ICS51 - MIPS Multicycle Datapath

- The single cycle implementation of the MIPS datapath was inefficient.
    - Clock cycle time is dictated by the longest instruction. The rest of the instructions are wasting time doing nothing but waiting for the clock.
    - Throughput (number of instructions per unit time) as a result is low

Single Cycle Performance

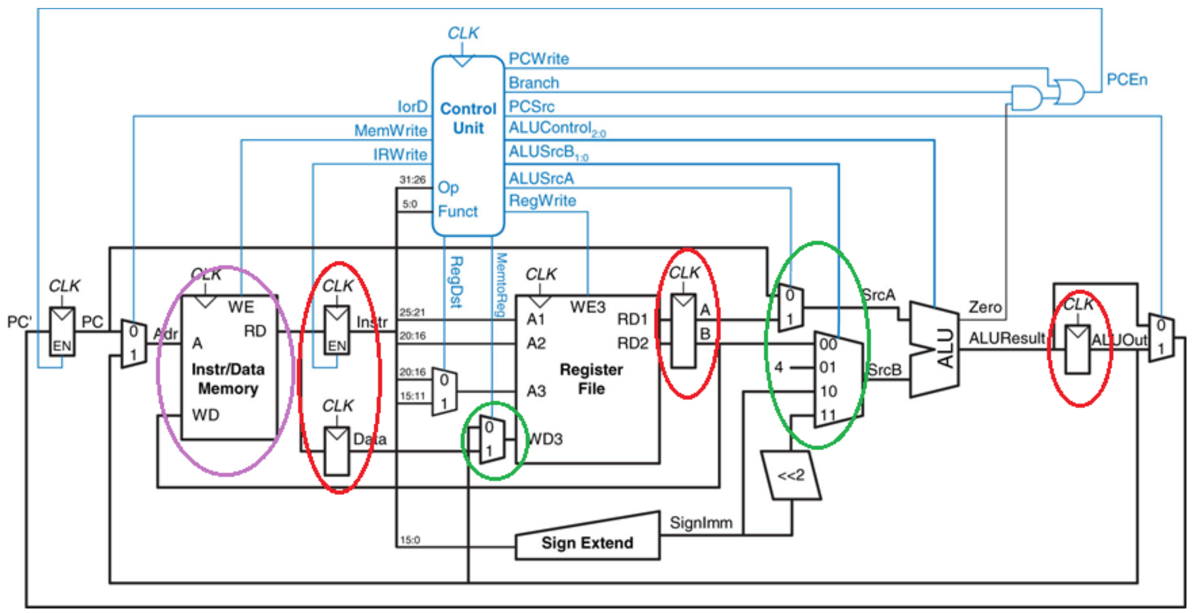
- One approach to address this issue is to split the datapath into smaller pieces/stages each which can be performed efficiently in a single clock cycle.
    - Each type of instruction must then execute only the stages of the datapath that are required for the instruction.
    - The clock cycle time is now dictated by the longest stage of the datapath rather than the full instruction.
    - However, now extra control is needed to determine how many and which stages are required for the current instruction.
        - In addition to using the opcode to determine the control signals, now we need to know which stage of the datapath we are executing, and which is next.
    - An additional advantage is that we can reuse components of the data path. Within each piece of the datapath a component can only be used once, but a unit can be used for different purposes during different pieces.
        - The ALU can be used to compute PC+4, the memory address, and the branch address
        - A single instruction/data memory can be used rather than separate units

The Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
    - balance the amount of work to be done in each cycle, meaning try to keep the timing of each piece similar and the tasks per cycle logical
    - restrict each cycle to use only one major functional unit (this functional unit is the bottleneck; 2 units would make the cycle time longer)
- At the end of a cycle, we must
    - store the intermediate values for use in the later cycles
    - We can do so by introducing additional "internal" registers

Multicycle Datapath Overview

- Below is the multicycle datapath:



- o Red circles indicate the added registers.
- o The Instruction memory and data memory are now combined in to a single unit (purple).
- o The adders are removed. ALU can perform all necessary operations.
- o Additional multiplexors are added to select between more options (green). The Jump mux is removed.

- The above datapath changes were decided by considering the MIPS instruction set/ISA
- Ex: add $1, $2, $3
  - o The Instruction was specified by the PC address in memory
  - o The instruction changes a register, which is specified by bits 15:11 of instruction.
  - o The value to store in the register is the sum ("op") of two registers, specified by bits 25:21 and 20:16 of the instruction

  - o Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]] op Reg[Memory[PC][20:16]]

  - o We can break up the above statement into smaller manageable pieces like we would in HLL programming (introducing variables)
  - o Could break down to:
    - ▪ IR <= Memory[PC]
    - ▪ A <= Reg[IR[25:21]]
    - ▪ B <= Reg[IR[20:16]]
    - ▪ ALUOut <= A op B
    - ▪ Reg[IR[15:11]] <= ALUOut

  - o We forgot an important part of the instruction!
    - ▪ PC <= PC + 4
  - o Each of the intermediate variables are where we have to store temporary variables/values. These are the new registers in the datapath.

