ICS53 - MIPS Instruction Set Architecture & Single-Cycle Datapath

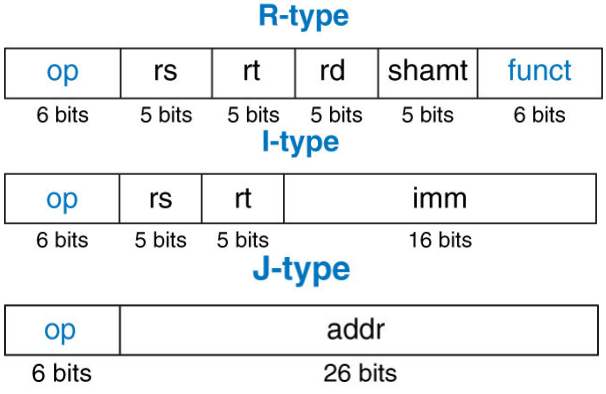
Big Picture Overview

- In the previous modules we studied how to build the basic building blocks of the datapath, such as multiplexors and the ALU.
- In this module, we will examine how to connect these building blocks together to build a working processor and how to control the movement of data through the datapath (control signals). The connection of the components together creates the ***Processor Datapath***.
- When designing a processor datapath, the goal is:
  - To use the minimal amount of hardware (cost).
  - To execute the instructions (specified by ISA) efficiently (best performance).

- In this module we discuss the simplest version of the MIPS ISA implementation - single-cycle, and only with a small subset of the MIPS instructions:
  - Memory instructions: load (lw), store (sw)
  - ALU instructions: add, sub, and, or, slt
  - Control instructions: branch on equal (beq), jump (j)

MIPS Instruction Set Architecture (MIPS ISA) – Recap/Summary

- This is where the knowledge of the MIPS assembly language and the architecture theory come together.
- A few important points to remember:
  - MIPS is a load-store architecture, which means only load and store operations access memory.
  - MIPS is a register machine (32 general purpose registers), which means the processor only performs operations on the data in the registers.
  - MIPS uses 32 general purpose registers (plus 32 separate floating-point registers, but we have not discussed/used these). Registers are fast memory located within the CPU datapath and are costly compared to other types of memory, which is why we have a limited number.
    - 32 Registers means we have 2^5 possible values and need 5 bits to represent in binary
  - All MIPS instructions are a fixed of length 32-bits.
  - All MIPS instructions are formatted with one of 3 instruction formats: Register, Immediate, Jump.



op: operation identifier (instruction), "opcode"
rs: 1st register source operand
rt: 2nd register source operand or destination operand
rd: register destination operand
shamt: shift amount, at most 31 bits
funct: function; specifies specific variant of opcode; aka *function code*.
imm: immediate; 2's complement value
addr: lower bits of address for jump

Instruction Format Recap

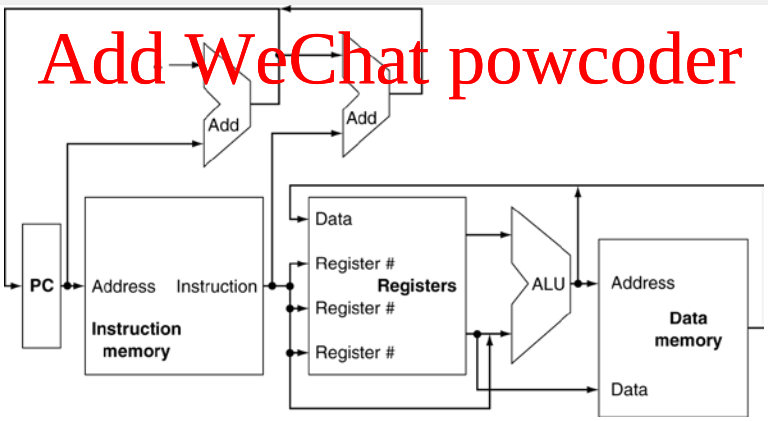
- The assembly code we have been writing are basic instructions which use pneumonic "programmer friendly" naming conventions to reference the operation and the registers which are being used by each instruction.
  - Each of these instructions is replaced by 1 or more assembly instructions when *assembled* by the *assembler*.
  - Each assembled instruction is an instruction which is directly implemented by the datapath. We will call these "hardware instructions".
  - Each hardware instruction is represented by 32-bit binary, also referred to as *bytecode*. Ex: hello world in MIPS assembly:

```
00111100000000010001000000000001
00110100001001000000000000000000
00100000000000100000000000000100
00000000000000000000000000001100
00100000000000100000000000001010
00000000000000000000000000001100
```

