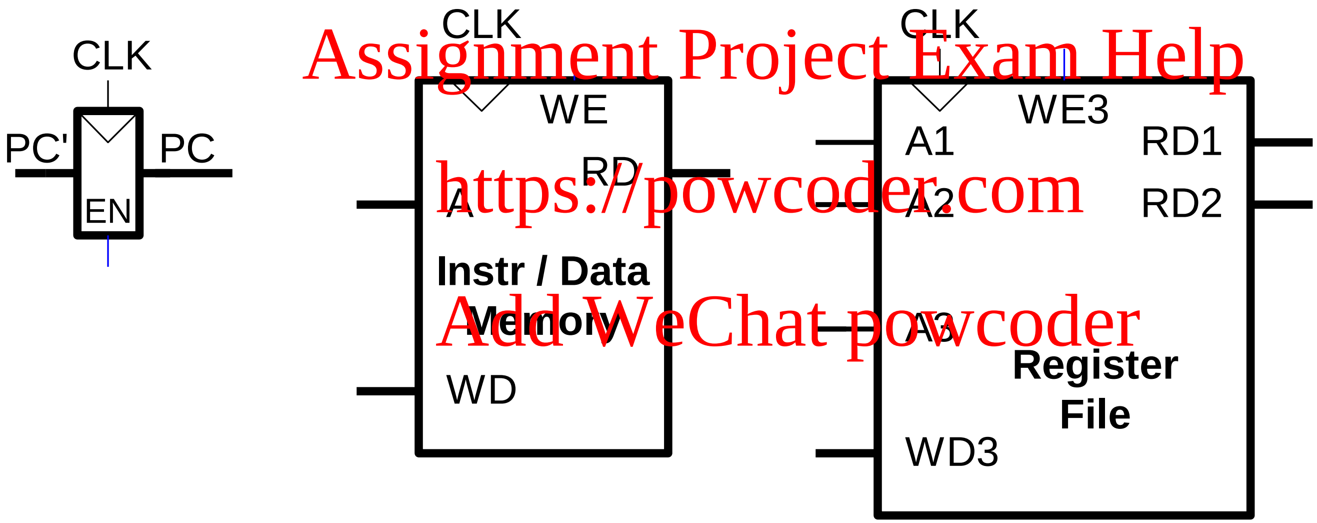
# Multicycle MIPS Processor

## One Cycle to Multi-Cycle Conversion



# Multicycle State Elements

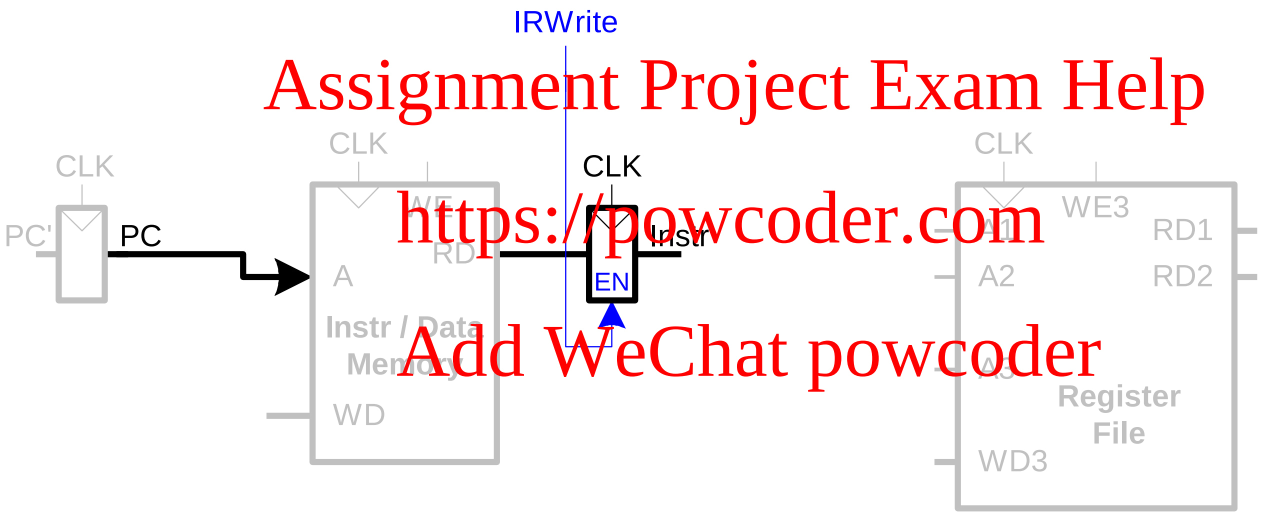
- Replace Instruction and Data memories with a single unified memory – more realistic



# Multicycle Datapath: Instruction Fetch

Example: **`lw rt, imm(rs)`**

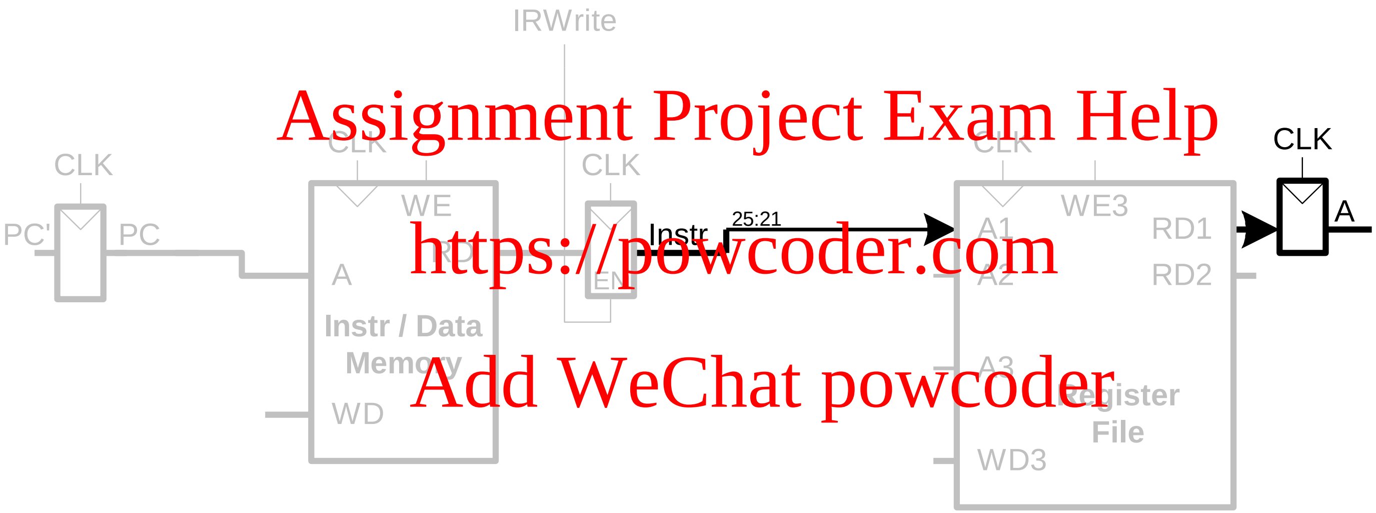
**STEP 1:** Fetch instruction



# Multicycle Datapath: lw Register

Example: **`lw rt, imm(rs)`**

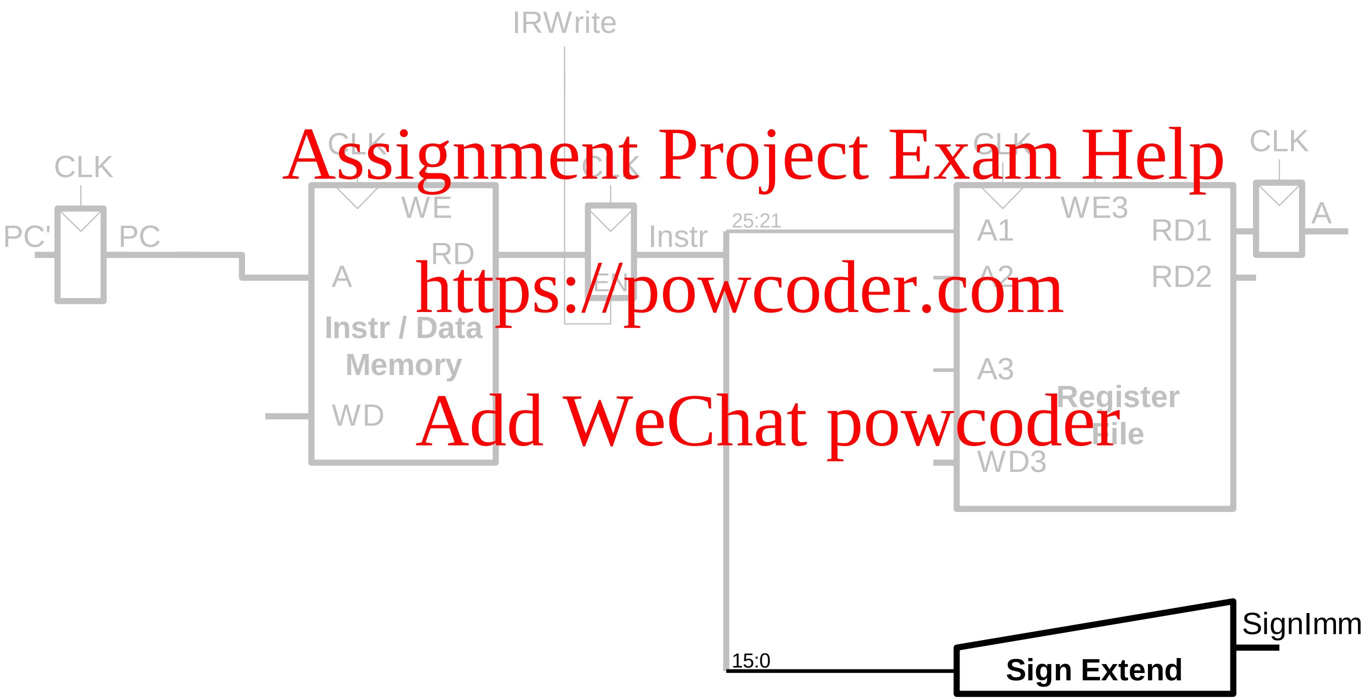
**STEP 2a:** Read source operands from RF