Five Execution Stages of the Multicycle Datapath

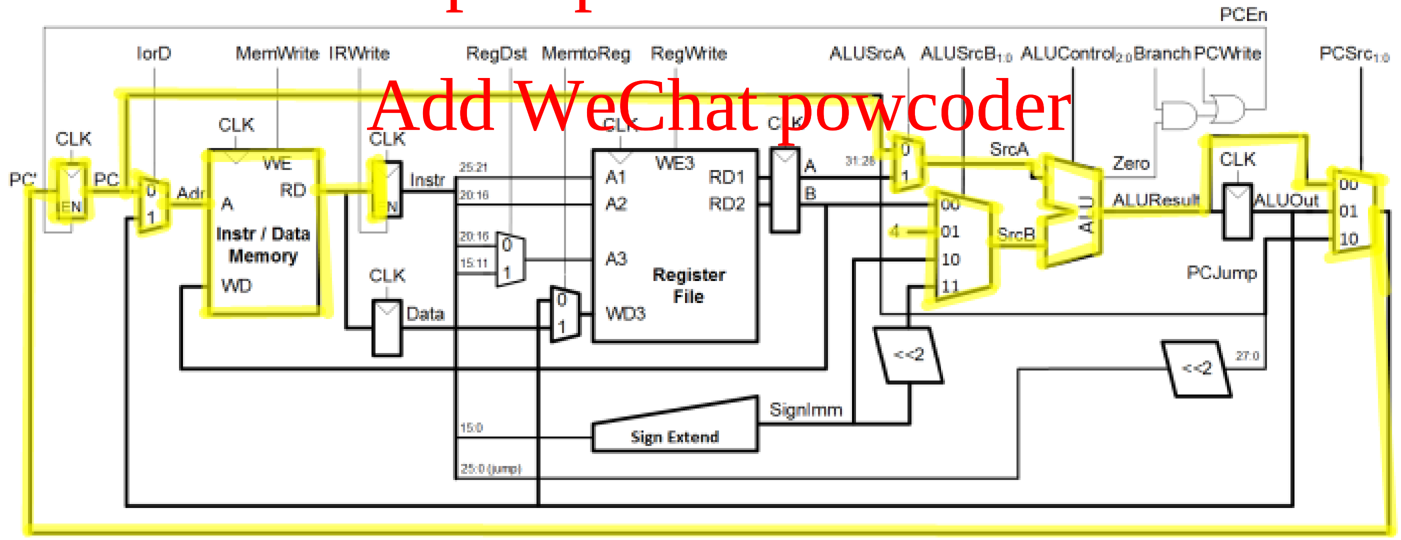
Multicycle Datapath Stages

- The multicycle datapath is split into 5 stages (one for each major task in the instruction, only one main functional unit per stage)
  - Instruction Fetch and PC <= PC+4
  - Instruction Decode and Register Fetch
  - Execution, Memory Address Computation, or Branch Completion
  - Memory Access or R-type instruction completion
  - Write-back step
- Each instruction will only use a subset of these stages. Therefore, instructions will take from 3-5 clock cycles to execute. As a result, each instruction will take different amounts of time to execute.

- **Stage 0:** Instruction Fetch

Instruction Fetch Walk-through

  - There are two main tasks
    - Use PC to retrieve the instruction from the Memory and store the 32-bit instruction value in the Instruction Register. IR <= Memory[PC]
    - Increment the PC by 4 and store the result back in the PC. PC <= PC + 4;
  - This stage is instruction independent. No knowledge of the type of instruction is available.



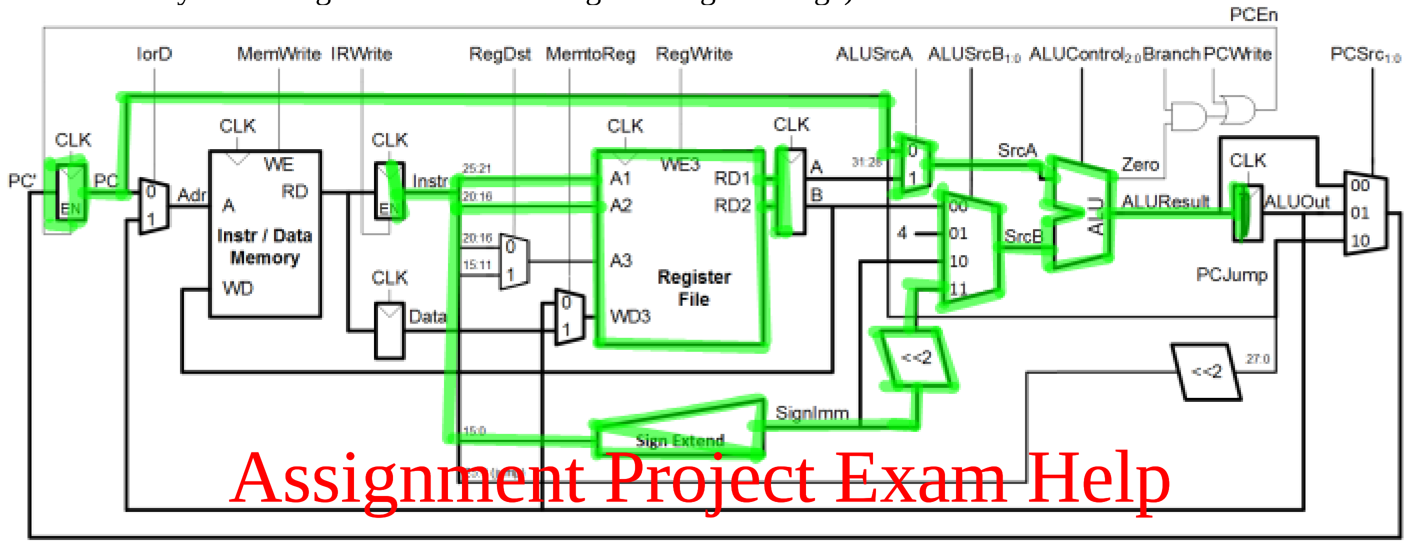
- **Stage 1:** Instruction Decode and Register Fetch (Reg Read)

Instruction Decode Walk-through

  - This stage has 3 components:
    - Use the opcode to determine which instruction this is (control unit)

  - Read registers rs and rt from the register file (in case we need them)
    A <= Reg[IR[25:21]];
    B <= Reg[IR[20:16]];
  - Compute the branch address (in case the instruction is a branch)
    ALUOut <= PC + (sign-extend(IR[15:0]) << 2);

  The ALU was not being used for anything in this step, therefore we anticipate the need for the branch address.
- This stage is instruction independent. The control lines are not set on the instruction type yet (we are busy "decoding" it in our control logic during this stage)



Note: The lecture notes present the remaining stages based on stage. The videos are recorded based on instruction. This provides 2 different viewpoints of the same material.