- Each instruction is stored in memory with an address associated with its position. MIPS memory is byte addressable, which means that each byte (8bits) has a unique address. MIPS instructions are each 32-bits, therefore each address must be a multiple of 4.
- A MIPS program is made of a sequence of hardware instructions (stored in the .text section of memory) and a set of data values (stored in the .data section of memory)

Abstract/Basic MIPS Single-Cycle Architecture

Single-Cycle Datapath - Overview



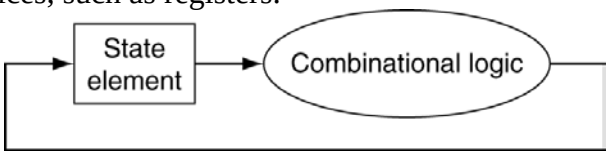
- The basic datapath is composed of the following main components:
  - PC, Program counter (the address of the instruction to execute)
  - Instruction memory (the .text section, program instructions)
  - Register File (32 32-bit registers, 2 read registers, 1 write register)
  - ALU (address calculation and operand arithmetic)
  - Data memory (the .data section, the data values)
  - In addition, there are two Adders used to calculate the address of the next instruction to execute (all instructions) and for the calculation of the address to branch to (branch instructions)

- The name single-cycle datapath comes from the fact that each hardware instruction is executed in **1 clock cycle of the processor**.

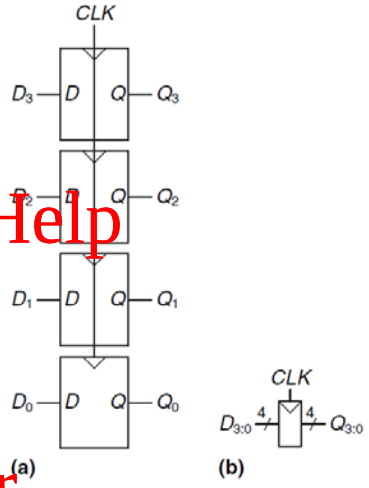
Clock & Clock Cycle

What is the Clock? What is a clock cycle?

- Each of the digital circuits studied so far were combinational logic. This means when the values are supplied to the inputs of the gates the gates will output the computation after the associated delay of the gate. The value will continue to output as long as the input(s) do not change value.
  - Think about our critical path computations. We calculated based on the assumption that the input values were available at time 0 and never change values.
- To ensure the inputs stay constant while the combinational logic (gate network) is computed, the input values are stored in state element devices, such as registers.



- Registers are memory storage devices which hold a value constant until instructed to change. They are a component of *sequential logic*.
  - Registers are built from bit-saving units called Flip-Flops (FF). There are different varieties: SR, JK, and D.
  - Each FF is controlled by a Clock (CLK). The value stored in the FF will only change on the edge of the clock (transition between a clock cycle to the next).
  - D FFs are the simplest. The D stands for "delay". The value on the input of the FF will be stored on the CLK edge. The value is stored and consistently appears on the output until the CLK edge occurs again.
  - FFs are chained together to create Registers which hold n-bit values (see figure).