# Multicycle Datapath: lw Immediate

Example: **`lw rt, imm(rs)`**

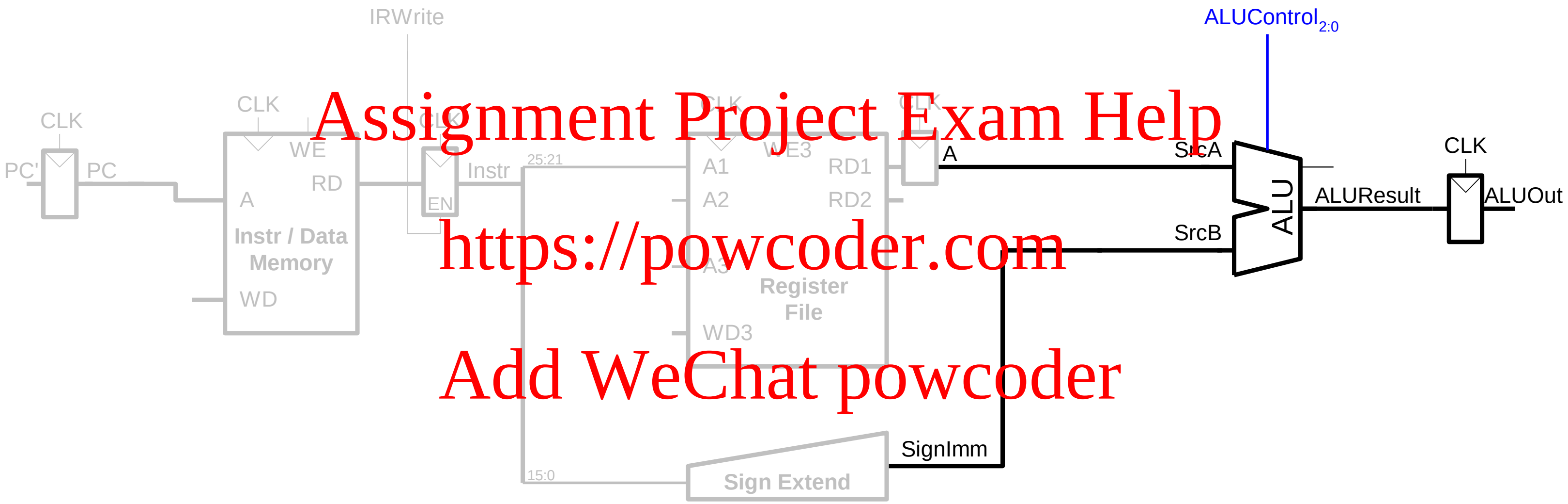
**STEP 2b:** Sign-extend the immediate



# Multicycle Datapath: lw Address

Example: `lw rt, imm(rs)`

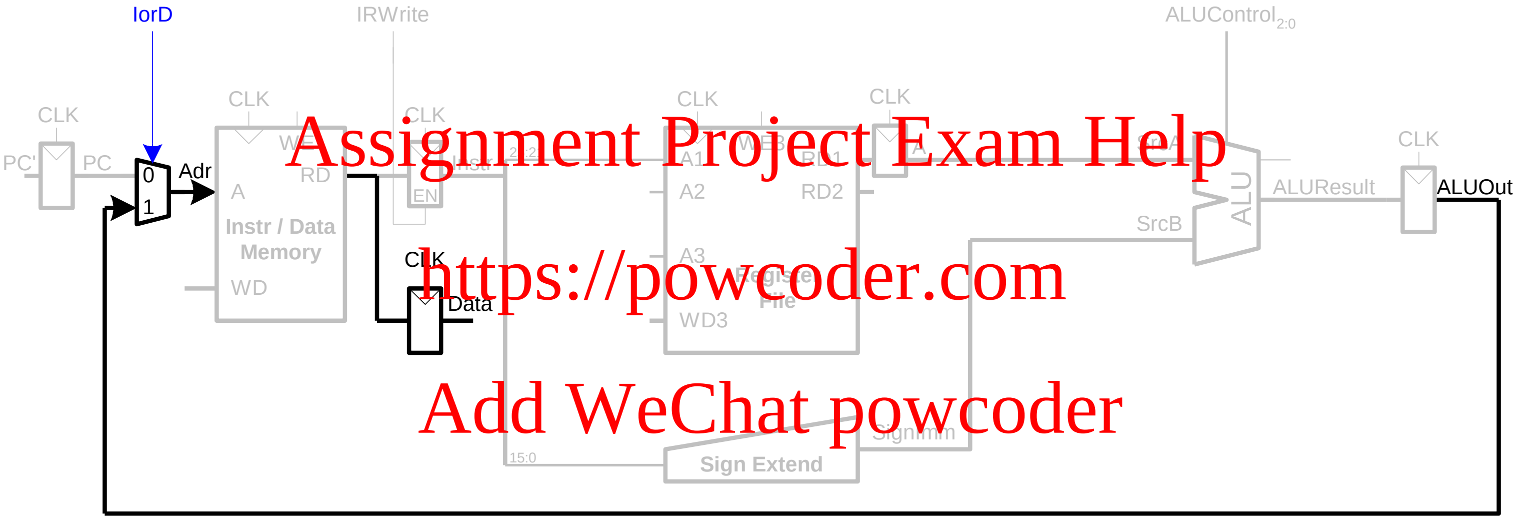
**STEP 3:** Compute the memory address



# Multicycle Datapath: lw Memory Read

Example: `lw rt, imm(rs)`

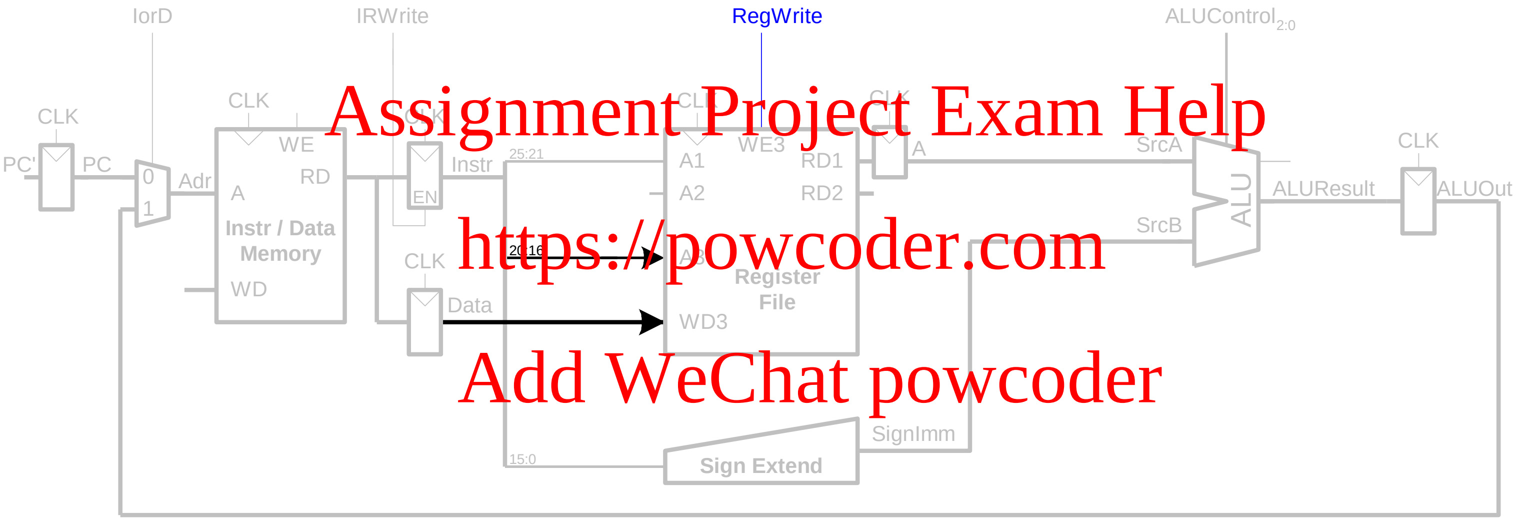
**STEP 4:** Read data from memory



# Multicycle Datapath: lw Write Register

Example: `lw rt, imm(rs)`

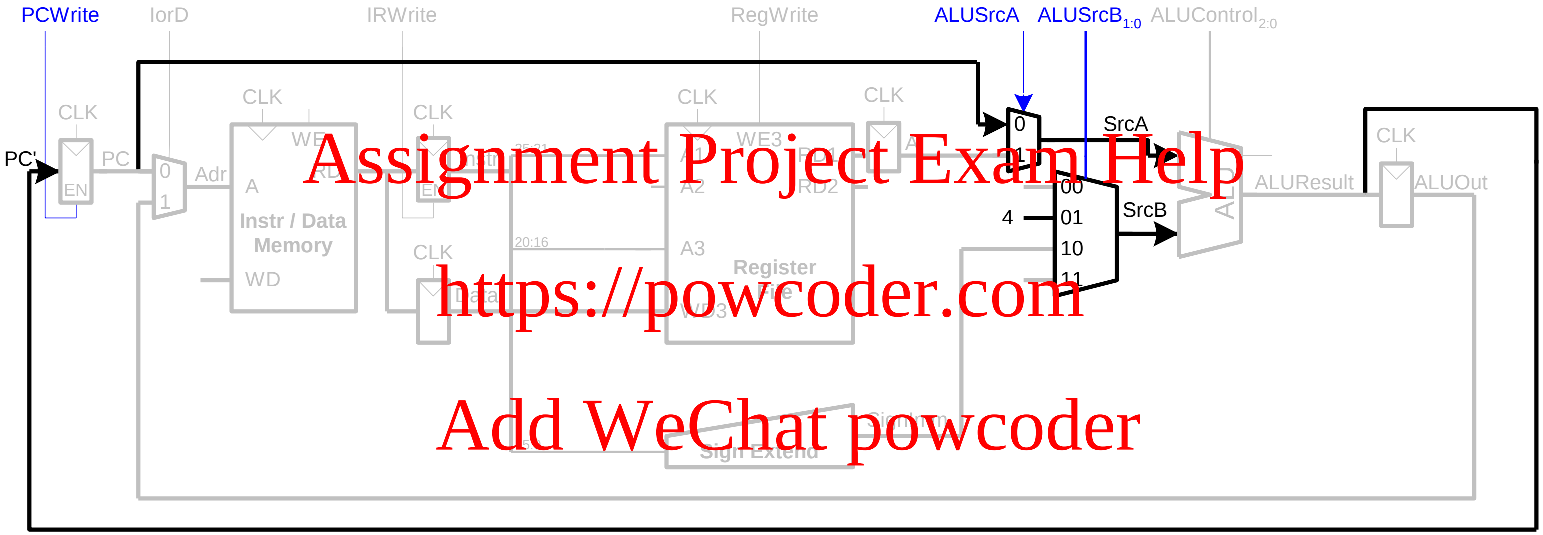
**STEP 5:** Write data back to register file