LW Walk-through

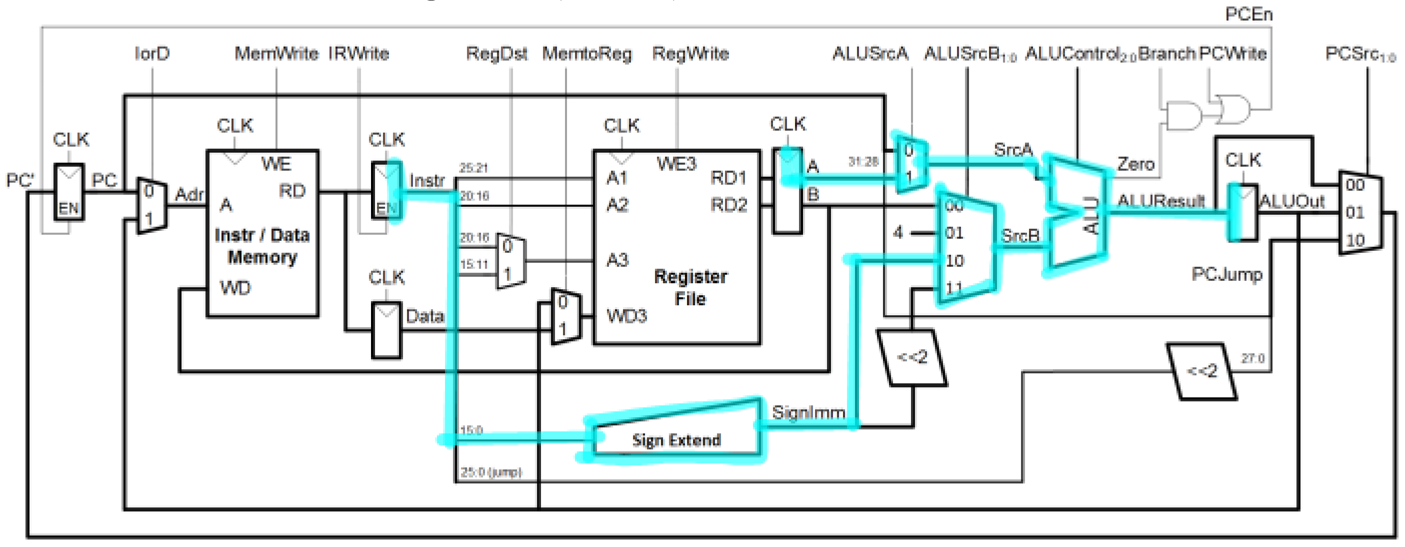
SW Walk-through

R-type Walk-through

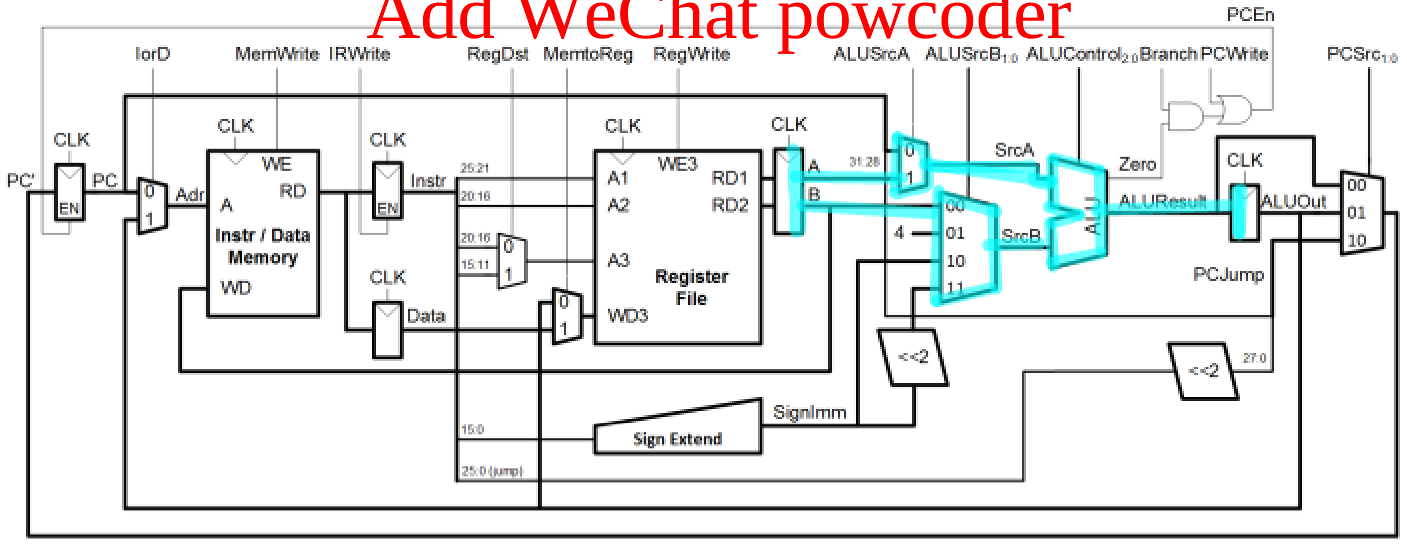
BEQ Walk-through

Jump Walk-through

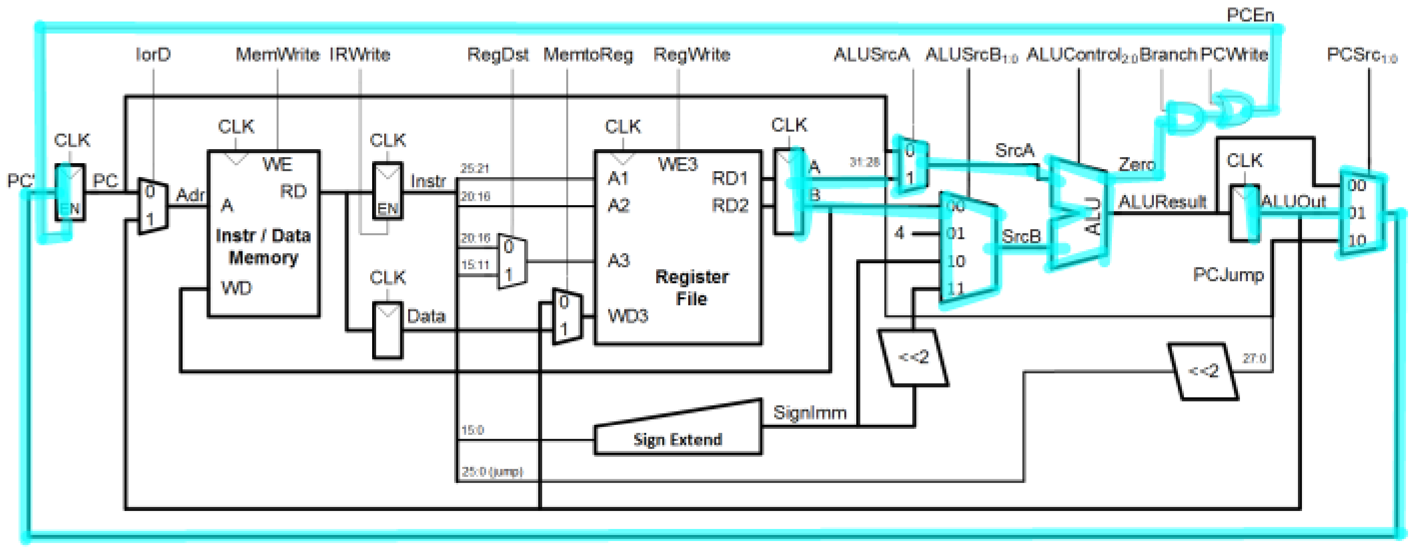
- **<u>Stage 2: ALU</u>**
    - This stage is instruction dependent. ALU is performing one of 4 functions, based on instruction type (memory, R-type, Branch, jump). Note Jump is not illustrated below.
    - **Bubble 2:** Memory Reference/immediate instructions (lw,sw,addi, etc):
      ALUOut <= A + sign-extend(IR[15:0]);



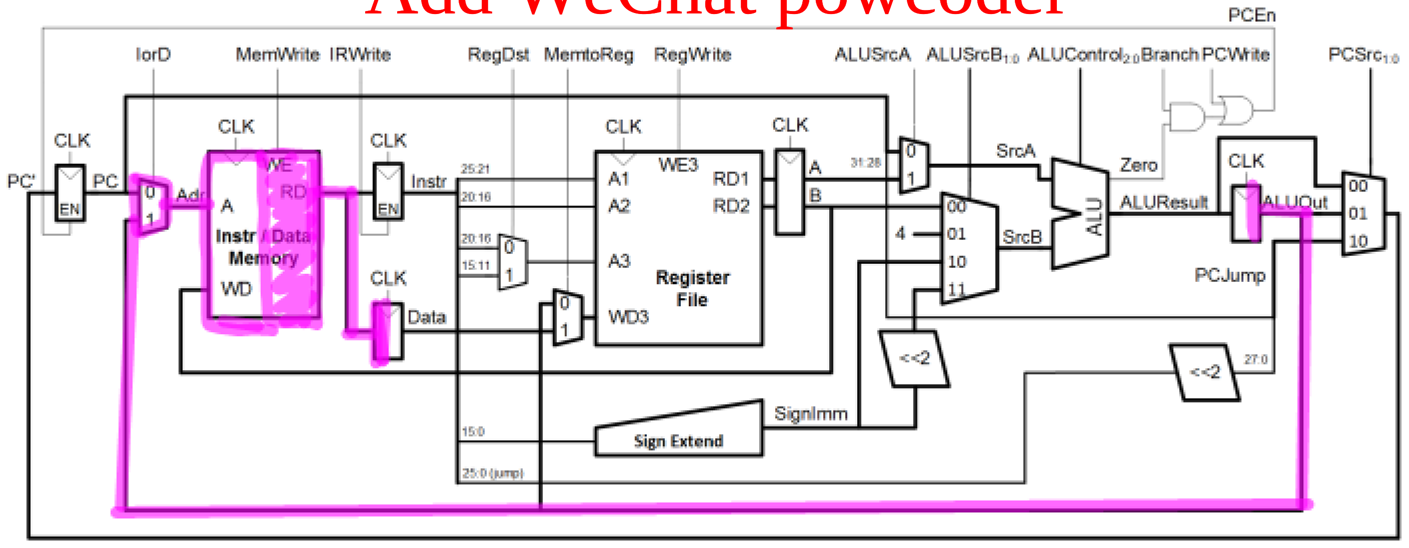
- **<u>Stage 2: R-type</u>** (add,sub,slt, etc):
    - **Bubble 6:** ALU operations
      ALUOut <= A op B;