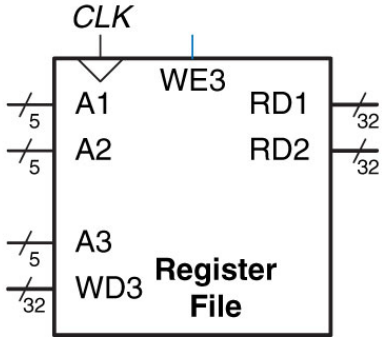


Flip-Flop Explanation

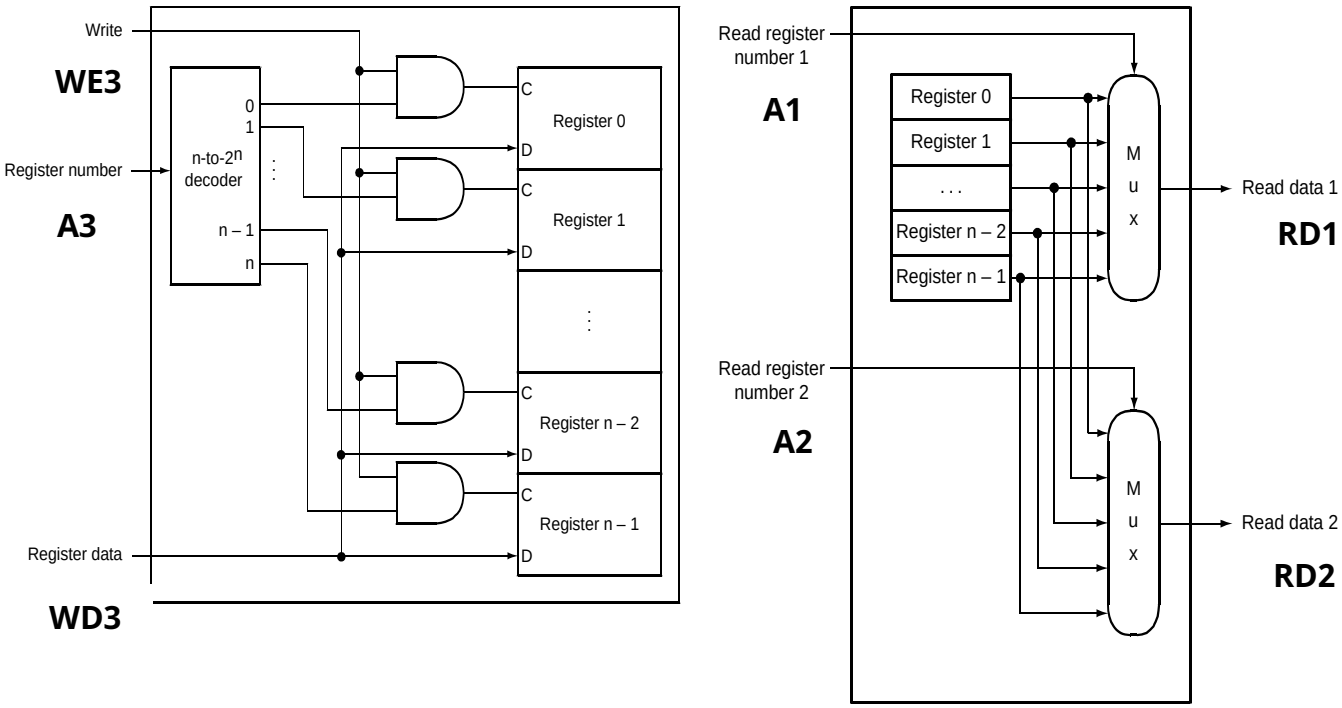
- On each clock cycle, the single cycle datapath will take the outputs of the registers and compute all of the combinational logic to execute the instruction. At the end of the clock cycle, the computed values are stored into the registers to complete the execution.
  - The length of the clock cycle is pre-determined and is based on the critical path of the datapath (more on this in terms of the Single-Cycle Datapath later)

Register File

- The register file is a unit which holds all 32 general purpose registers for the single cycle datapath. It is constructed in hardware as parts, Read and Write.