# Multicycle Datapath: Increment PC

Example: **`lw rt, imm(rs)`**

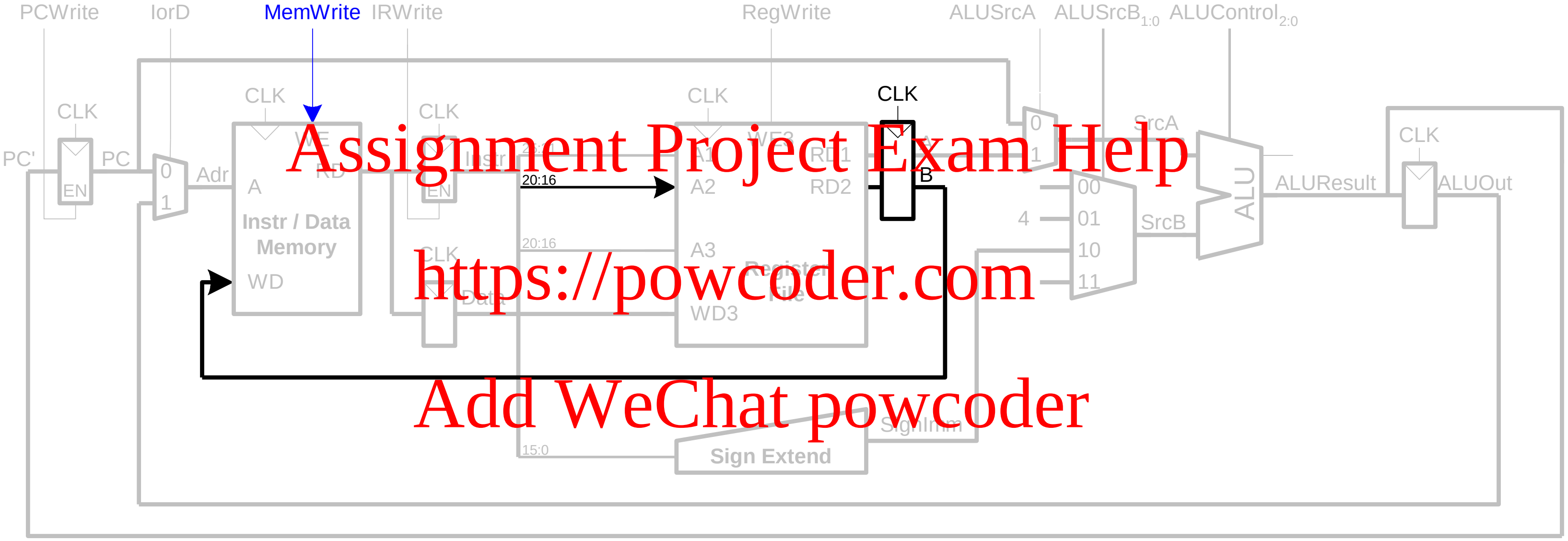
**STEP 6:** Increment PC



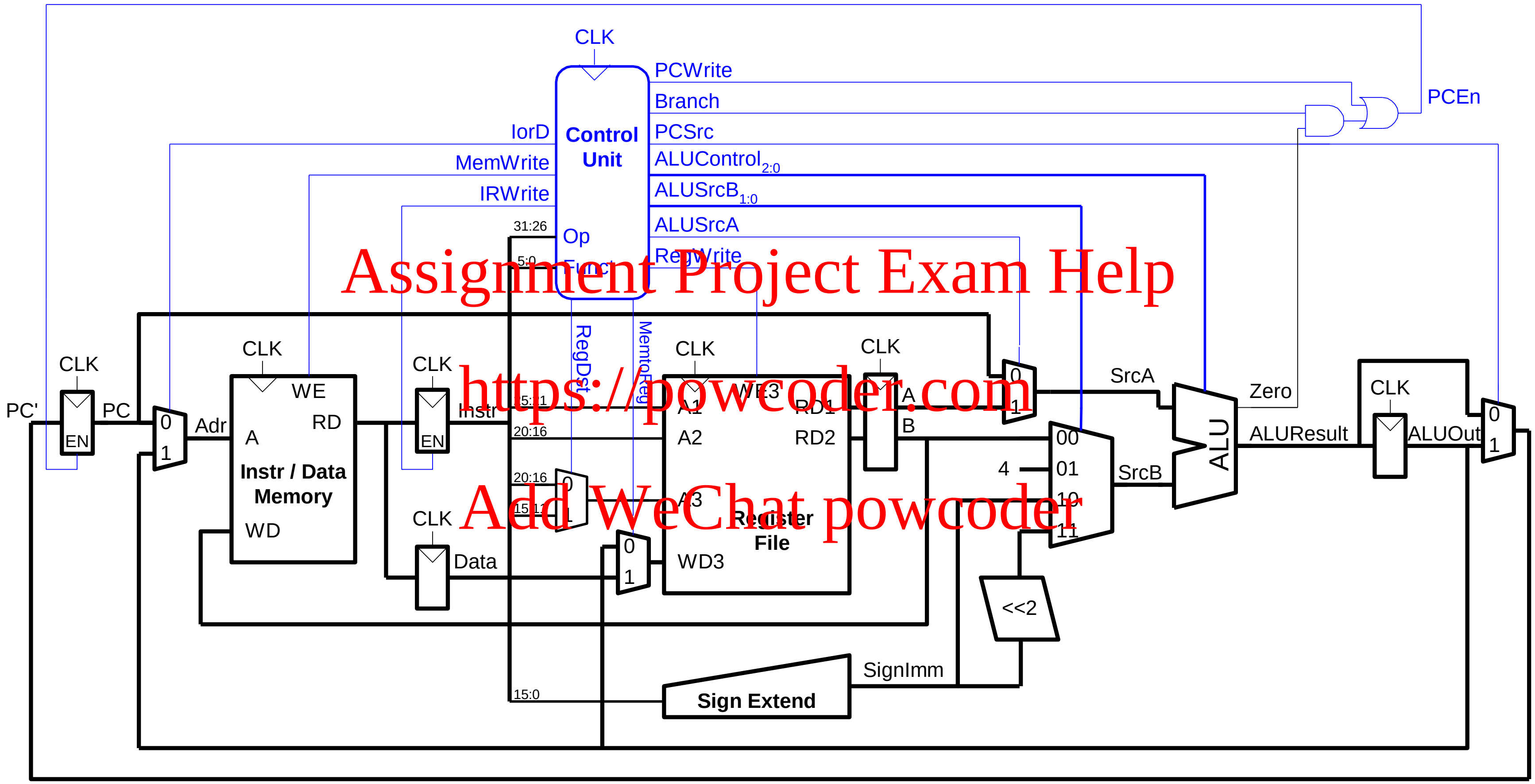
# Multicycle Datapath: sw

Example: **`sw rt, imm(rs)`**

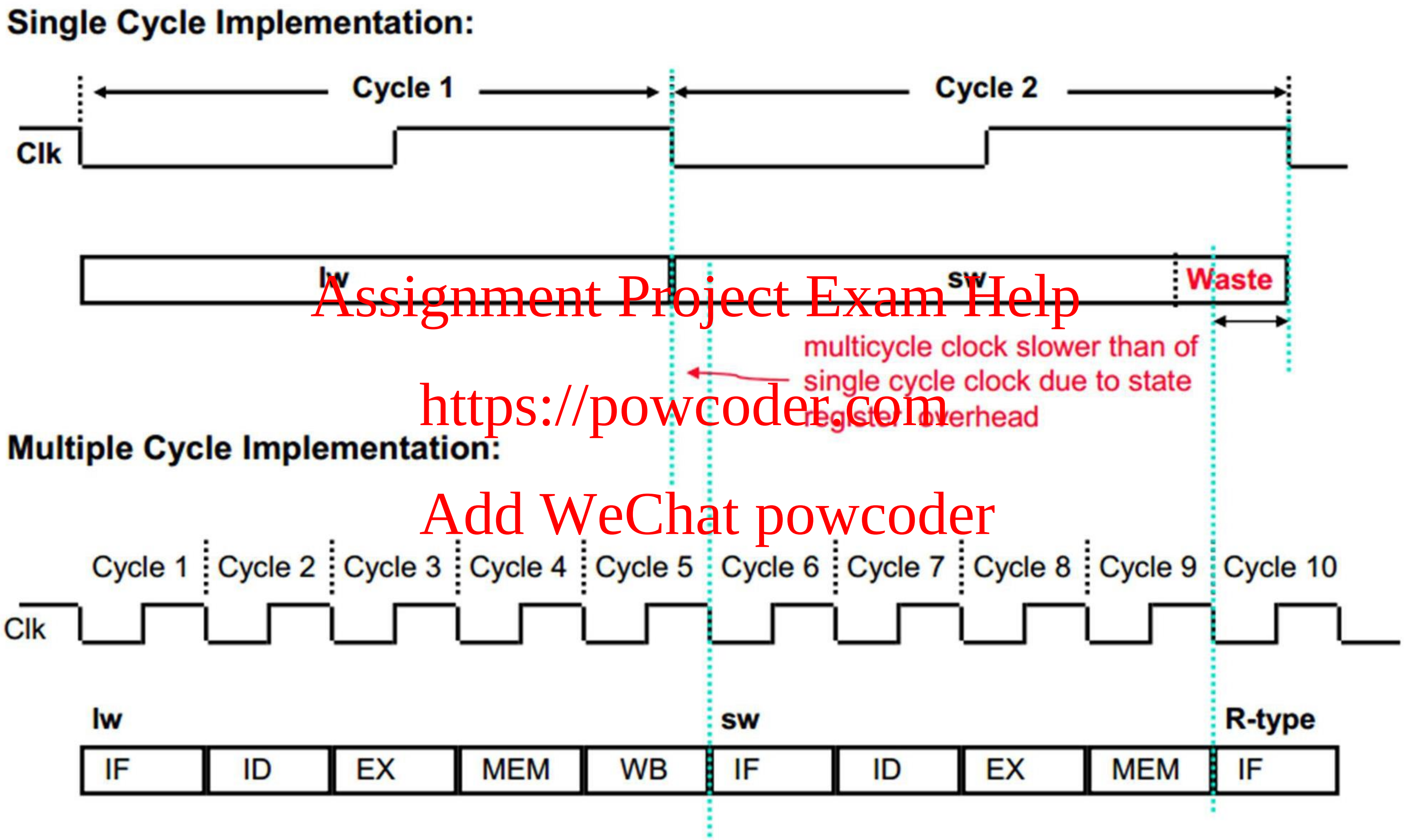
Write data in `rt` to memory



# Multicycle Processor



# Single Cycle vs. Multiple Cycle Timing


Single Cycle Implementation:
Cycle 1
Cycle 2
Clk
lw
sw
Waste
multicycle clock slower than of single cycle clock due to state register overhead
Multiple Cycle Implementation:
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Cycle 10
Clk
lw
sw
R-type
IF
ID
EX
MEM
WB
IF
ID
EX
MEM
IF