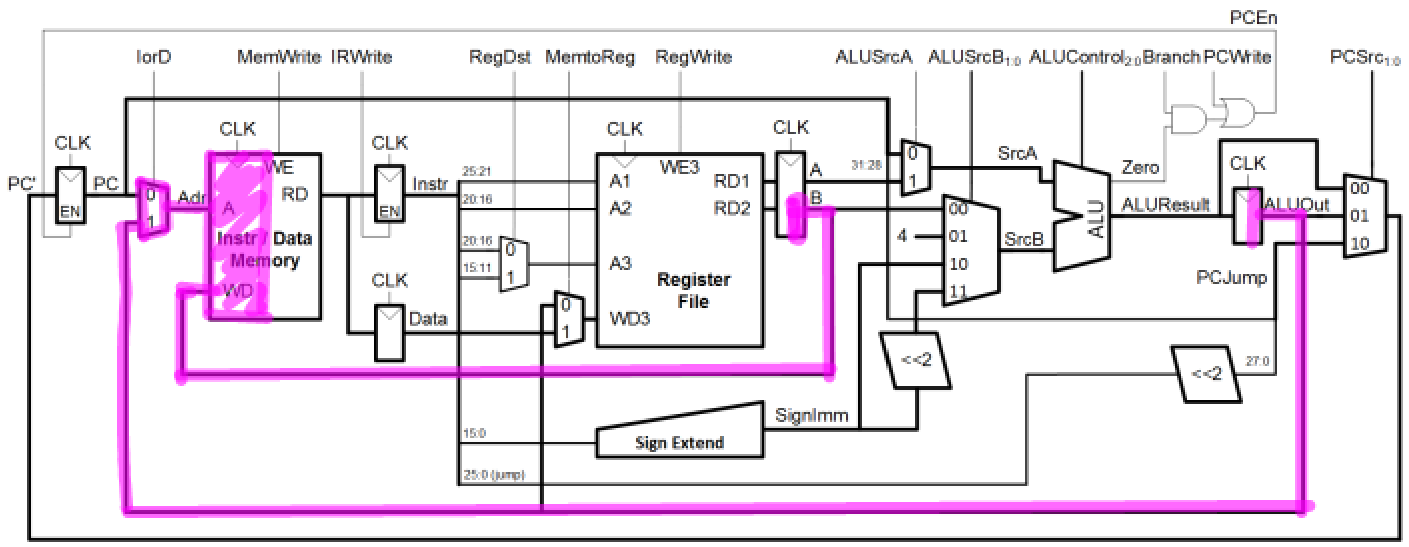
- **Stage 2: Branch** (beq):
  - **Bubble 8:** comparison of values for branch
    if (A==B) PC <= ALUOut;



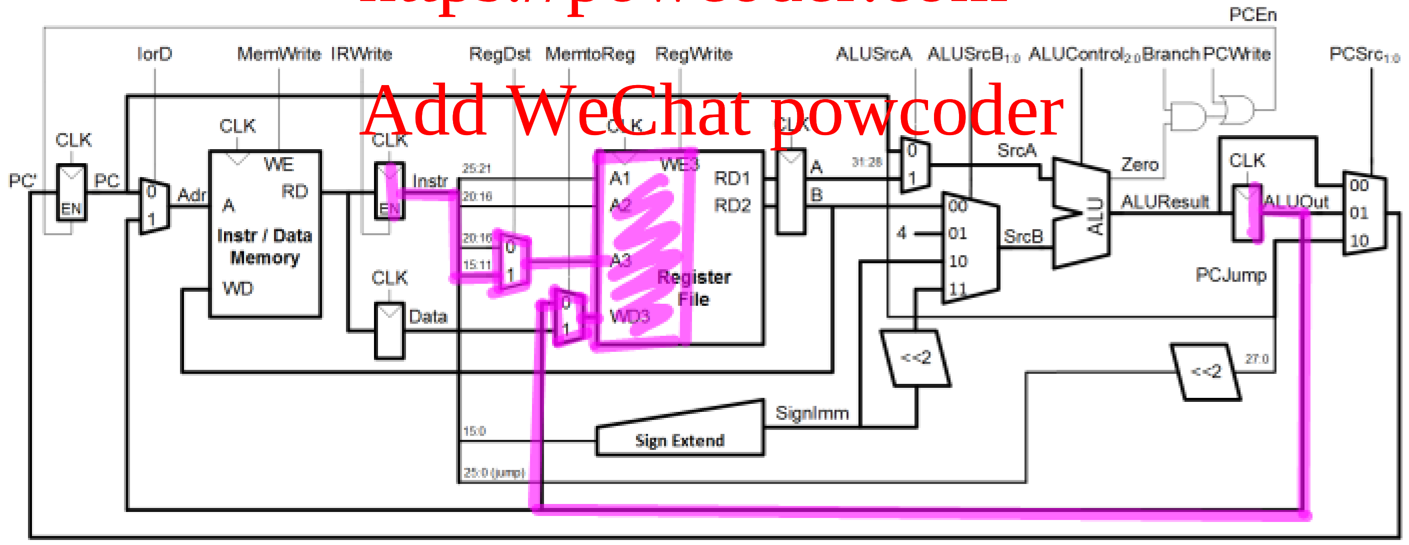
- **Stage 3:** Memory (lw)
  - This stage is only required for R-type or memory-access (lw,sw)
  - **Bubble 3:** Load Word
    MDR <= Memory[ALUOut];