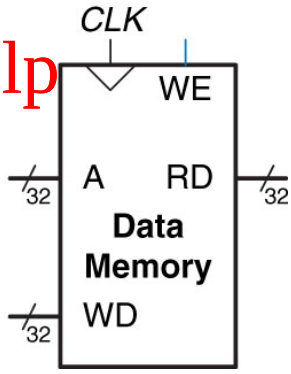


Register File Explanation



Instruction & Data Memory

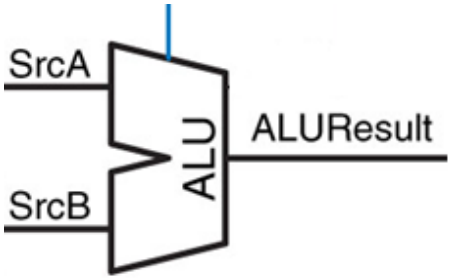
- The instruction and data memory units are the main memory of the processor.
- Each unit takes in a single 32-bit address and returns the 32-bit value stored at the address.
- The data memory has the additional functionality of writing (storing) data into memory. The unit is controlled by a write-edge (WE) signal and a the clock (CLK). The data is



Memory Unit Explanation

ALU Unit

- The 32-bit ALU within the single-cycle datapath is built using the concepts discussed in the previous module. Specifically, the following control signals are used to support the operations (function) specified in the table.
  - SrcA and SrcB are both 32-bit values
  - ALUResult is the calculated 32-bit result
  - 3-bits are used for the ALUControl



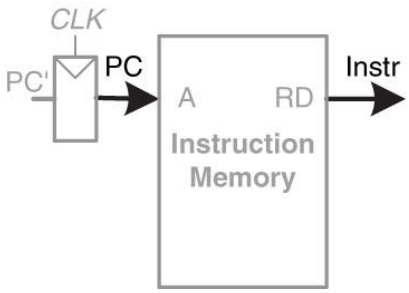
ALUControl	Operation
000	AND
001	OR
010	add
110	subtract
111	SLT

Fetch- Decode-Execute Cycle

Fetch-Decode-Execute Explanation

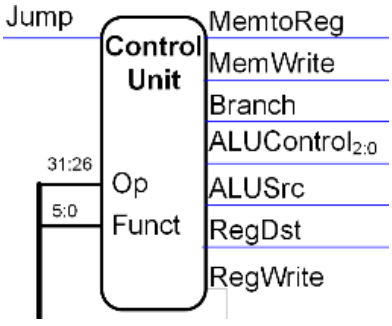
- The execution of an instruction in the datapath can be split into three phases: Fetch-Decode-Execute.

Instruction Fetch Stage



- The bits of the instruction to be executed during the clock cycle must first be fetched from the instruction memory
  - The PC (program counter) contains the address of the current instruction to execute
- When the execution of this instruction is completed, the address of the next instruction to execute must be stored into the PC.
  - PC = PC + 4
    - We increment the PC to point to the next instruction to execute in the program.
    - Remember that memory is byte addressable, not Word addressable. Each machine word is 32-bits, but because we want to be able to refer to bytes (for things like ASCII characters in memory). Each byte is 8-bits, and therefore each word contains 4 bytes. Memory is byte addressable, meaning that each byte has a unique address. Since instructions are only in word size, then to refer to each word, we must increment the address by 4-bytes.
- This section of the datapath does not change depending on the instruction type. We ALWAYS fetch the instruction and increment the PC.
  - Why? A large majority of the time the next instruction in the program will be the next instruction. Only branch and jump instructions change the flow of execution. Extra hardware will be added to CHOOSE which address to store into the PC at the end of the clock cycle.