# Pipelined Processor

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- Today, pipelining is nearly universal.
- MIPS Instruction Set is designed for pipelining.
  - All instructions are of the same length – 32 bits
  - Easier to fetch and decode in one cycle

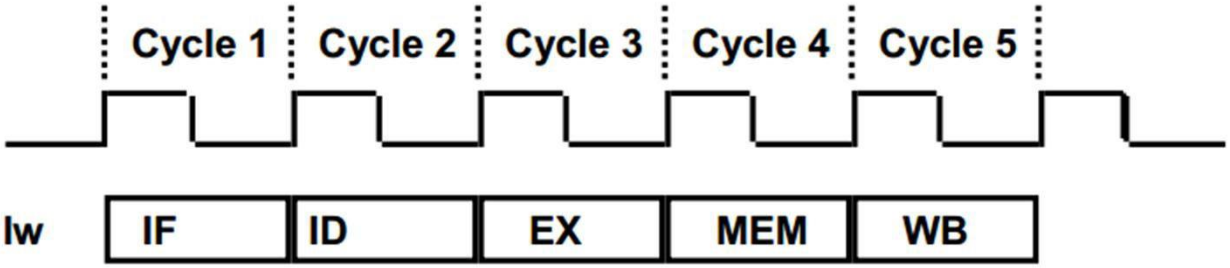
# Pipelined Processor

- Recall that an instruction can be executed in stages (with each stage partially-completed). This is seen previously in single-cycle and multi-cycle processors.
    - IF: Instruction Fetch, Increment PC
    - ID: Instruction Decode, Read Registers
    - EX: Execution (ALU)
  - Load/Store: Calculate Address Others: Perform Operation

    - MEM:
  - Load: Read Data from Memory Store: Write Data to Memory

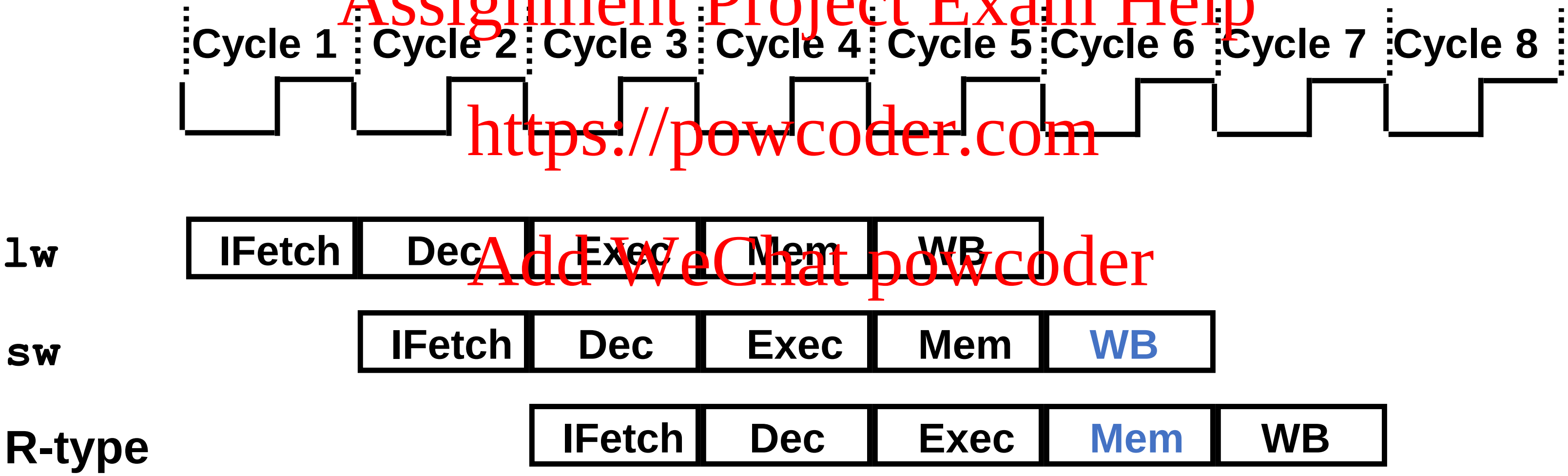
    - WB: Write the result (Data) Back to Register file

- The term "stages" implies datapath resources at each stage



# Pipelined Processor

- Pipelined processor start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced

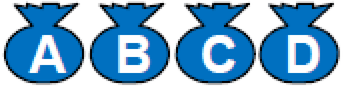


|  | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 |
|---|---|---|---|---|---|---|---|---|
| lw | IFetch | Dec | Exec | Mem | WB |  |  |  |
| sw |  | IFetch | Dec | Exec | Mem | WB |  |  |
| R-type |  |  | IFetch | Dec | Exec | Mem | WB |  |

clock cycle (pipeline stage time) is limited by the slowest stage
for some instructions, some stages are wasted cycles

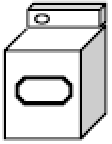
# Pipeline Analogy: Doing Laundry

- Andy, Benny, Charles, and Denny each have one load of clothes to wash, dry, fold, and put away