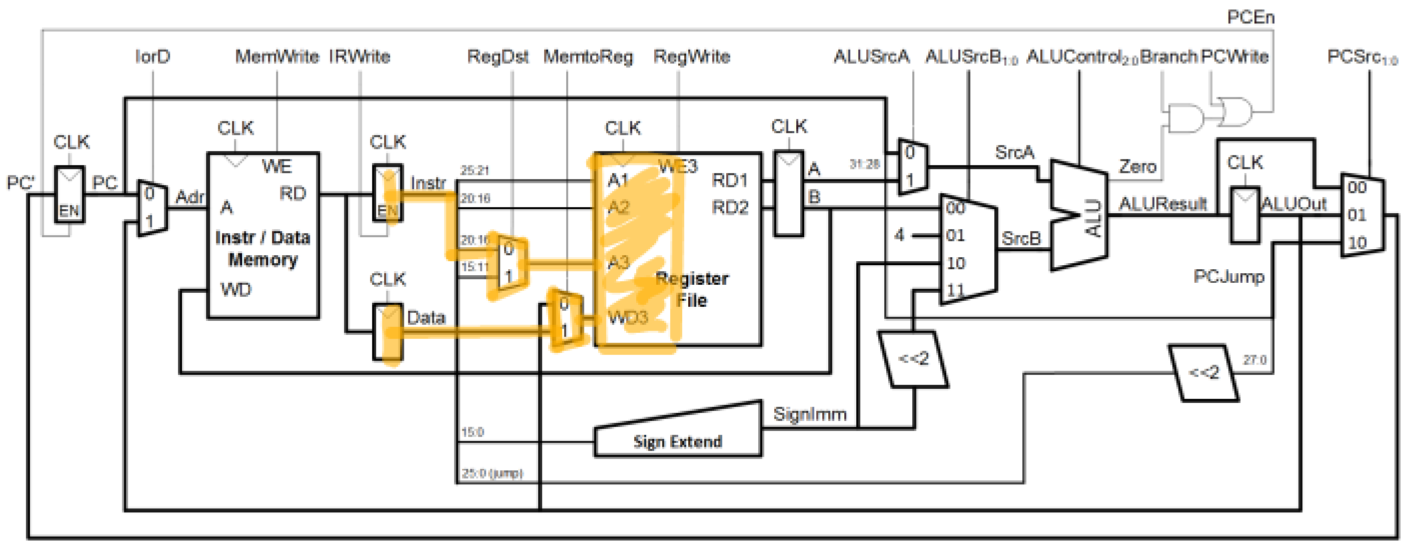
- **Stage 3:** Memory (sw)
  - **Bubble 5:** Store Word
    Memory[ALUOut] <= B;



- **Stage 3:** Reg File (r-type)
  - **Bubble 7:** R-type instructions store to the register rd
    Reg[IR[15:11]] <= ALUOut;



- **Stage 5:** Write Back (WB)
  - This is instruction dependent and only required for load word instructions.
  - **Bubble 4:** Write the data from memory into the registers
    Reg[IR[20:16]] <= MDR;



Multi-cycle Implementation Summary

- The chart below summarizes all the different operations which occur in each stage/step.
- Remember each stage/step is a single clock cycle. The clock cycle time is dependent on the critical path for any of the different steps.

- Each instruction type (columns) require a different number of clock cycles to complete depending on the number of stages/steps required.
  - R-type:4 clock cycles
  - Lw: 5 clock cycles
  - Sw: 4 clock cycles
  - Beq: 3 clock cycles
  - J: 3 clock cycles

- The length of the clock cycle is determined by the length of the longest critical path. The critical path must be calculated for each of these stages/steps separately.
  - Remember: for any clock cycle, the processor does not know which stage/step or the instruction it is executing. So the clock has to operate for the worst case timing.

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Note: IR is shown in the figure as “Instr”. MDR is shown in the figure as “Data”

Thinking about Timing and Performance - Questions

Multi-cycle Datapath Timing

Consider the following MIPs code:
```
     lw $t2, 0($t3)
     lw $t3, 4($t3)
     beq $t2, $t3, Label   #assume not
     add $t5, $t2, $t3
     sw $t5, 8($t3)
Label:    ...
```

How many cycles will it take to execute this code?

21 clock cycles = 5+5+3+4+4 = lw + lw+ beq+R-type+sw

What is going on during the 8th cycle of execution?

lw + partial lw = 5 + 3 for second lw. The third stage of lw is ALU.

ALUOut = Reg[$t3]+sign extended immediate value 4

In what cycle does the actual addition of $t2 and $t3 takes place?

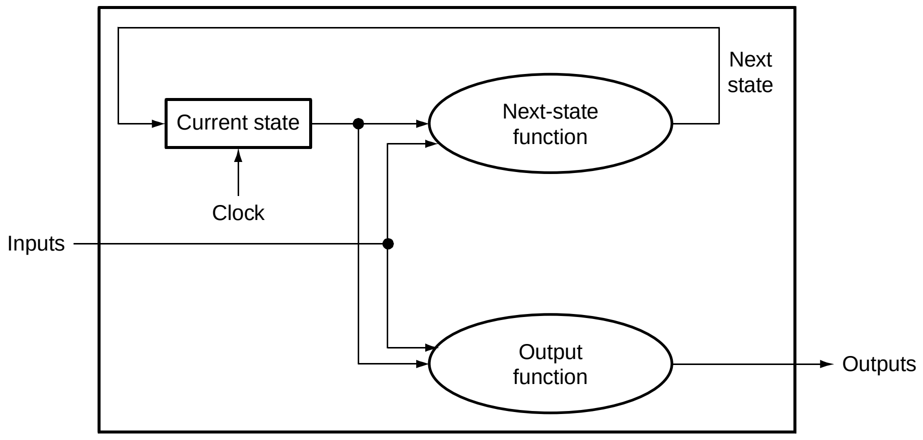
Cycle 16. lw + lw + beq + add until ALU stage = 5+5+3+3