Decode Stage



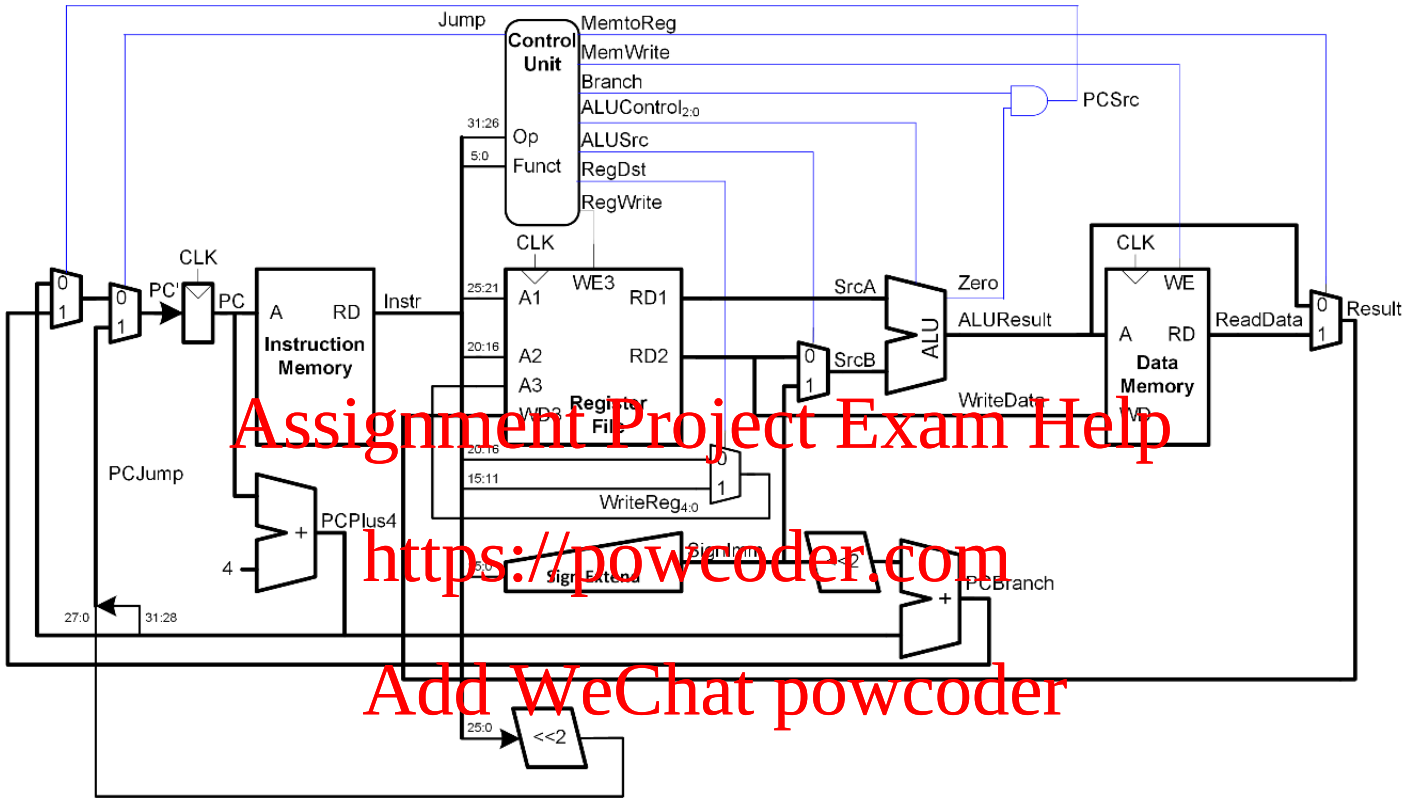
- The opcode bits of the instruction are used by the Control Unit (aka "the all-knowing brain" in Prof. Wong-Ma lingo) to determine which instruction is to be executed and to create all the control signals for all the units of the datapath.
  - In the case of R-type instructions, the function field is also needed to determine the operation to perform (more on this later).
- The output control lines of the datapath are shown in blue in the figures.
- The Control Unit is combinatorial logic – just like a unit we built in the earlier modules.
  - Each output is expressed as a boolean expression based on the opcode and function bits (inputs)
- In addition, since most instructions use values in the registers, while the control unit is working, the register file is read in parallel. When the control has determined the operation, the values are waiting to be used!