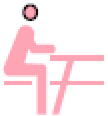


  - Washer takes 30 minutes

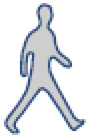
  - Dryer takes 30 minutes



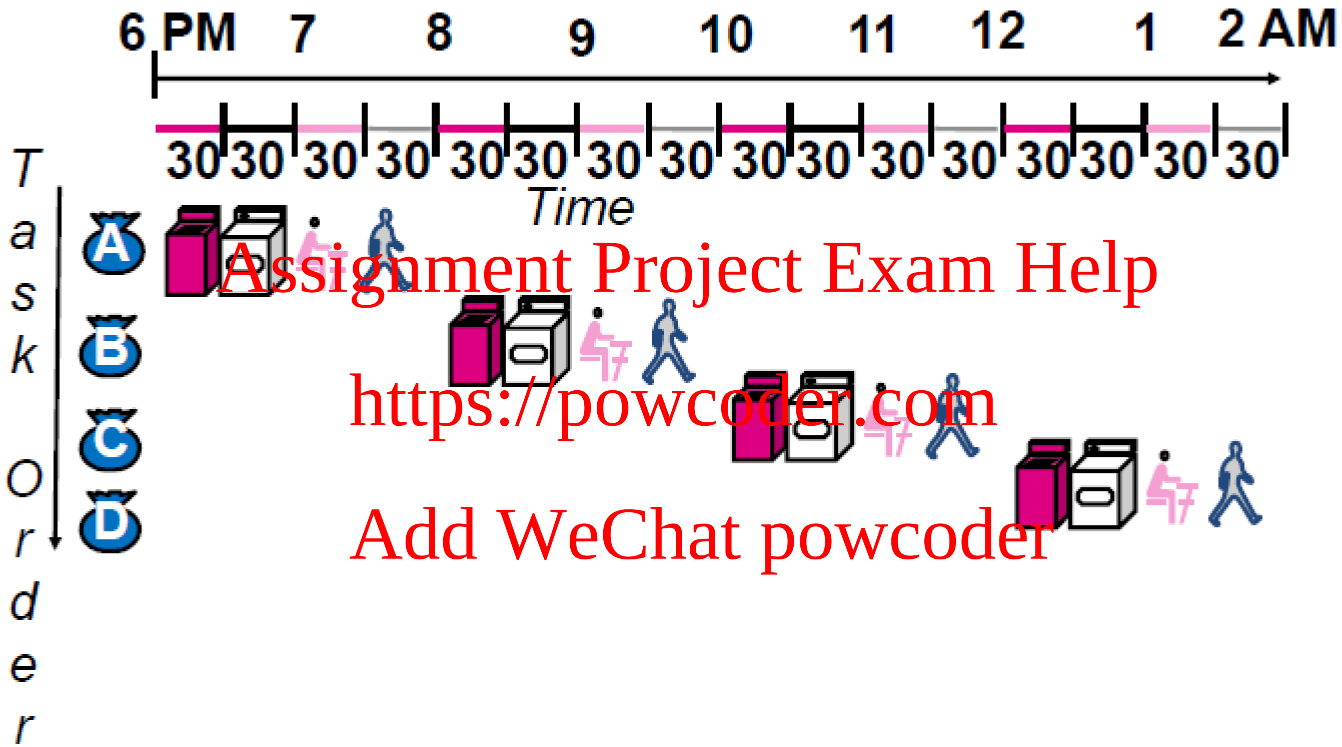
  - “Folder” takes 30 minutes



  - “Stasher” takes 30 minutes to put clothes into drawers

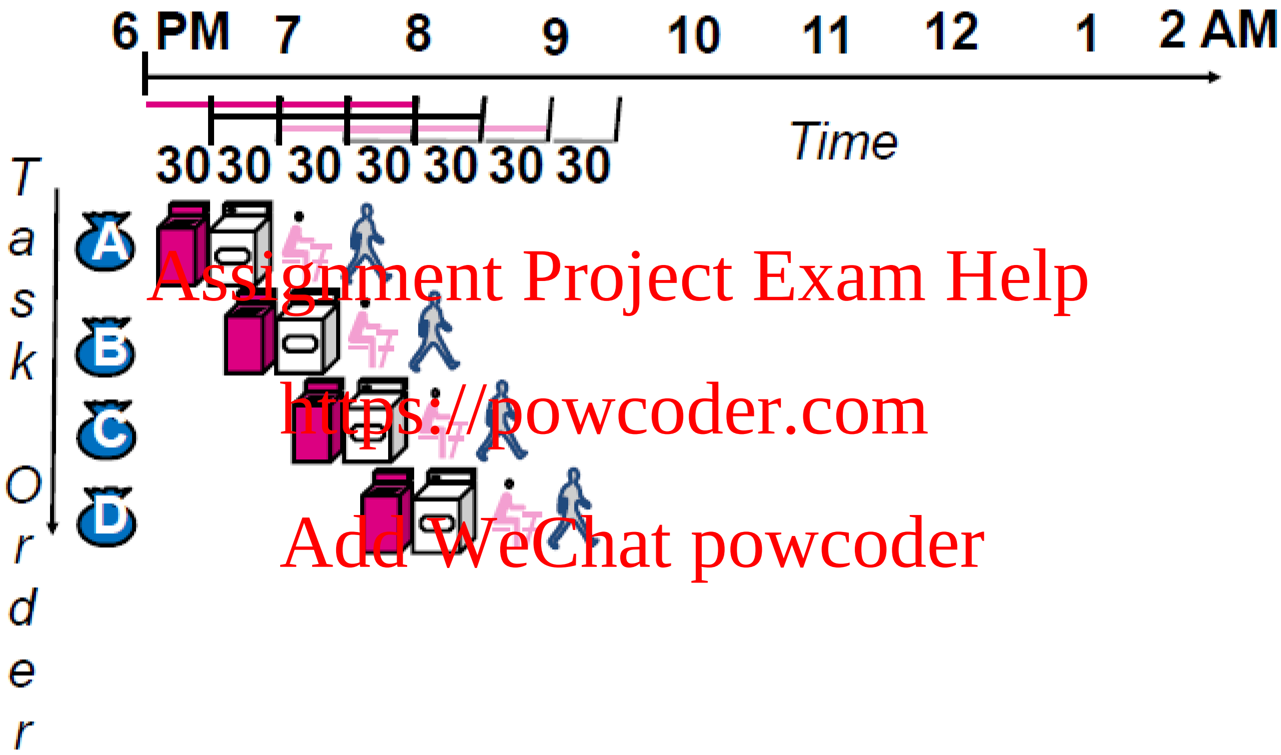


# Pipeline Analogy: Doing Laundry



- Sequential laundry takes 8 hours for 4 loads

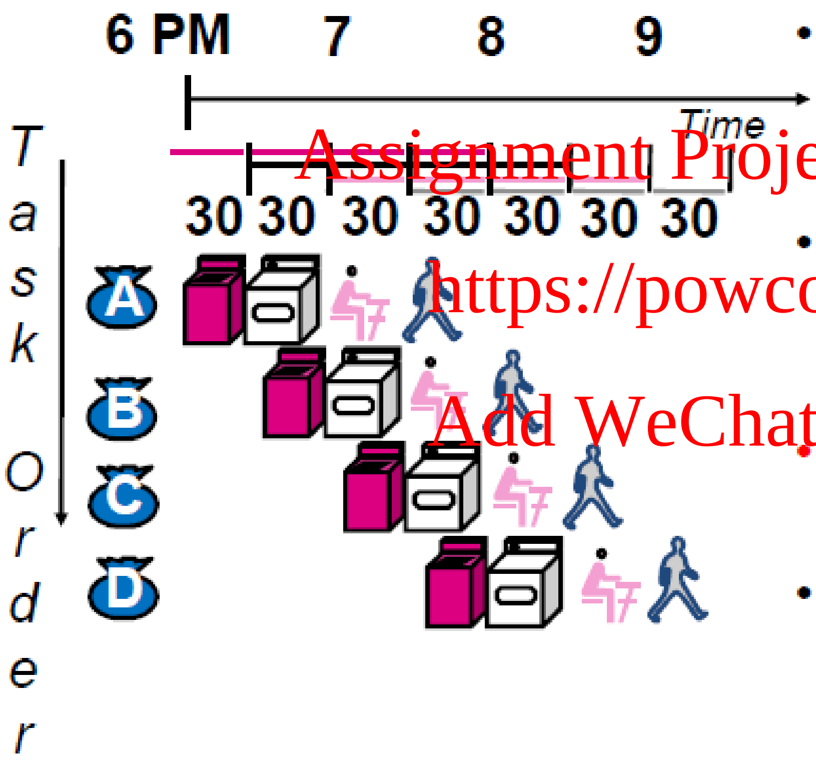
# Pipeline Analogy: Doing Laundry



- *Pipelined laundry takes 3.5 hours for 4 loads!*

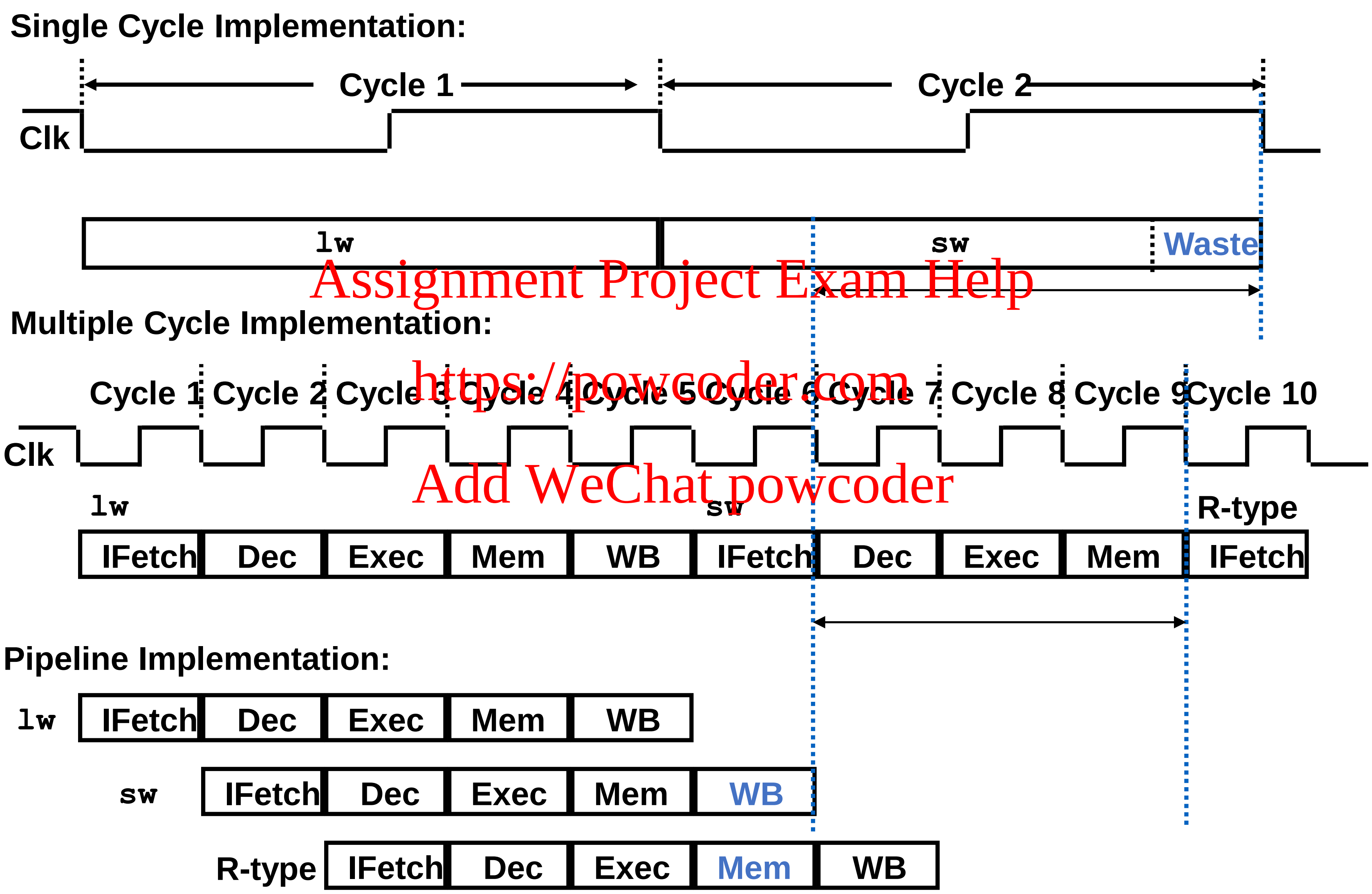
# Pipeline Analogy: Doing Laundry

## Pipelining Lessons:



- Pipelining doesn't help *latency* of single task, just *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup = number of pipeline stages
- Speedup reduced by time to *fill* and *drain* the pipeline: 8 hours/3.5 hours or 2.3X v. potential 4X in this example

# Single Cycle, Multiple Cycle, vs. Pipeline



Single Cycle Implementation:

Multiple Cycle Implementation:

lw: IFetch | Dec | Exec | Mem | WB

sw: IFetch | Dec | Exec | Mem

R-type: IFetch

Pipeline Implementation:

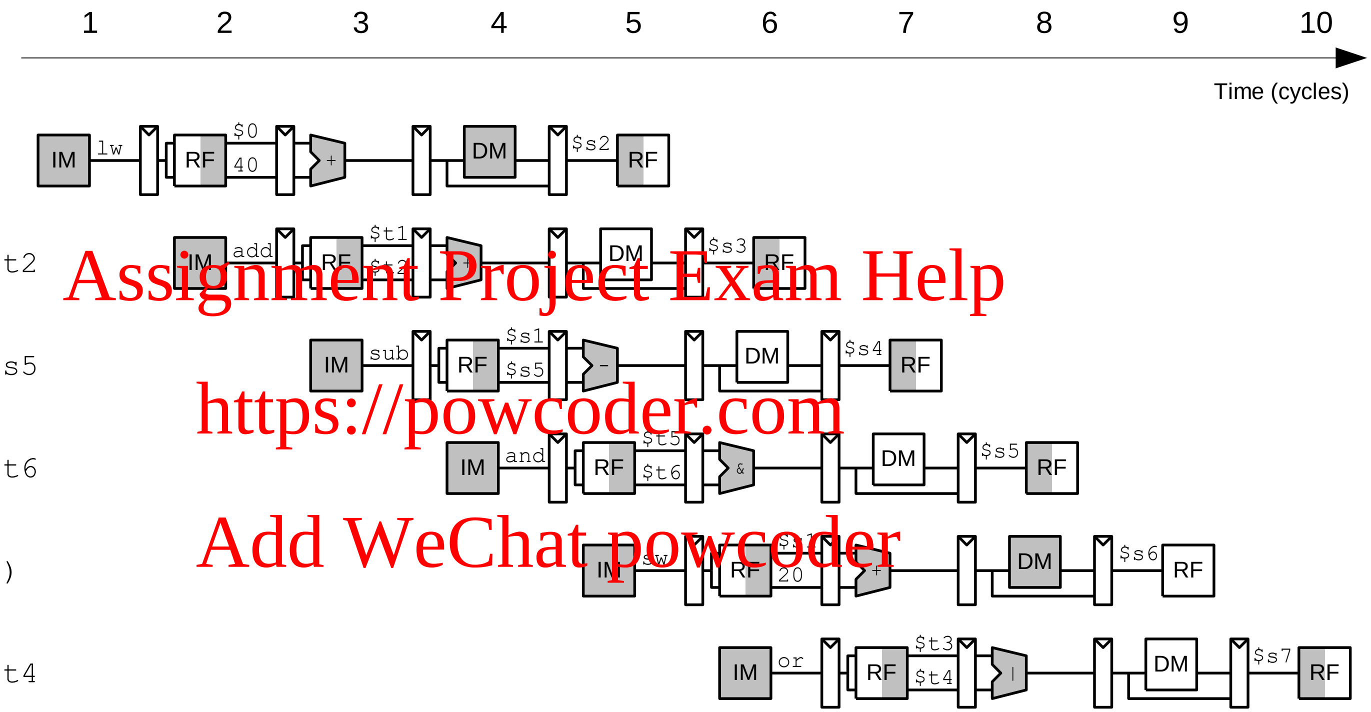
lw: IFetch | Dec | Exec | Mem | WB

sw: IFetch | Dec | Exec | Mem | WB

R-type: IFetch | Dec | Exec | Mem | WB

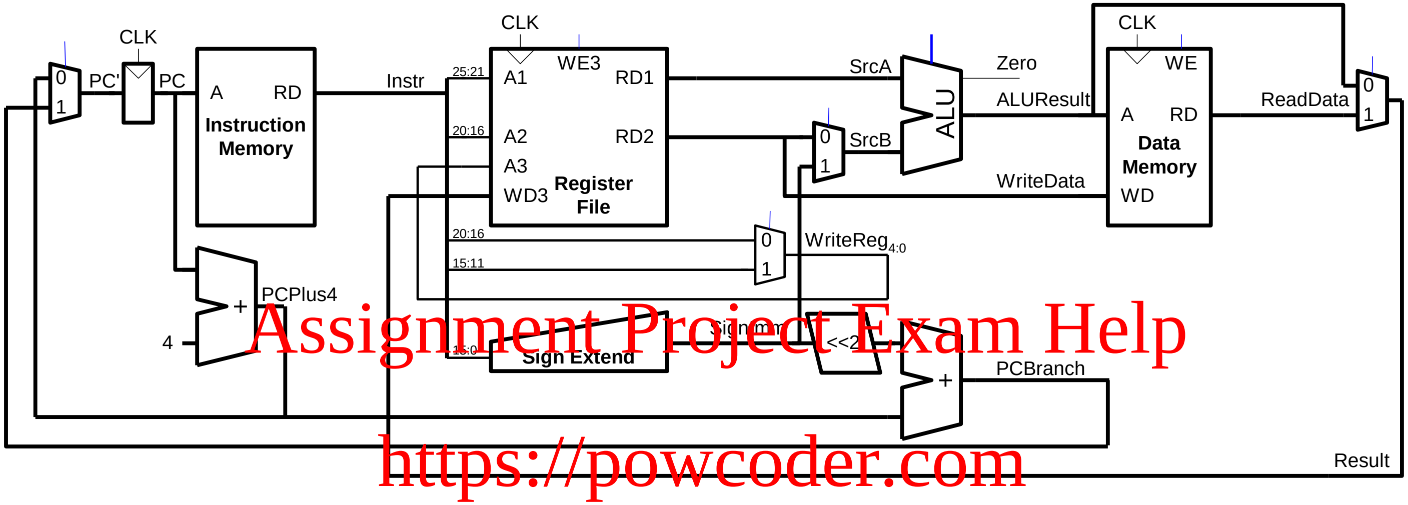
# Pipelined Processor

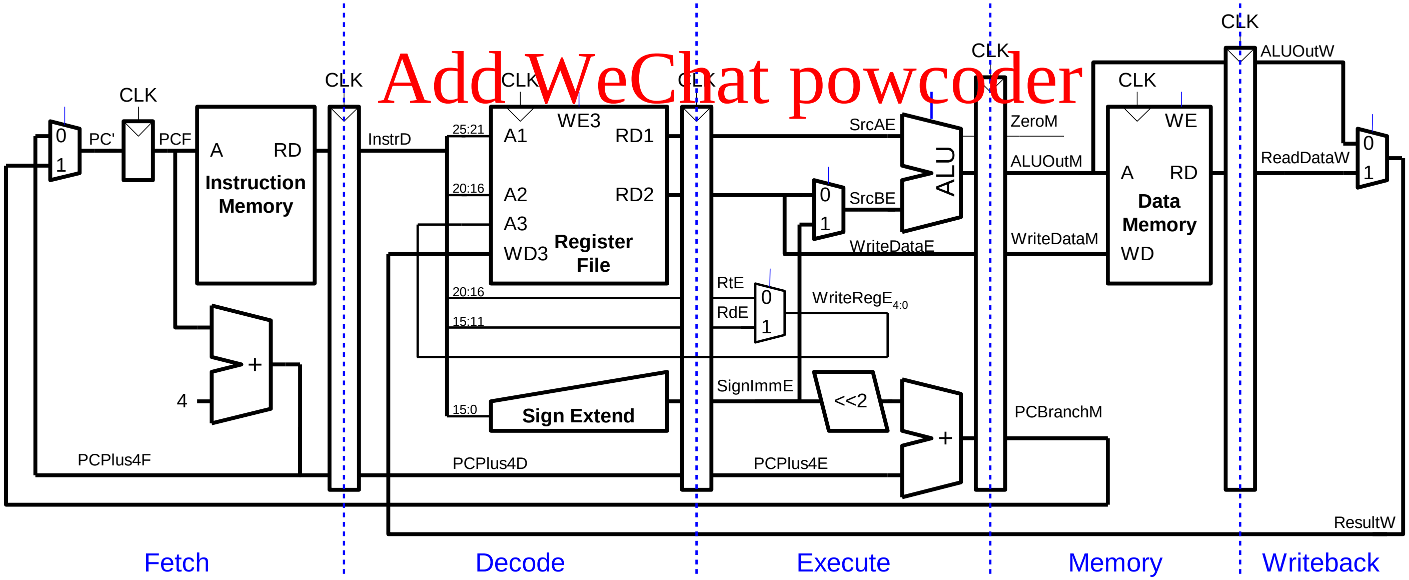
```
lw   $s2, 40($0)

add $s3, $t1, $t2

sub $s4, $s1, $s5

and $s5, $t5, $t6

sw   $s6, 20($s1)

or   $s7, $t3, $t4
```