Multicycle Datapath Control

- The multicycle datapath created smaller stages to perform the tasks for each instruction.
- For each of these stages, the control signals for the entire datapath must be determined.
- Because each stage will now operate as an independent clock cycle, we must document/remember which state we are currently in and determine which state to go to next for the particular type of instruction.
- This type of control is performed using a Finite State Machine.

Finite State Machines

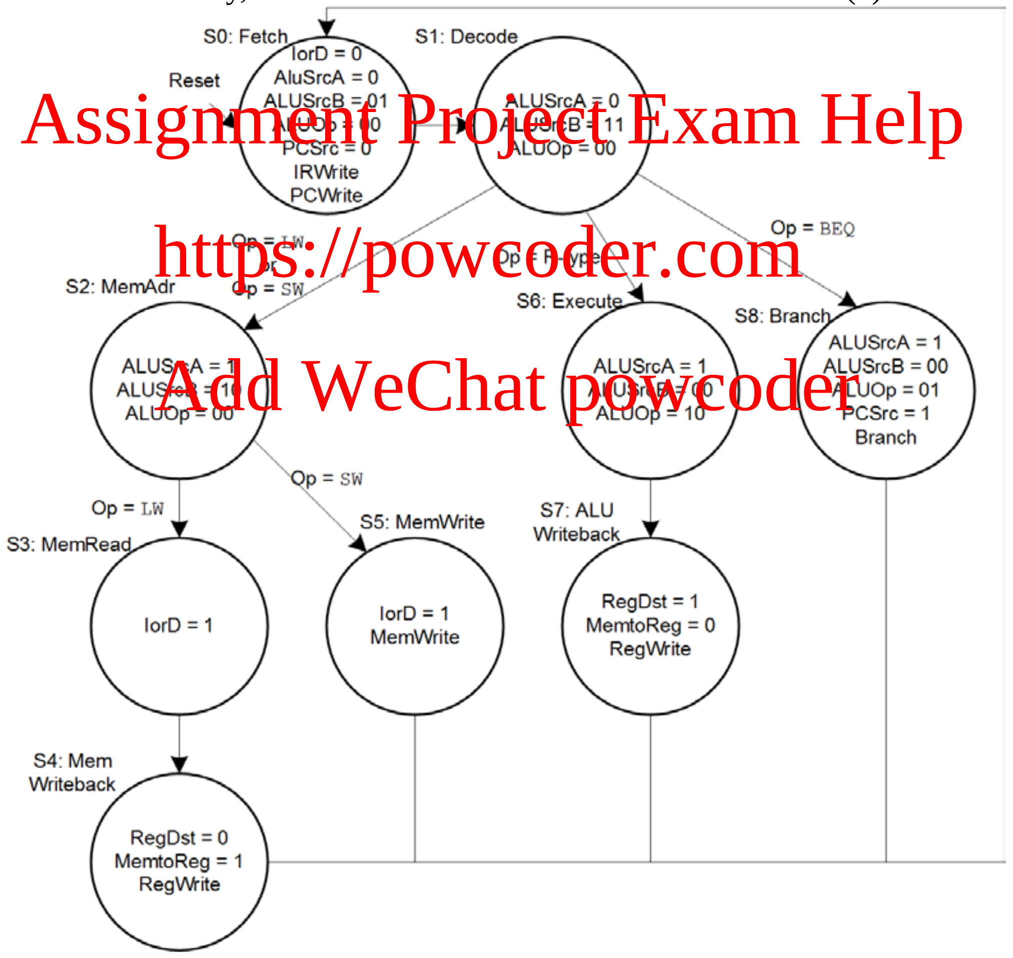
- Finite state machines, have a finite set of defined states. The name of each state (or binary value) is used to determine both the next state and the output values.
  - The next state is determined by the current state and any input values
  - The output values are determined by the current state and possibly the input values



Multicycle Datapath Control

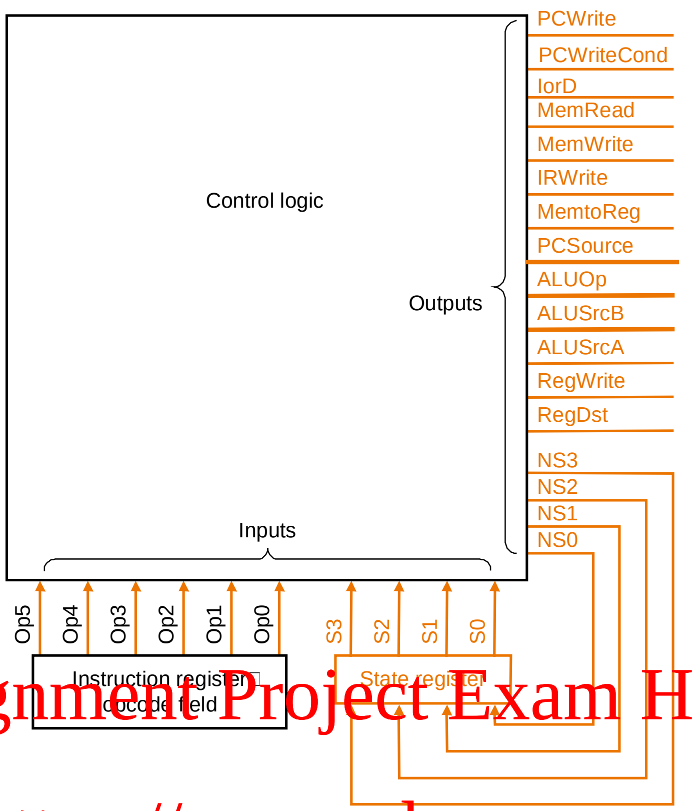
[Multi-cycle Datapath Control – Finite State Machine]