Execute Stage

- Once the instruction is decoded, the task of executing/performing all required operations for the instruction is completed by the units of the datapath based on the control signals. This is known as the Execute phase.

Single-Cycle Datapath

[Single Cycle Datapath Walk-through]



- The above datapath shows how the components are combined to perform the basic instructions (lw, sw, beq, j, R-type). We will study these 5 instructions in detail.
- Some specifics to take note of:
  - Bits of the instruction (specified by the instruction format: Register, Immediate, Jump) are sent to the main components (Register File, ALU, Data Memory).
  - The sign-extension (16 to 32 bits) is required because the ALU takes two 32-bit operands. The immediate instruction format only contains a 16-bit constant. The values must be sign extended (2's complement format) so the ALU operations can be performed.
  - The additional adder (PCBranch) is used to calculate the branch address.

Load & Store Instructions

- Load and store instructions are the only instructions which interact with the Data Memory.
  - They add an offset to the contents of a base register to calculate the memory address – Immediate format
  - The main difference is load reads from memory and store writes to memory.
- Ex:
  - `lw $t0, 8($s0)  # Reg[$t0] = Mem[Reg[$s0] + 8]`

- `sw $t1, 16($t2)` `# Mem[Reg[$t2] + 16] = Reg[$t1]`
  (Note: we use Array index notation to describe obtaining the value from memory or registers)
- The instructions can be broken up into the Fetch-Decode-Execute stages in the following way:

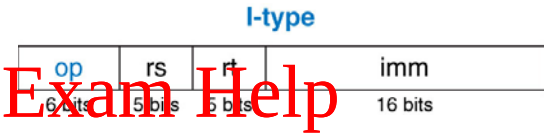
Load	Store
Fetch – Fetch Instruction	**Fetch** – Fetch Instruction
Decode - Register Read (rs: `$s0`), Sign Extend Immediate Value	**Decode** - Register Read (rt: `$t1`, rs: `$t2`), Sign Extend Immediate Value
Execute - Calculate Address using ALU (`Reg[$s0]+8`)	**Execute** - Calculate Address using ALU (`Reg[$t2]+16`)
Execute - Read from memory (`Mem[Reg[$s0] + 8]`)	**Execute** - Write to memory (`Mem[Reg[$t2] + 16] = Reg[$t1]`)
Execute - Write back to a register (rt: `$t0`)	