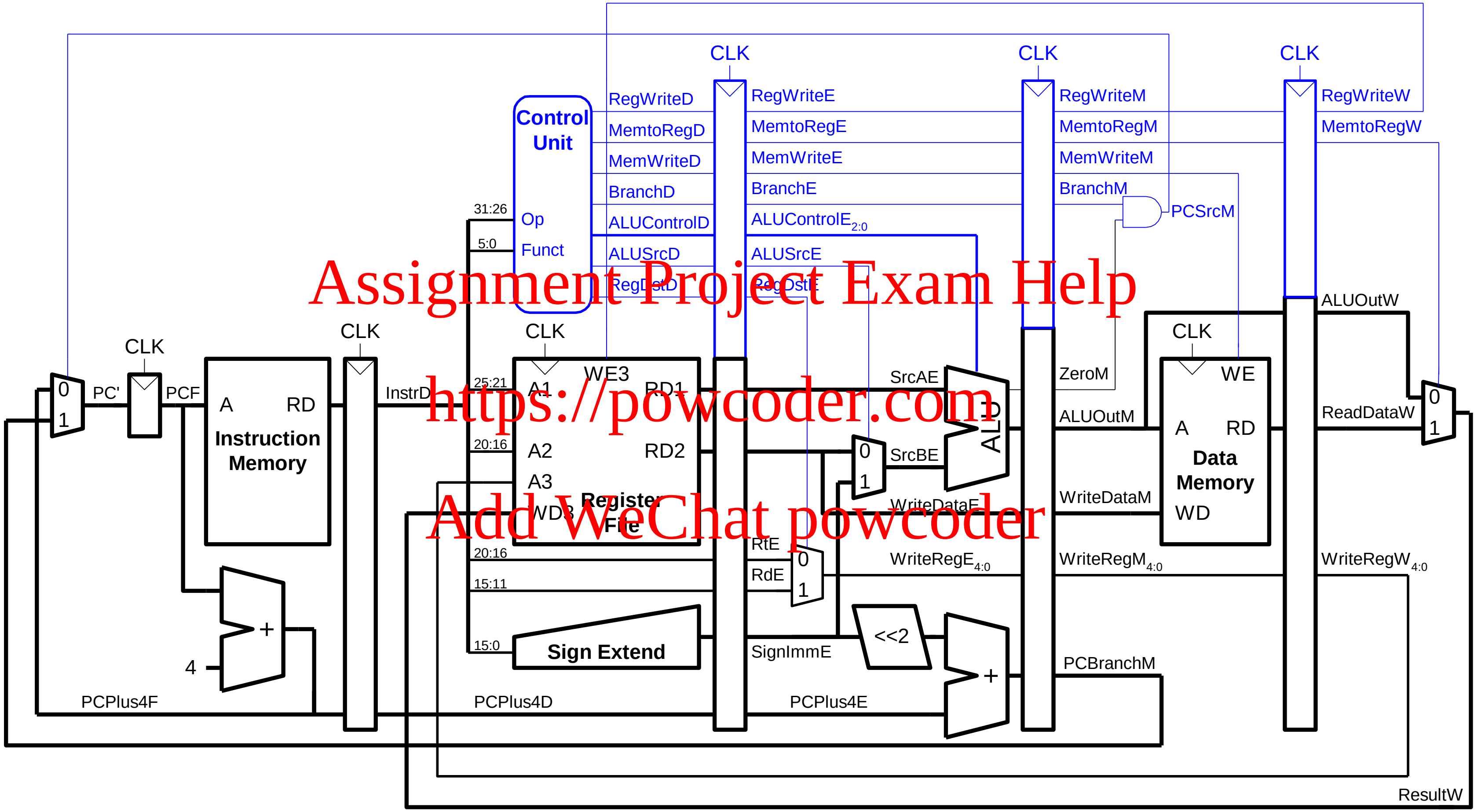
Resource usage

# Single-Cycle vs. Pipelined Datapath





# Pipelined Processor Control



# Pipeline Performance

- Use $T_c$ ("time between completion of instructions") to measure speedup

$$T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$$

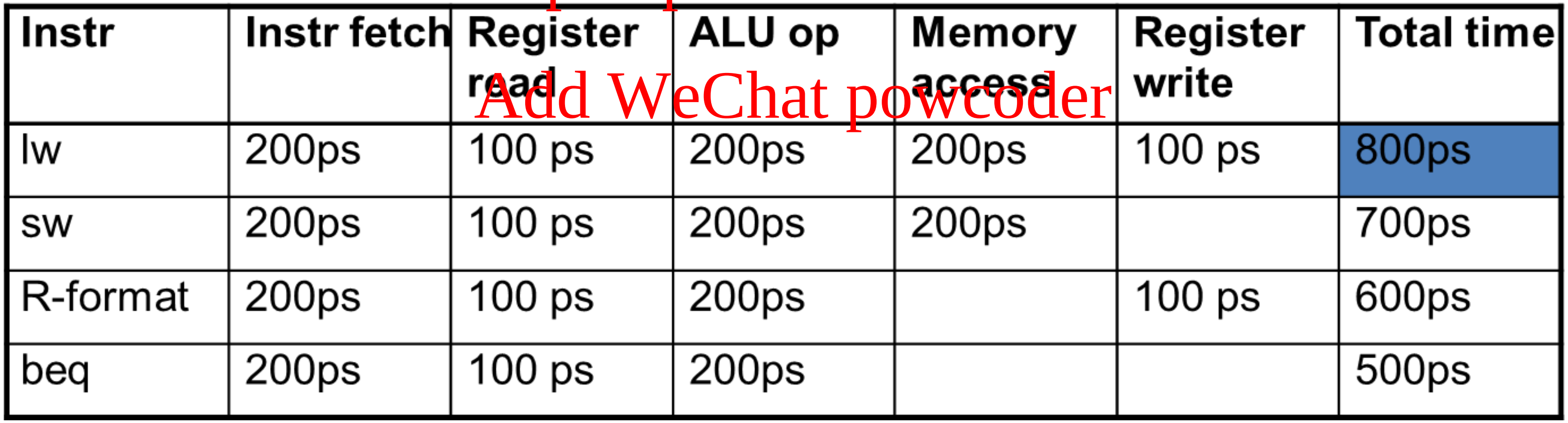
– Equality only achieved if stages are *balance* (i.e. take the same amount of time)

- If not balanced, speedup is reduced
  - Speedup due to increased *throughput*
    - *Latency* for each instruction does not decrease

# Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath



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

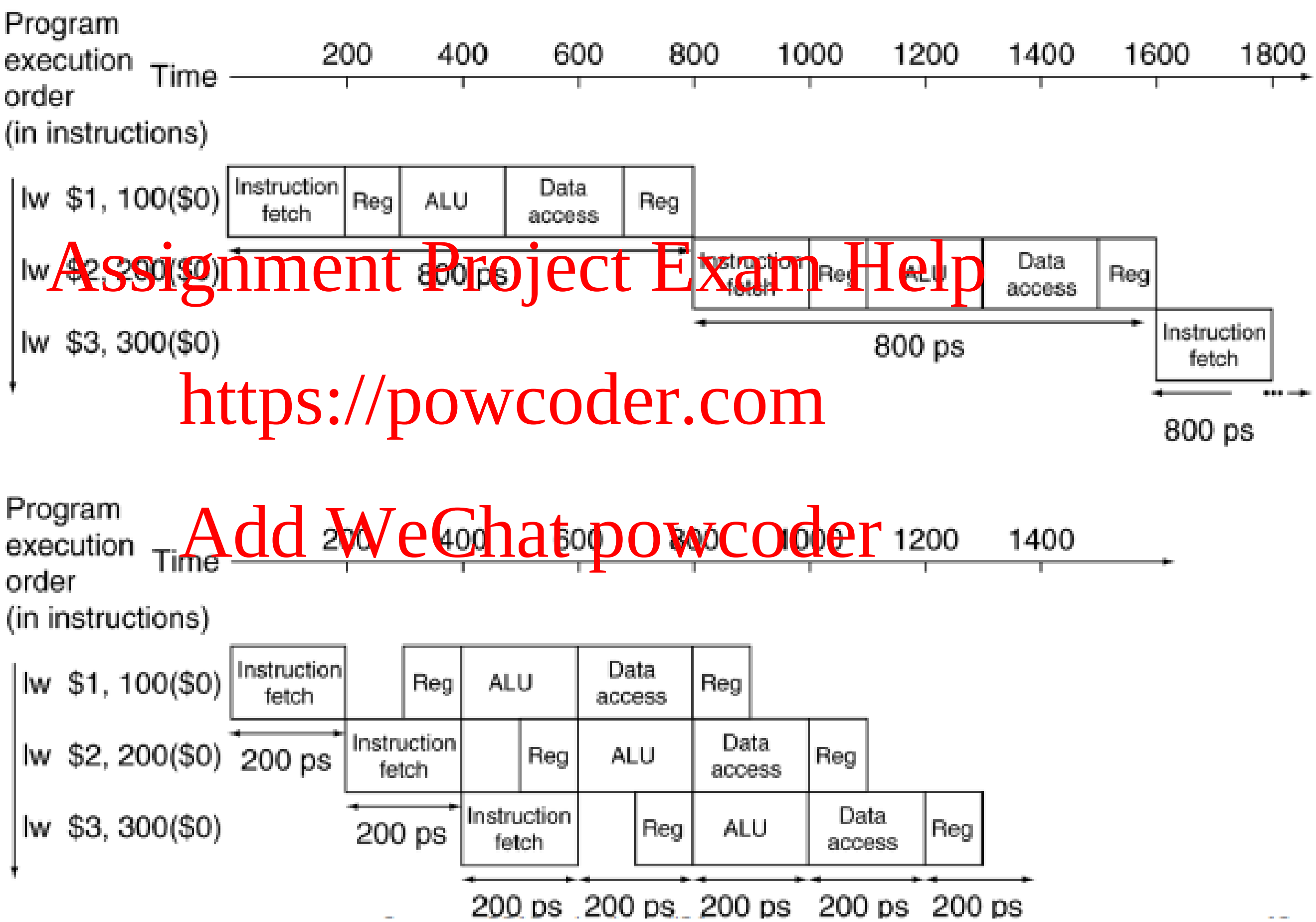
# Pipeline Performance

Single-cycle

$T_c$ = 800 ps

Pipelined

$T_c$ = 200 ps



# Pipeline Hazards

- Problem for pipeline:
  - instruction depends on result from instruction that hasn’t completed

- Types:
  - **Structural hazard:** attempt to use the same resource (hardware unit) two different ways at the same time
    - E.g., two instructions try to read the same memory at the same time.

  - **Data hazard:** register value not yet written back to register file
    - E.g.,
      ```
      add $r1, $r2, $r3
      sub $r4, $r2, $r1
      ```

  - **Control hazard:** next instruction not decided yet (caused by branches)
    - E.g.,
      ```
      beq $r1, $r2, loop
      add $r3, $r4, $r5
      ```

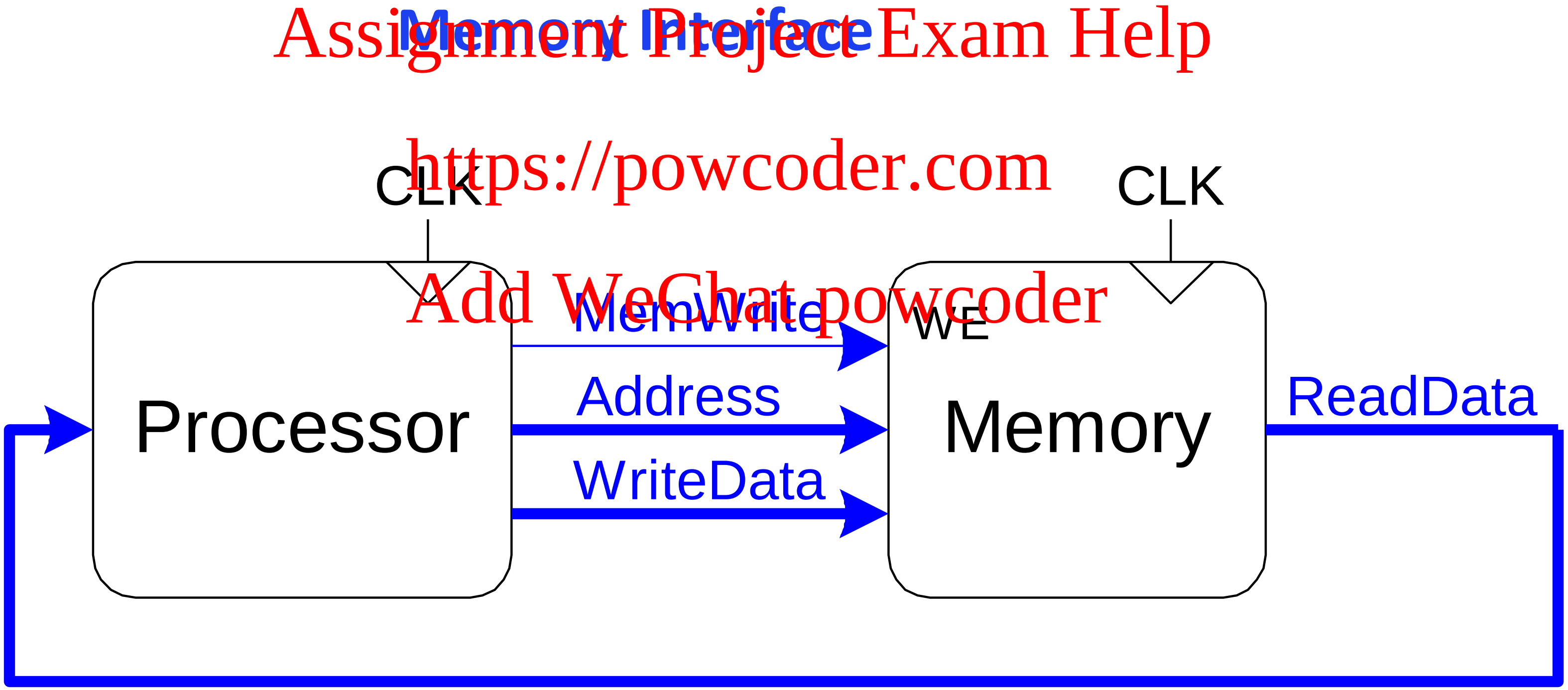
- Pipeline control must detect hazard
  - Resolve hazards by waiting / delay action