- We use a finite state machine to denote the state transitions and the control values for each type of operation in the multicycle datapath
- To implement the control in this case, the value of the datapath control signals (all outputs from the control Logic oval) are dependent upon:
  - what instruction is being executed
  - which step is being performed
- In the following figure we denote each of the unique states required for all basic instructions.
- Note:
  - If a control signal is not mentioned in the bubble, then it is a don't care (except for write signals)
  - Any control that is deals with write (PCWrite, RegWrite, IRWrite, etc) is deasserted (0) if not specified
  - If the name is listed only, this means the value of the control is asserted (1)



- The current finite state machine has 10 states, therefore 4 pits are required to specify the state number (shown to the top left of each bubble)
- Transitions between states are determined by the opcode when specified.
- Note that state 0 and 1 are instruction independent and will always be executed for any instruction (basic and new).

- The control values in each bubble/state are specified by the datapath for each stage for each type of instruction as presented in last lecture.



- To implement the Control Logic oval, the above system can be created
  - o The state register holds the current state number (S0-S3)
  - o The Opcode is directly from the instruction register in the datapath
  - o Each of the outputs, the next state (NS0-NS3) and the control signals for all components in the datapth are determined by digital logic as performed at the start of the semester. (truth table mapping state and opcode to outputs.

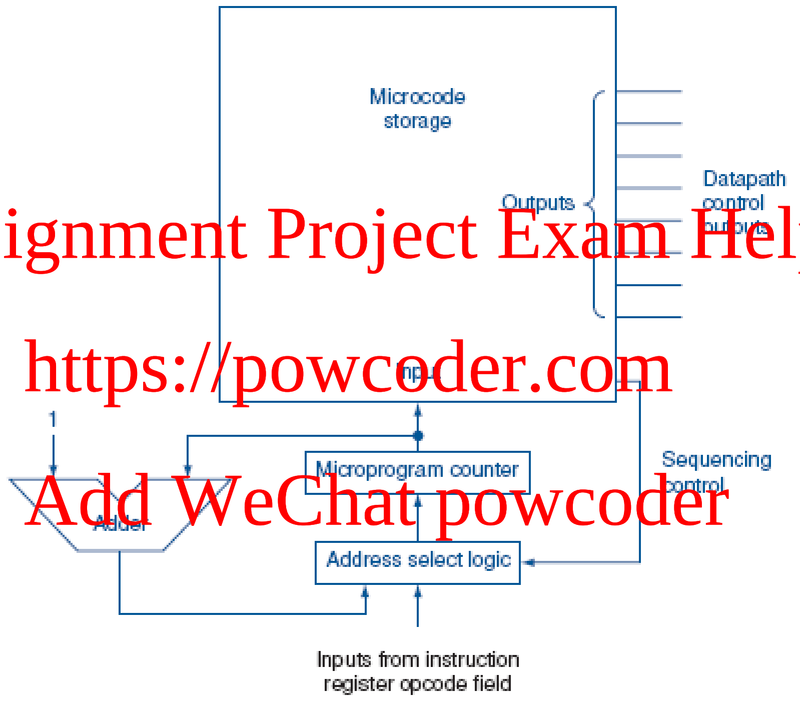
Multi-cycle Datapath Control Unit & Microprogramming

Microprogramming

- An alternate approach to finite state machines is to use microprogramming
- A static program for each instruction is created. The processor executes one microinstruction per clock cycle.
- Similar to the finite state machine the microinstruction itself consists of several control signals for the datapath. If a particular microinstruction bit is 1, then the particular corresponding output is 1.
- The size of microinstruction is dictated by the number of control bits we need.
- For each MIPS instruction (eg. lw, sw, slt, beq), has its own microprogram.
  - o Execution of an instruction means executing its microprogram.
- The microprograms for each instruction is stored in a ‘control store’ which is a ROM (read only memory)
- Why?
  - o A microprogram never changes. The instructions and the sequence of these microinstructions is set in stone when the datapath is designed.
  - o A given instruction is always executed the same way, therefore the control store is written when the

processor chip is made. There is no need to rewrite it.

- How do we get the first microinstruction?
  - The opcode usually provides the address of the first microinstruction in the corresponding microprogram. This is done in the decode stages. The Fetch and decode microinstructions are the same in all microprograms.
- The microcode storage works like a look-up table.
- A microinstruction also carries some explicit information about the next microinstruction (address). This is the sequencing control part. Sometimes we add 1 to obtain the next address, but not always (think end of instruction).
- We also have a microprogram counter (MPC).
  - It stores the address of the next microinstruction.
  - Many (MIPS) instructions may share some microcode. Similar to how states in the finite state machine are shared.
  - Using explicit sequencing (+1 increment) control keeps the size of the ROM small. Less repeated code. Less computation to calculate next microinstruction.


Microcode storage
Outputs
Datapath control outputs
Input
1
Microprogram counter
Sequencing control
Adder
Address select logic
Inputs from instruction register opcode field