- We will step through the operation of each instruction separately

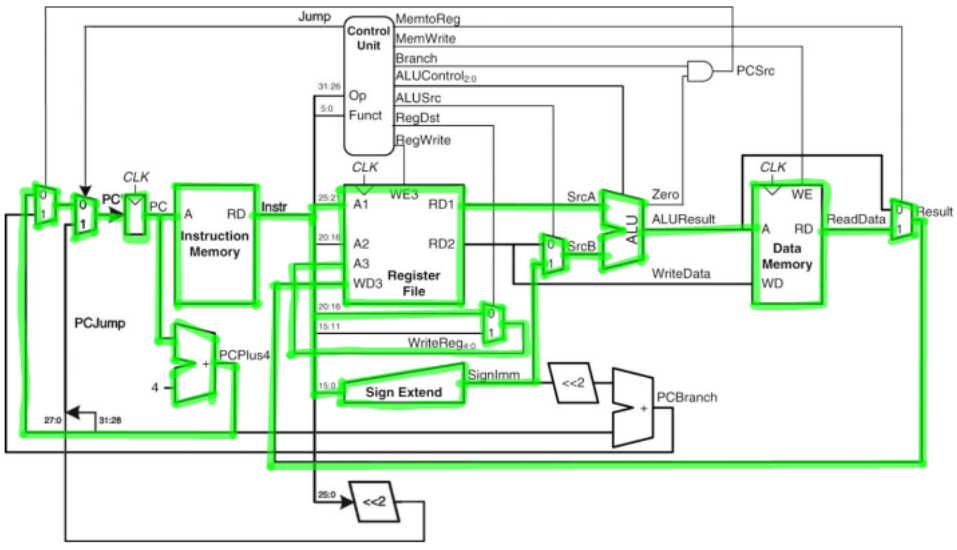
Load

Ex: `lw $t0, 8($s0)` `# Reg[$t0] = Mem[Reg[$s0] + 8]`

Load Instruction Walk-through



- The rs register (`$s0`) is read from the register file and the value in the register (`Reg[$s0]`) is available on Read data 1 (RD1).
- To this value the immediate value (after sign extension) is added using the ALU , Reg[$s0]+8
- This is now the address of the data in memory, the data memory is then read at this address and the value is output on the read data line of the data memory (`Mem[Reg[$s0] + 8]`).
- This value is stored back in the register file (`Reg[$t0]`). The data to store is the write data line, and the register to store it at (`$t0`) on the write register line.
- Starting at the Program Counter (PC) we trace the information we have to each unit, highlighting the pieces of the unit which are used by the instruction to obtain/calculate the information needed to complete the instruction operation.



Store

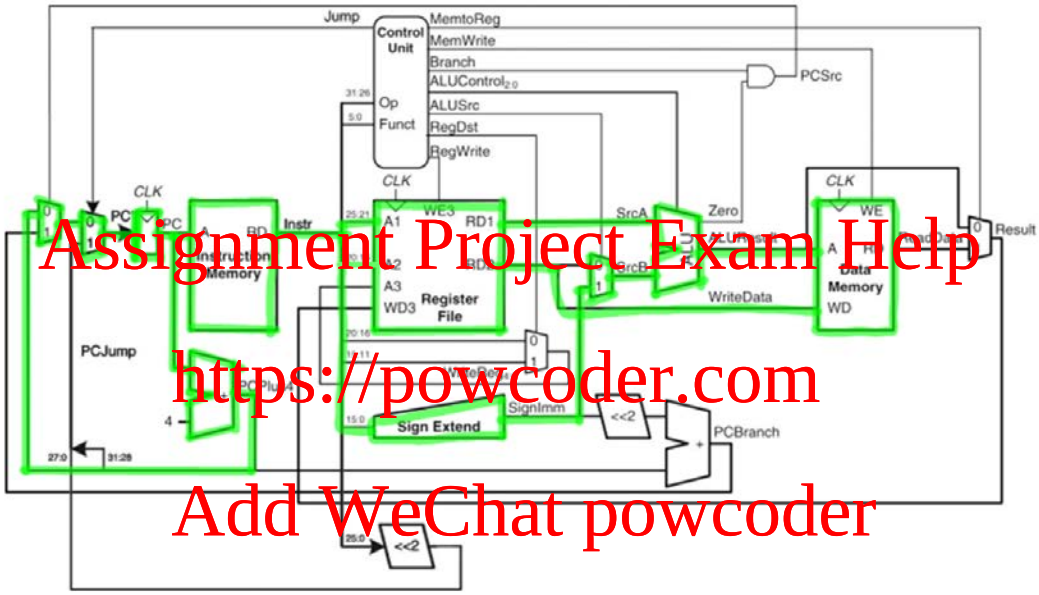
Ex: `sw $t1, 16($t2)  # Mem[Reg[$t2] + 16] = Reg[$t1]`

[Store Instruction Walk-through]

I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

- The rs register (`Reg[$t2]`) and the rt register (`Reg[$t1]`, the data to store) are read from the register file. `Reg[$t2]` is available on Read data 1 (RD1 - rs). `Ref[$t1]` on the Read data 2 (RD2 - rt).
- To the `Reg[$t2]` (RD1 - rs) value the immediate value (after sign extension, `SignImm`) is added using the `ALU Reg[$t2]+16`
- This is the address where the data is to be stored, the value to be stored (`Reg[$t1]`) is on the write data (WD) line of the data memory.



ALU instructions (R-type instructions)

Ex: `add $t0, $s0 $s1  # Reg[$t0] = Reg[$s0] + Reg[$s1]`

[R-type Instruction Walk-through]

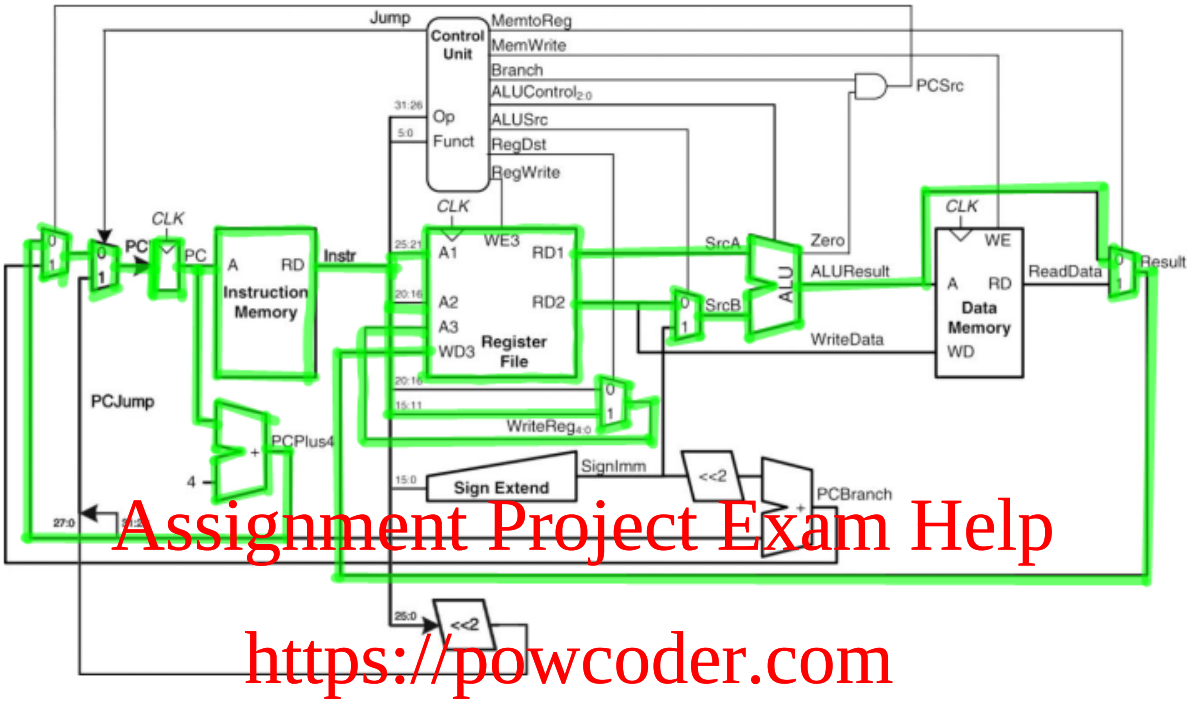
R-type

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

- These instructions take two operands from register (rs & rt) and perform the ALU operation on them. Store the resultant value back in a register (rd).
- Their operation can be split into the following steps:

R-type
Fetch – Fetch Instruction
Decode - Register Read (rs: `$s0`, rt: `$s1`)
Execute - Perform ALU operation (Reg[$s0] operation Reg[$s1])
Execute - Write data to register (rd: `$t0`)

- Remember, the fetch phase is identical for all instructions.
- Many of the same components used by load and store are used. A few differences are:
  - The sign extension unit is not used (output is generated, but not chosen for use) because there is no immediate value. The ALU performs operation on the RD1 (rs) and RD2 (rt) values.
  - The Data memory is not used. The result value of the ALU is then written back to the Register file.
  - Different values are passed to the ALU and Register File by selecting the proper MUX control signals.



(We will cover branch & jump instructions in the second part of the module)