Chapter 6-3

# Memory system and hierarchy

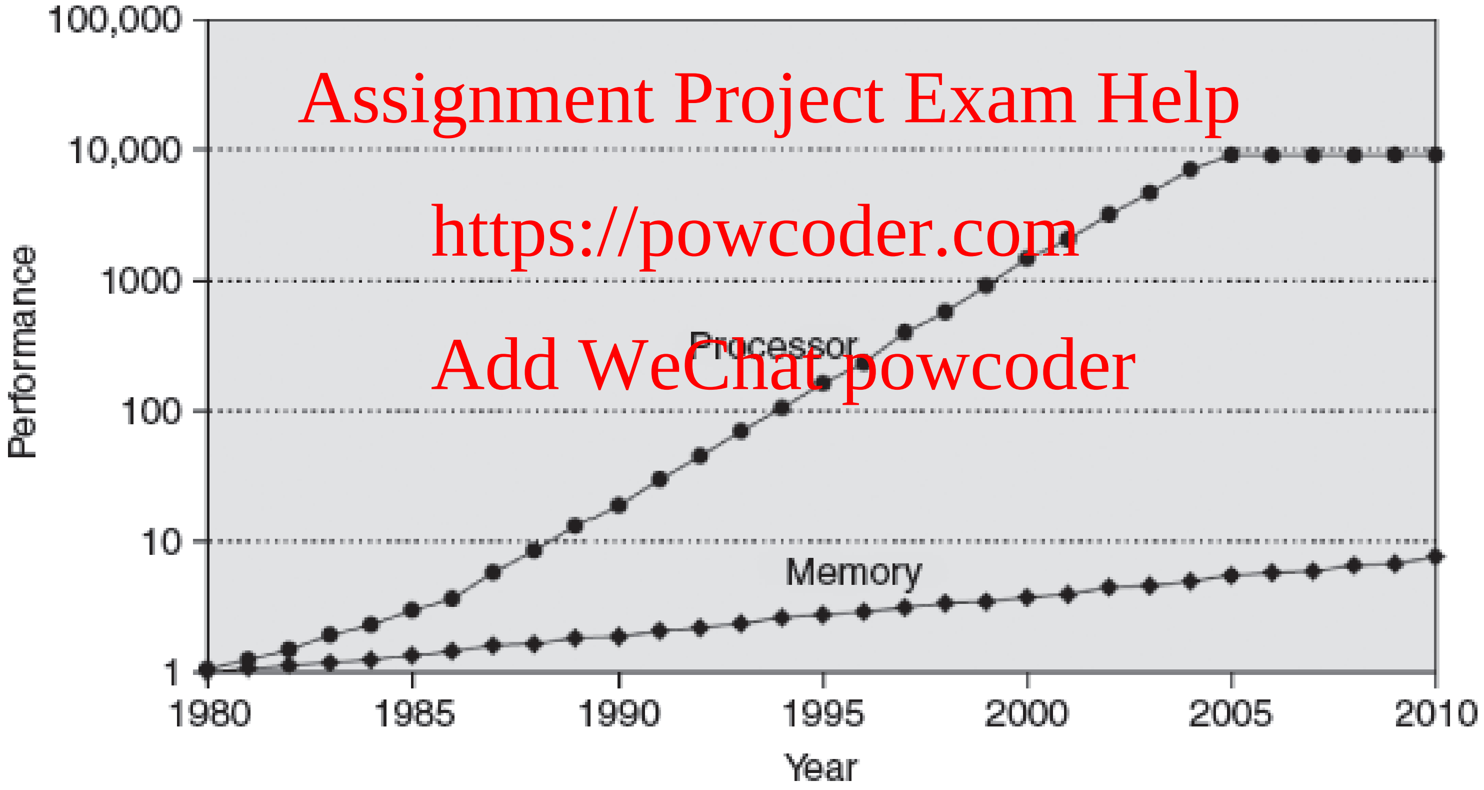
# Introduction

- Computer performance depends on:
  - Processor performance
  - Memory system performance



# Processor-Memory Gap

In prior chapters, assumed access memory in 1 clock cycle – but hasn't been true since the 1980's

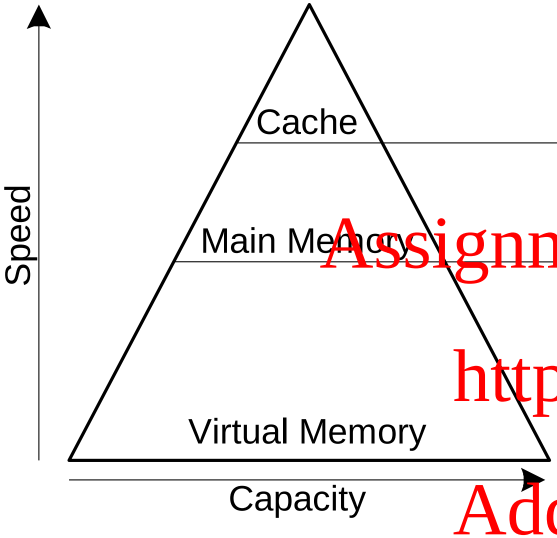


# Memory System Challenge

- Make memory system appear as fast as processor
- Use hierarchy of memories
- Ideal memory:
  - Fast
  - Cheap (inexpensive)
  - Large (capacity)

But can only choose two!

# Memory Hierarchy

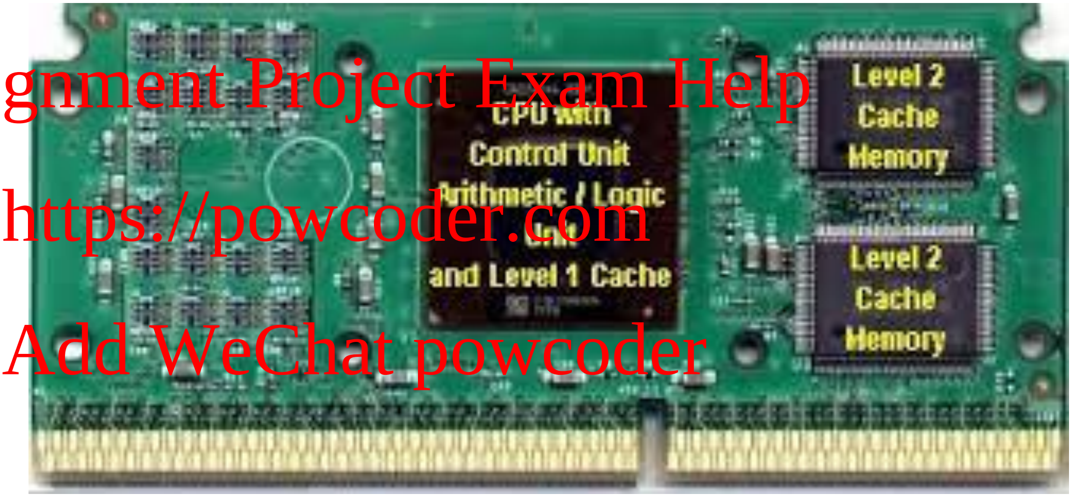


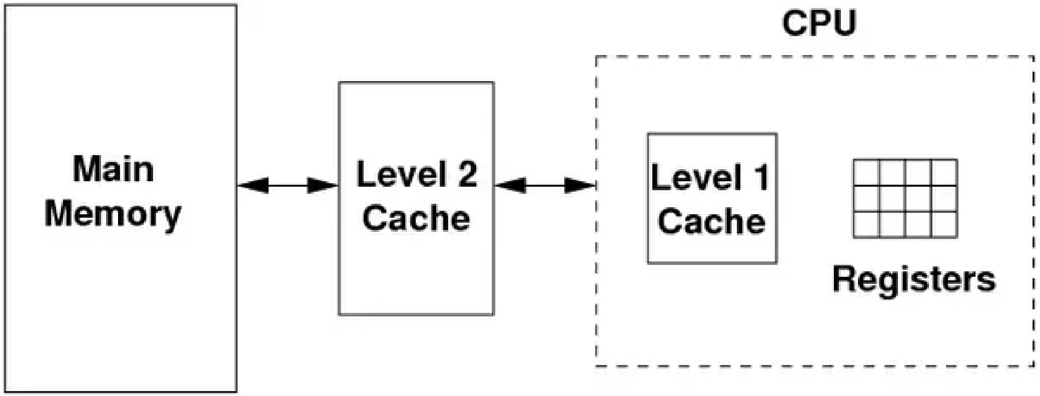
| Technology | Price / GB | Access Time (ns) | Bandwidth (GB/s) |
|---|---|---|---|
| SRAM | $10,000 | 1 | 25+ |
| DRAM | $10 | 10 - 50 | 10 |
| SSD | $1 | 100,000 | 0.5 |
| HDD | $0.1 | 10,000,000 | 0.1 |

- **Cache:** SRAM
- **Physical Memory:** DRAM (Main Memory)
- **Virtual Memory:** Hard drive
  - Slow, Large, Cheap

# Cache

- Highest level in memory hierarchy
- Fast (typically ~ 1 cycle access time)
- Ideally supplies most data to processor
- Usually holds most recently accessed data





# Memory Type

- Types of memory:
  - Random access memory (RAM)
  - Read only memory (ROM)

- RAM is volatile – loses its data when power off.
  - Historically called random access memory because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)
  - The main memory in your computer is dynamic random access memory (DRAM).
  - DRAM stores data using capacitor

- ROM is nonvolatile – it retains data when power off
  - Flash memory in cameras, thumb drives, and digital cameras are all ROMs.
  - Historically called read only memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.

# Virtual Memory

- Gives the illusion of bigger memory
- Main memory (DRAM) acts as cache for hard disk

Hard disk takes milliseconds to seek the correct location on disk

# Principle of Locality

Exploit locality to make memory accesses fast

- **Temporal Locality:**
  - Locality in time
  - If data used recently, likely to use it again soon
  - **How to exploit:** keep recently accessed data in higher levels of memory hierarchy
  - E.g., instructions in a loop.
- **Spatial Locality:**
  - Locality in space
  - If data used recently, likely to use nearby data soon
  - **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too
  - E.g., Sequential instruction access, array data.

# Taking advantage of Locality

- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU