Unit 12: Single Cycle Processors

CSE 220: System Fundamentals I

Stony Brook University

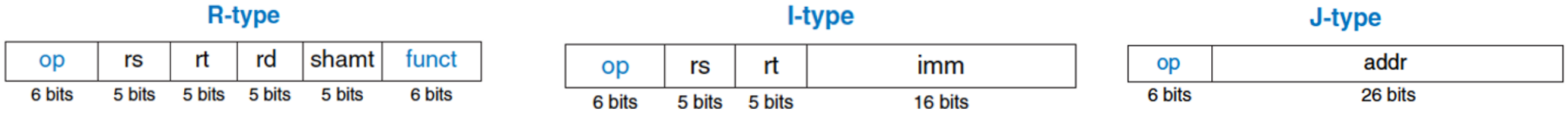
Joydeep Mitra

Microarchitecture

- Microarchitecture is the connection between logic and architecture – a specific arrangement of registers, ALUs, memories, and other logic blocks.
- A particular architecture (e.g., MIPS) may have numerous microarchitectures, each with different trade-offs of performance, cost, and complexity.
- We will study two microarchitectures for the MIPS processor architecture
  - Single Cycle – executes an entire instruction in one cycle
  - Pipelining – executes several instructions simultaneously with the goal of improving throughput

Architectural State And Instruction Set

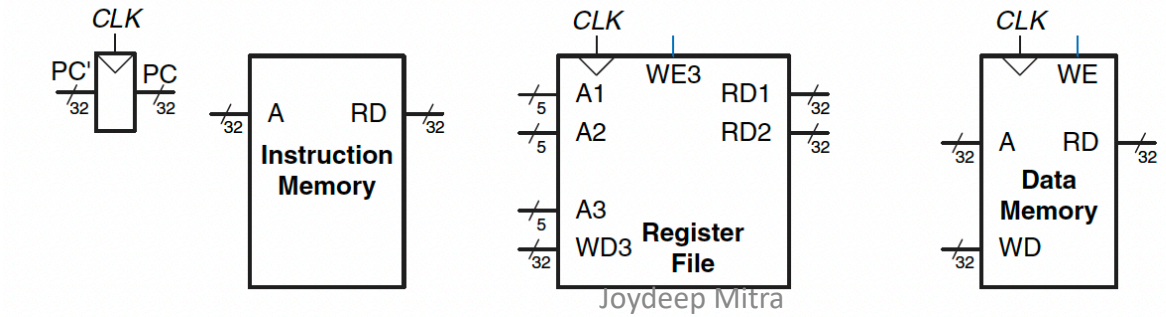
- Any computer architecture is defined by its architectural state and instruction set
- Recall the architectural state of MIPS is the program counter (keeps track next instruction to execute) and the 32 registers
- Also, recall the MIPS instruction formats – R-Type, I-Type, and J-Type



- A microarchitecture must remember all of this state
- Based on the current architectural state, a processor executes a particular instruction to produce a new state

Design Process

- We will divide a microarchitecture into **datapath** and **control.**
  - The datapath operates on words of data and contains structures such as memories, registers, ALUs, and multiplexers
  - The control unit receives the current instruction and tells the datapath how to execute it.
- It is convenient to first define the hardware that will contain the architectural state.
  - **Program State** *is 32-bit register. PC points to current instruction and PC' indicates next instruction address.*
  - **Instruction Memory** *has a single read port. It takes a 32-bit instruction address input, A, and reads the 32-bit instruction from that address onto the read data output, RD.*
  - **Register File** *has ports, A1, A2, and A3, which take 5-bit addresses. Data from A1 and A2 are read onto RD1 and RD2. Data from WD3 is written to A3. WE3 is a write enable signal, which if enabled (set to 1) allows writing to register on the rising edge of the clock.*
  - **Data Memory** *has a single read/write port. If WE is enabled, i.e., set to 1, then it writes data WD into A; otherwise, reads from address A onto RD*

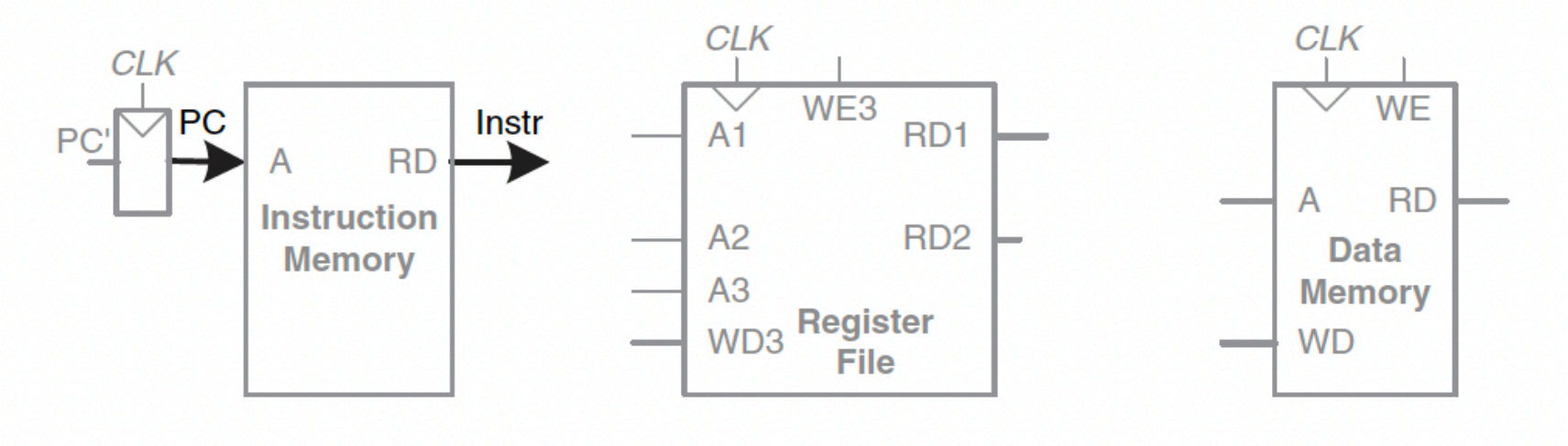


Single Cycle Data Processor

- We will learn how to design a single cycle processor by first looking at the `lw` instruction
- Then we will generalize the design to include other instructions
- To keep things simple, we will focus on:
  - R-type arithmetic/logic instructions: `add, sub, and, or, slt`
  - Memory instructions: `lw, sw`
  - Branches: `beq`

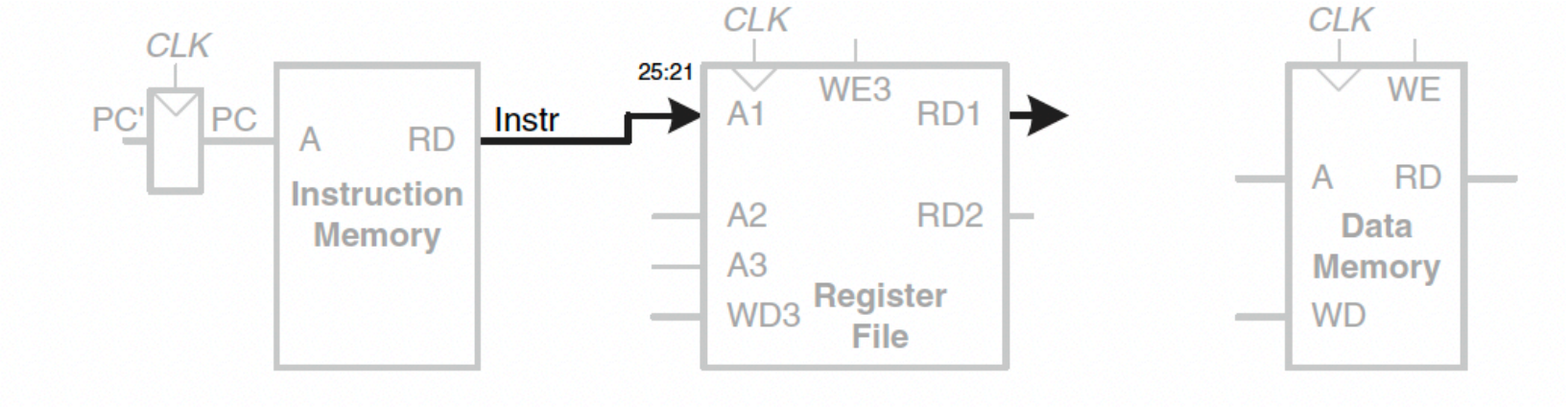
Single Cycle Data Processor: `lw`

- The first step is to fetch an instruction from instruction memory



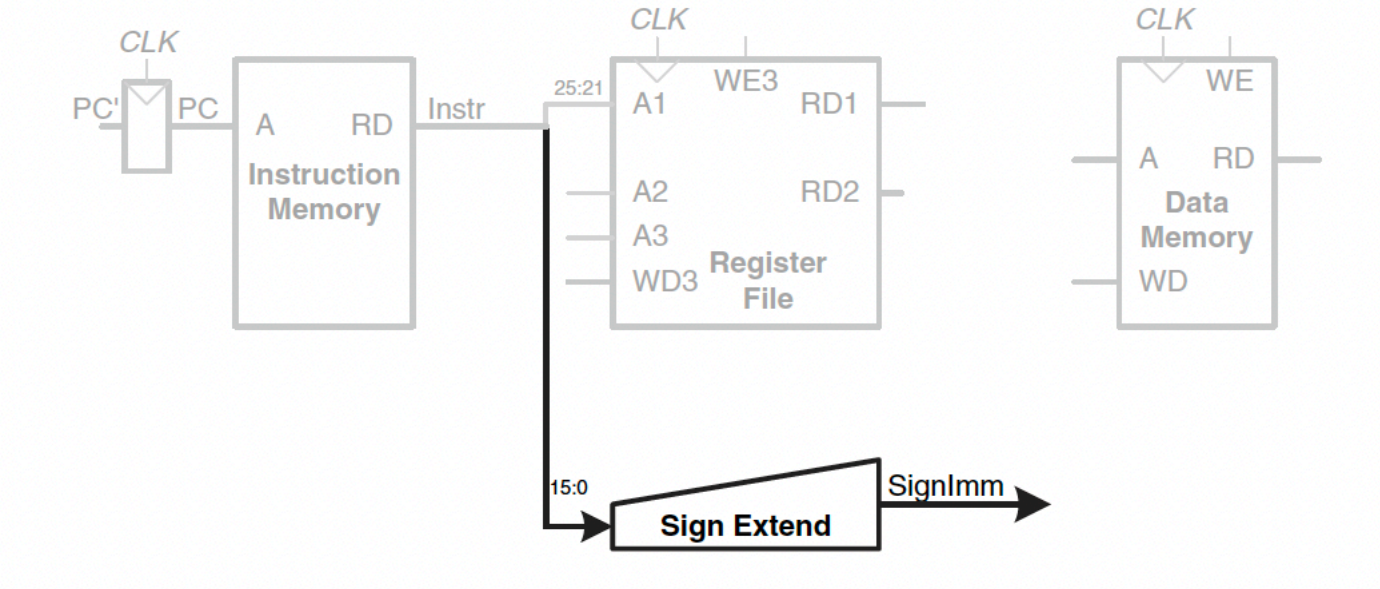
Single Cycle Data Processor: `lw`

- The next step is to read the source register containing the base address
- The register is specified in the `rs` field of the instruction, $\texttt{Instr}_{25:21}$
- Bits $\texttt{Instr}_{25:21}$ are connected to A1 of register file
- The register file reads the register value onto RD1



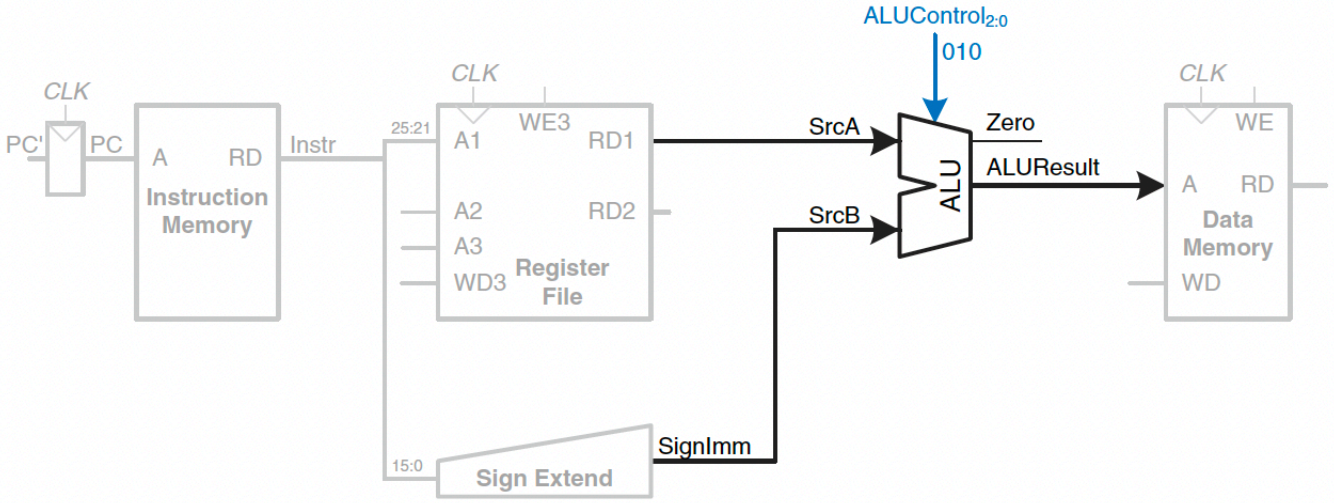
Single Cycle Data Processor: `lw`

- The `lw` instruction also needs an offset, which is added to the base address
- The offset is stored in the immediate field of the instruction, $\texttt{Instr}_{15:0}$
- The offset must be sign-extended since it can be positive or negative



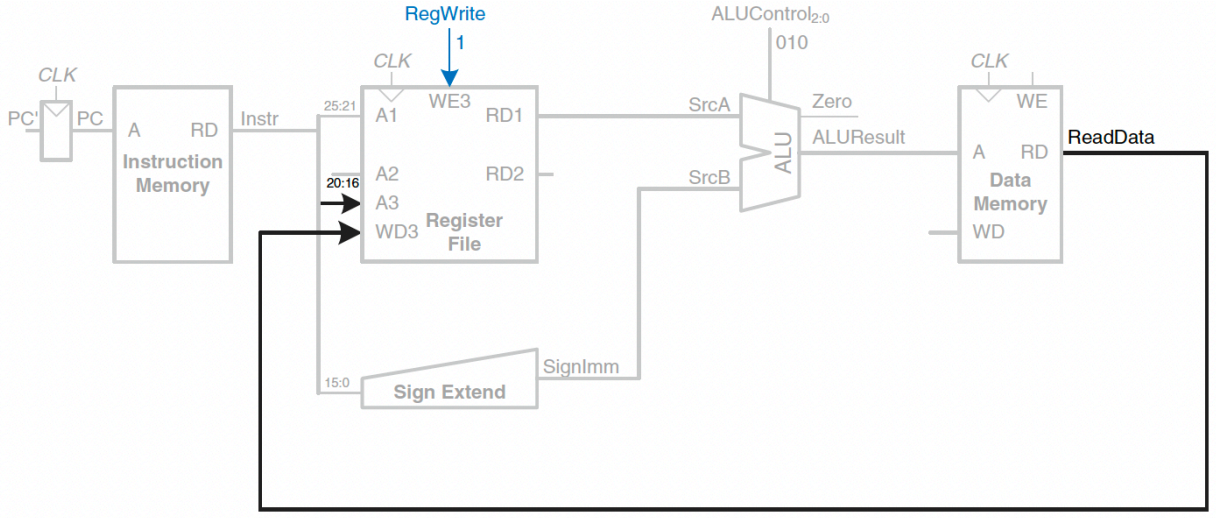
Single Cycle Data Processor: `lw`

- Next, we introduce an ALU to add the base address (*SrcA*) and the offset (*SrcB*)
  - *SrcA* comes from register file, and *SrcB* comes from sign-extended immediate
- The ALU can perform various operations based on the 3-bit *ALUControl* (assume it is 010 for *add* instruction for now)
- The ALU produces a 32-bit *ALUResult* and a *Zero* flag, that indicates if *ALUResult == 0*
- *ALUResult* is sent to the data memory as the address of the load instruction



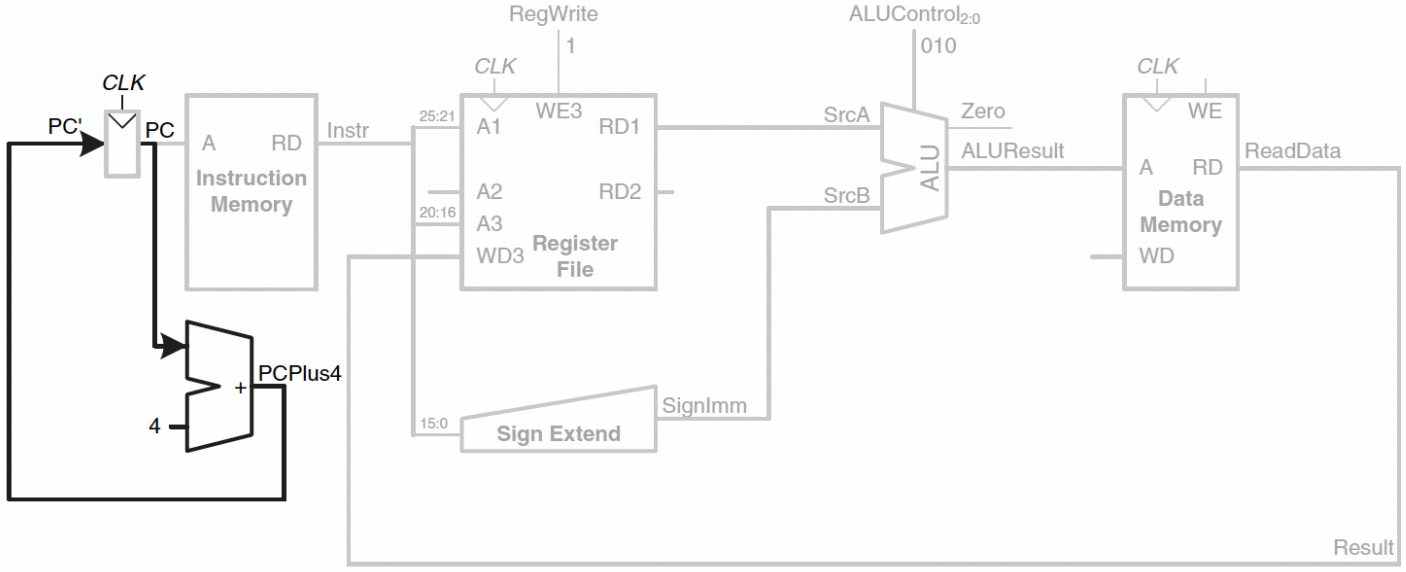
Single Cycle Data Processor: `lw`

- Data from port `A` in *data memory* is read onto bus `RD` and connected back to destination register in the *register file* at the end of the cycle
- The destination register is the `rt` field in the **lw** instruction. Hence, port `A3` is connected to bits $\texttt{Instr}_{20:16}$
- The signal `WE3` (connected to `RegWrite`) must be set to 1 so that data in `WD3` can be written to destination register
  - Write takes place on the rising edge of the clock cycle



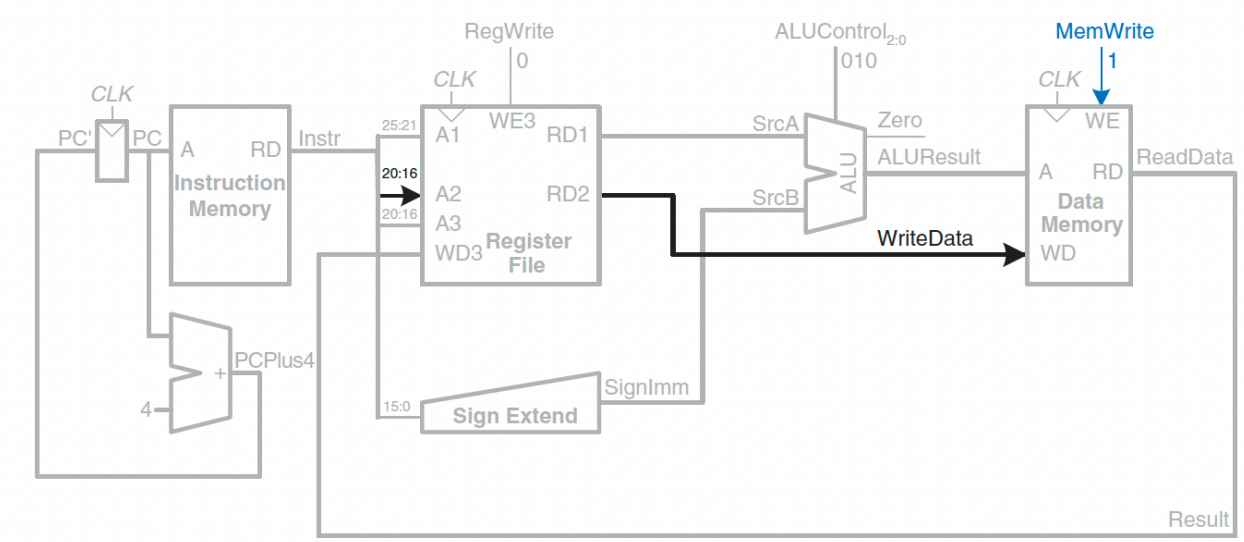
Single Cycle Data Processor: `lw`

- While `lw` is executing, the processor must determine the next instruction. How?
- Increment the program counter by 4 (since instructions are 32-bits = 4 bytes)
- We introduce an additional adder
- `PCPlus4` is written to program counter on the rising edge of the clock cycle



Single Cycle Data Processor: `sw`

- We will extend the datapath with `sw` instruction.
- The `sw` instruction reads from a register and stores in address + offset.
- Address calculation ALU already exists.
- From instruction format, we know the register field is `rt` (bits $\texttt{Instr}_{20:16}$).
- These bits are connected to A2 in the register file and are read out of the RD2 port.
- *WriteData* from RD2 is then connected to WD port of Data Memory, which is then written to address A on the rising edge of the clock when *MemWrite* is set to 1.

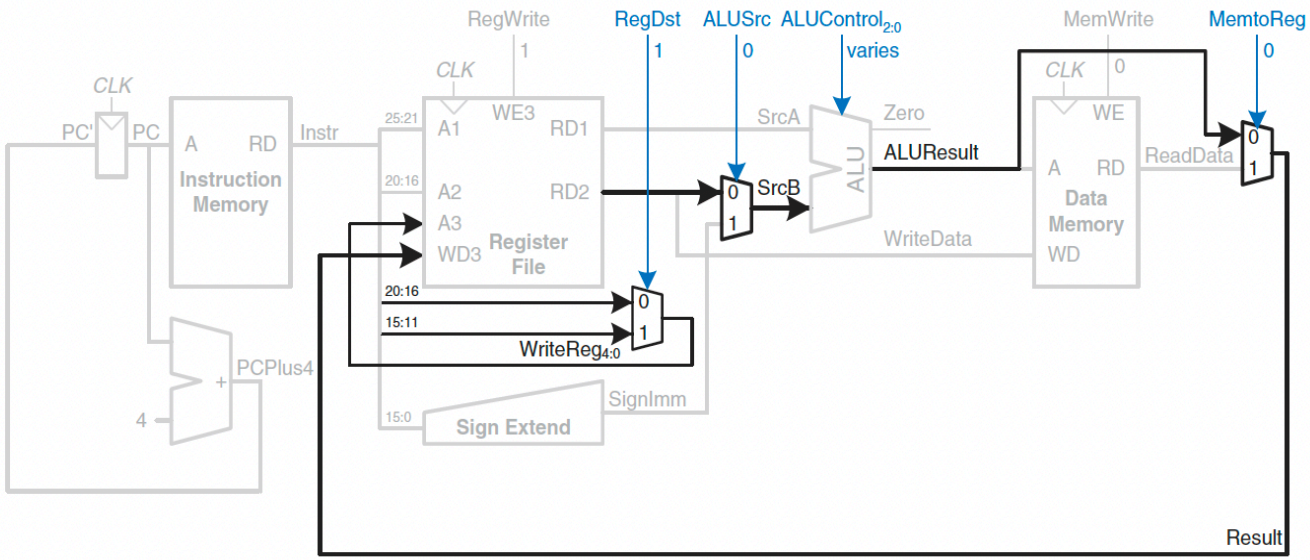


Single Cycle Data Processor: `RType`

- We will extend the datapath with R-Type instructions – `add`, `sub`, and, `or`, and `slt`
- These instructions have a common pattern
  - Read two registers from register file
  - Perform some ALU operation on them
  - Write result back to a 3rd register file
- They can be handled with the same hardware, except for the ALU operation that can be handled with different *ALUControl* signals

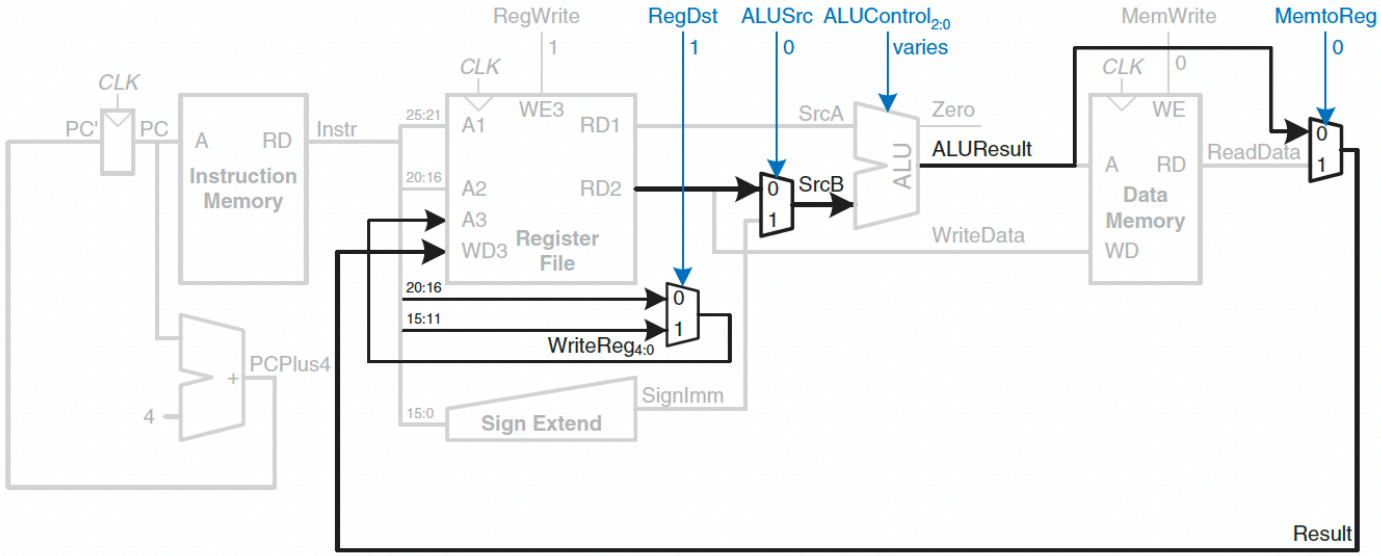
Single Cycle Data Processor: `RType`

- The ALU operation needs two operands, *SrcA* and *SrcB*, from the register file
- But previously the 2nd operand to the ALU was always the sign-extended immediate
- Clearly, we need a multiplexer to choose between the *RD2* and *SignImm*
- The control/select signal for the multiplexer is *ALUSrc*, which is 0 for R-Type instructions and 1 for `lw` and `sw`



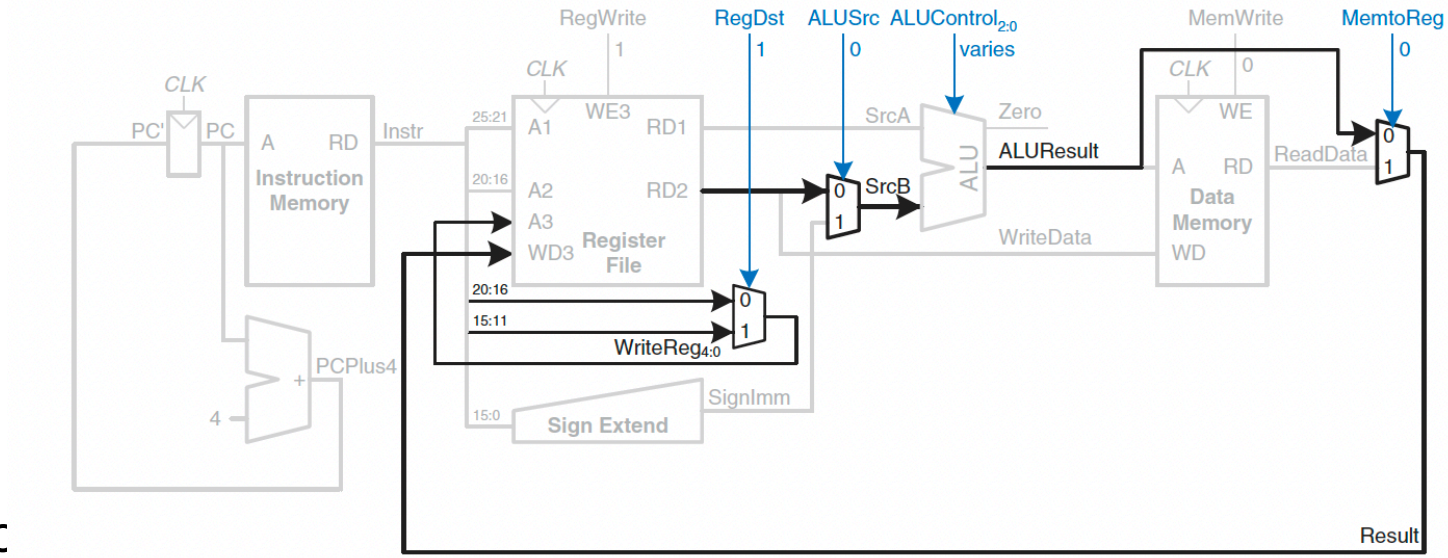
Single Cycle Data Processor: RType

- *ALUResult* is written to the register file
- But *ReadData* from Data Memory is also written to register file
- We need another multiplexer to select whether ALUResult or ReadData will be written to register file
- This is controlled by MemtoReg, which is 0 for R-Type and 1 for the lw instruction
- What about MemtoReg for `sw` instruction?
  - Don't care as `sw` does not write to register file



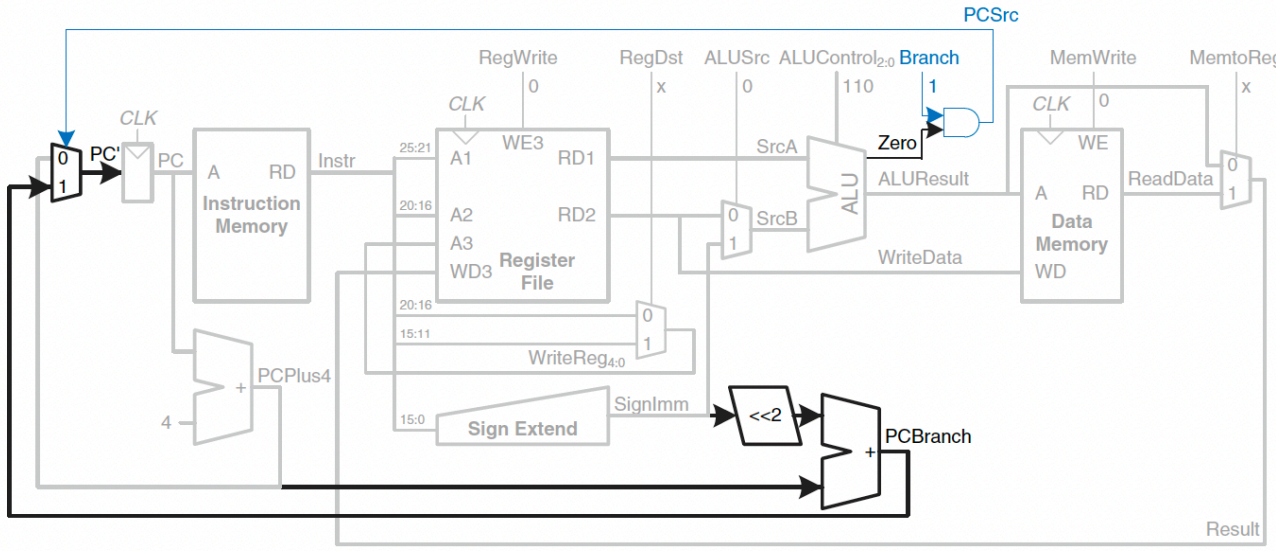
Single Cycle Data Processor: RType

- In R-type instructions the `rd` field (bits $\texttt{Instr}_{15:11}$) has the register address where the result will be written
- But in register file, the register address is always bits $\texttt{Instr}_{20:16}$
- Hence, we need another multiplexer with *RegDst* as control to select between `rd` fc R-type and rt for `lw` instruction
- *RegDst* is 0 for R-type and 1 for `lw`



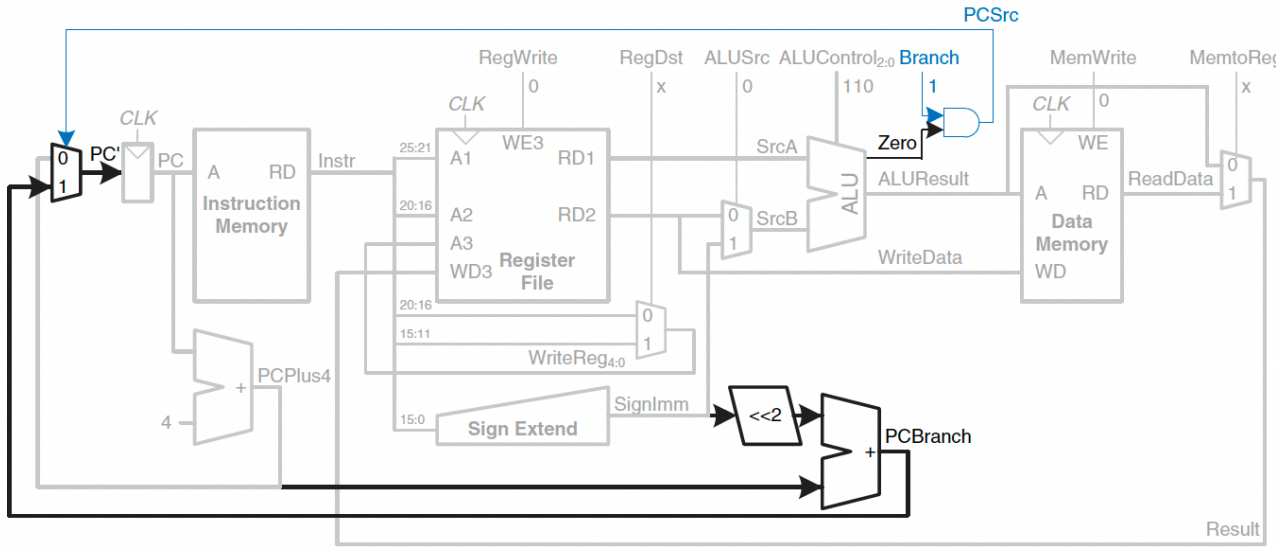
Single Cycle Data Processor: `beq`

- We extend the datapath with the `beq` instruction
- `beq` compares two registers; if equal branch is taken else next instruction is executed
- The actual comparison involves `SrcA - SrcB`
  - if *ALUResult* is 0 (*Zero* flag set to 1), then equal



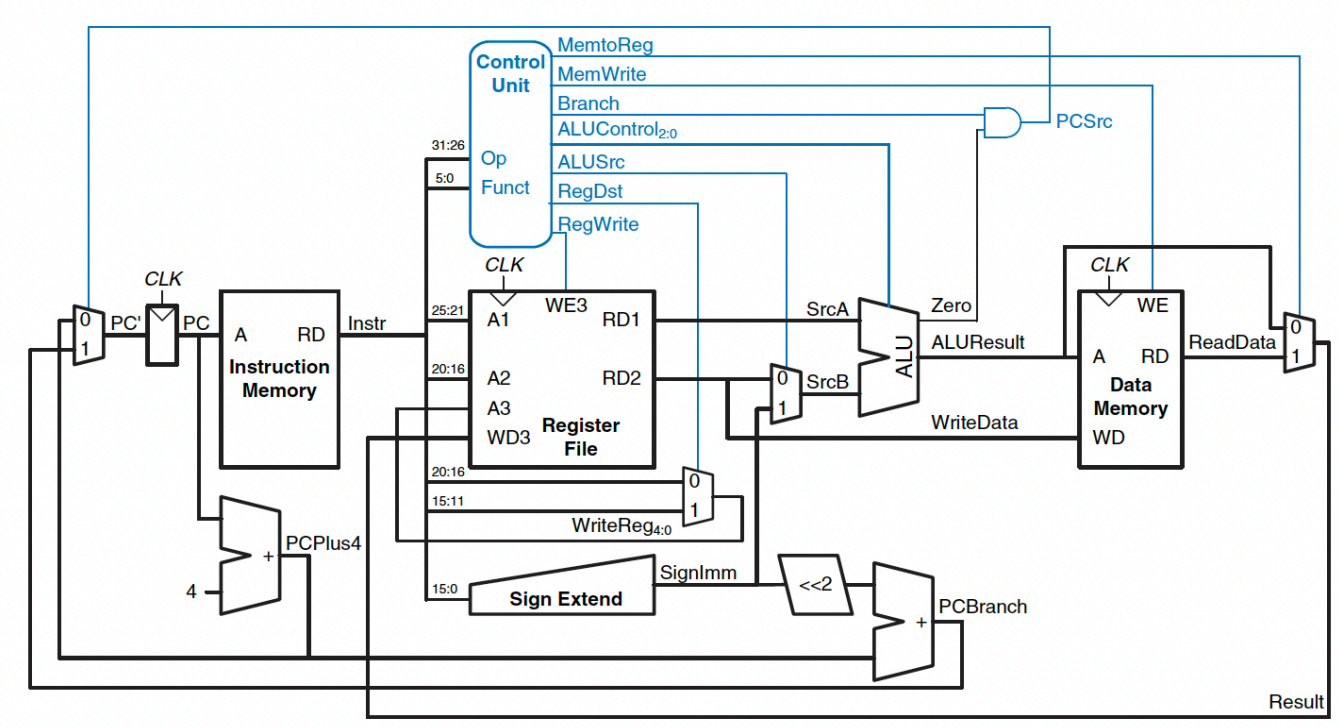
Single Cycle Data Processor: `beq`

- A branch is taken by adding the branch offset to the program counter. Recall the branch offset resides in the *imm* field of the instruction ($\texttt{Instr}_{15:0}$)
  - `PC' = PC + 4 + SignImm X 4`
  - Multiply by 4 is implemented as shift left by 2 (*PCBranch*)
- If operands are equal and instruction is branch (*Branch* = 1), then the next instruction comes from *PCBranch*; otherwise, *PCPlus4*



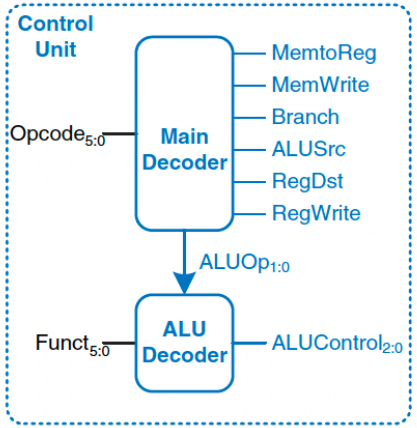
Single-Cycle Control

- The control unit controls the signals that determine the ALU operation
- Most control information comes from *opcode* but R-Type instructions use the *funct* field too



Single-Cycle Control

- We simplify the control unit by factoring out two blocks of combinational logic
  - The main decoder computes the output control signals along with a 2-bit *ALUOp* signal
  - The ALU decoder uses *ALUOp* and *funct* to compute the *ALUControl*
  - For our purposes, we will only have to know the meaning of the *ALUOp* signals



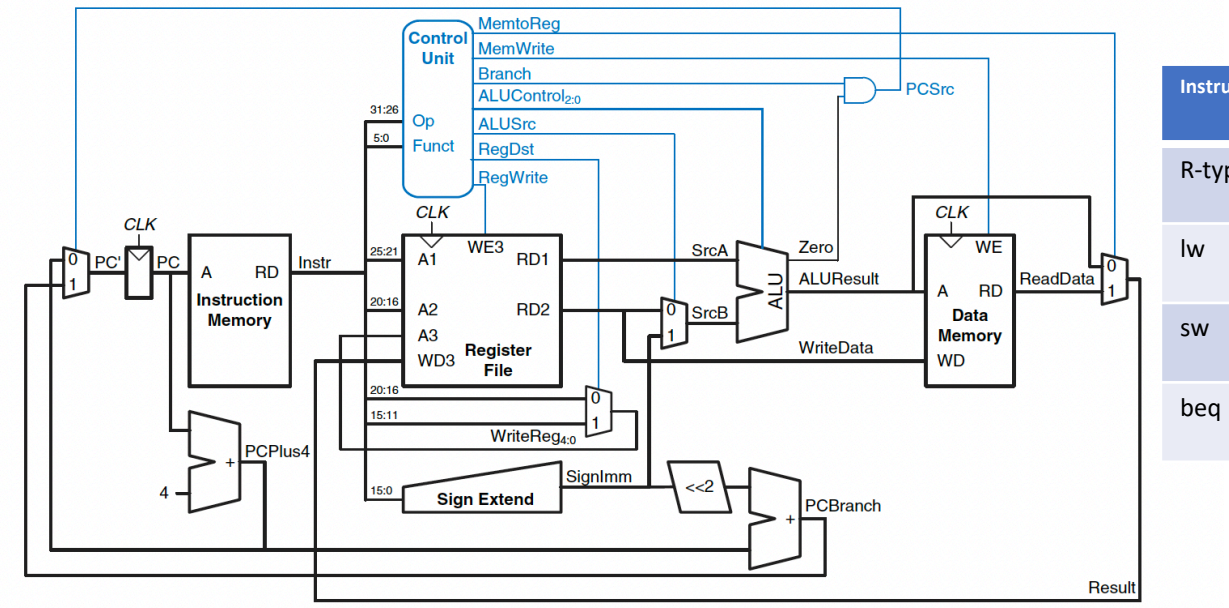
ALUOp	Meaning
00	add
01	subtract
10	look at `funct` field
11	n/a

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (`add`)	010 (add)
1X	100010 (`sub`)	110 (subtract)
1X	100100 (`and`)	000 (and)
1X	100101 (`or`)	001 (or)
1X	101010 (`slt`)	111 (set less than)

The *funct* fields for R-type instructions. X indicates don't care

Single-Cycle Control

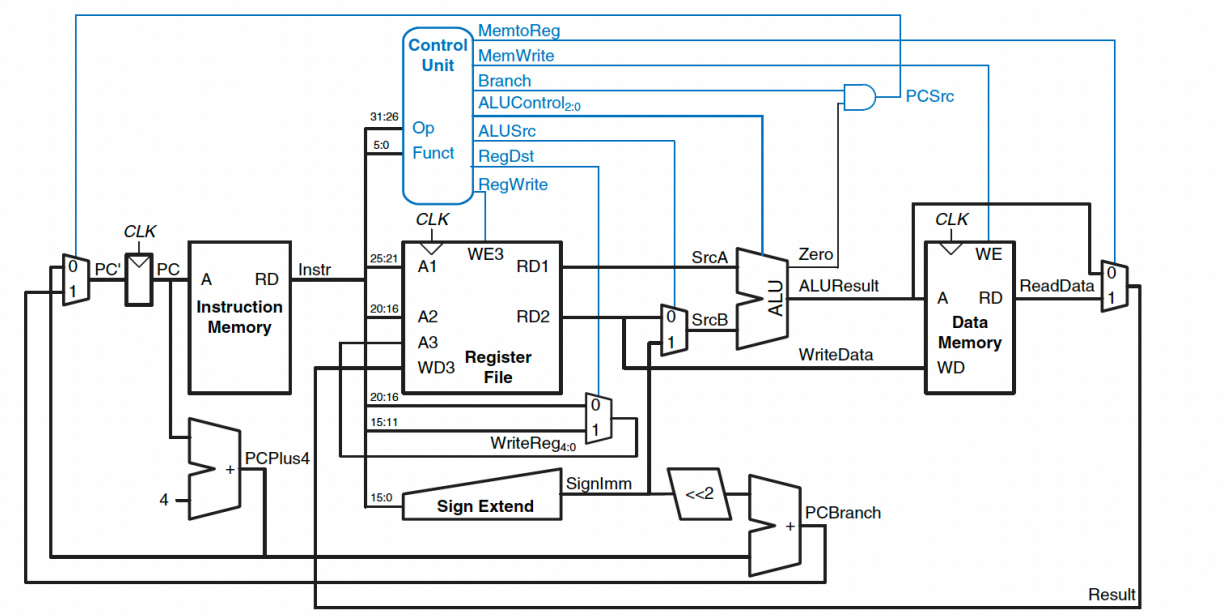
We determine the value for each control signal



Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000							
lw	100011							
sw	101011							
beq	000100							

Single-Cycle Control

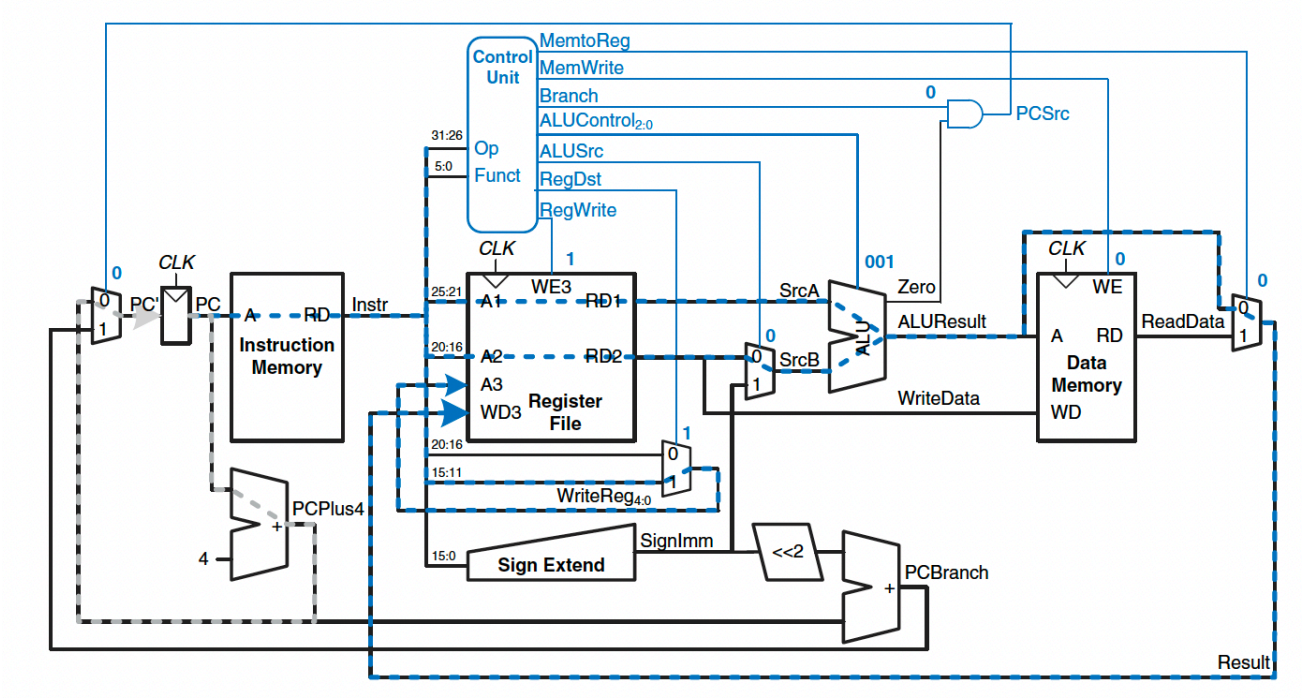
We determine the value for each control signal



Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Single-Cycle Example

- As an example, let us track the datapath and control path of the `or` instruction, which R-type
  - The main flow of data through the register file and ALU is represented with a dashed blue line
  - The updating of the PC is shown in dashed grey line

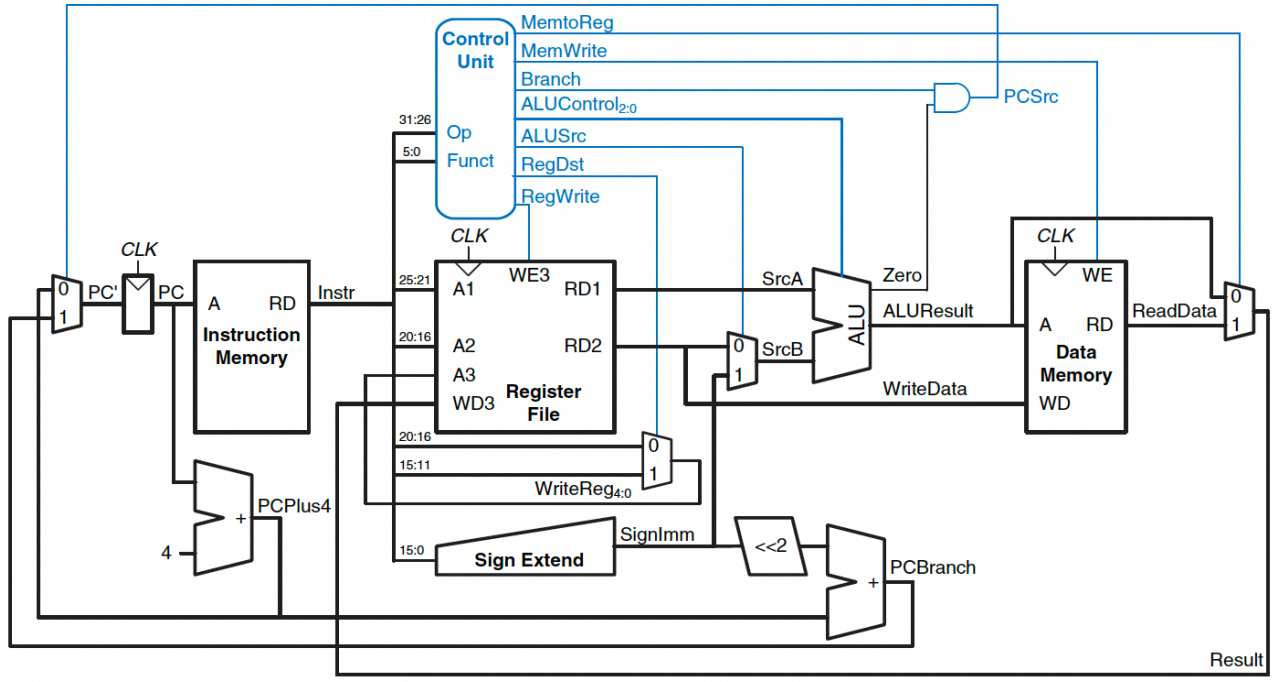


Extending The Single Cycle CPU

- So far, we have considered a limited set of instructions
- But we can extend the processor for more instructions
- As an example, we will extend the processor with the `addi` and `j` instructions

Extending The Single Cycle CPU

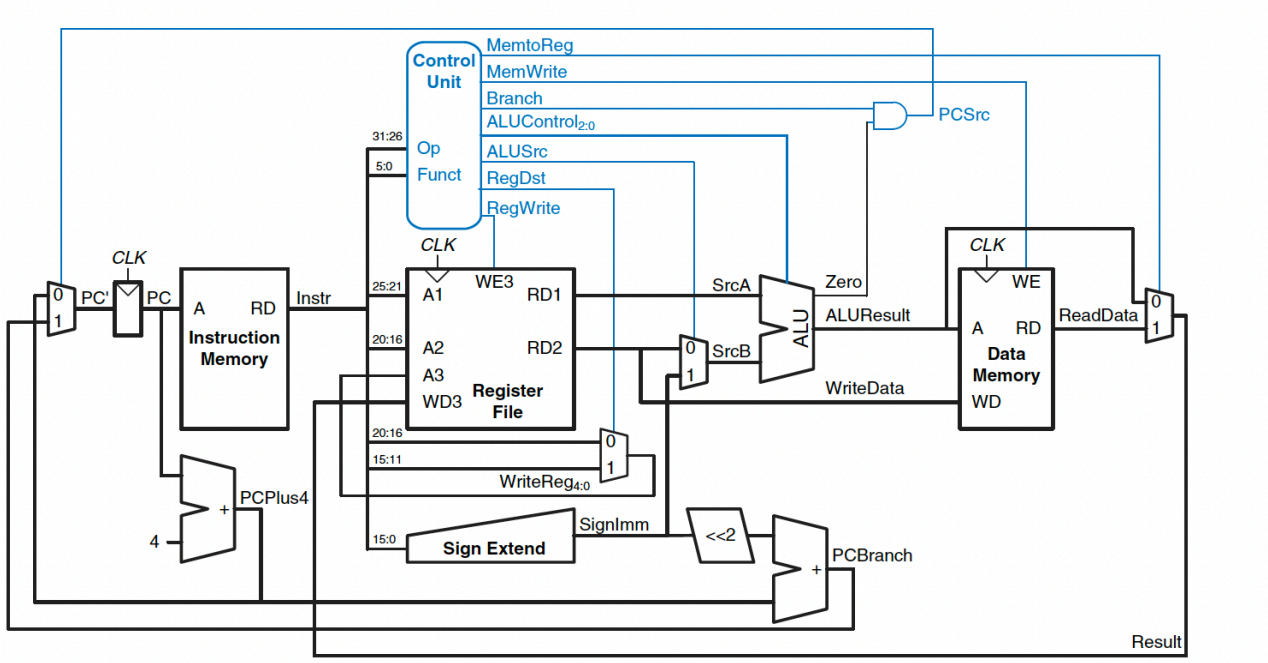
- The `addi` instruction adds the value in a register to an immediate and writes the result to another register.
  - The datapath for `addi` already exists. So, we don't need additional hardware!
  - We only have to determine the correct control signal values



Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
addi	001000							00

Extending The Single Cycle CPU

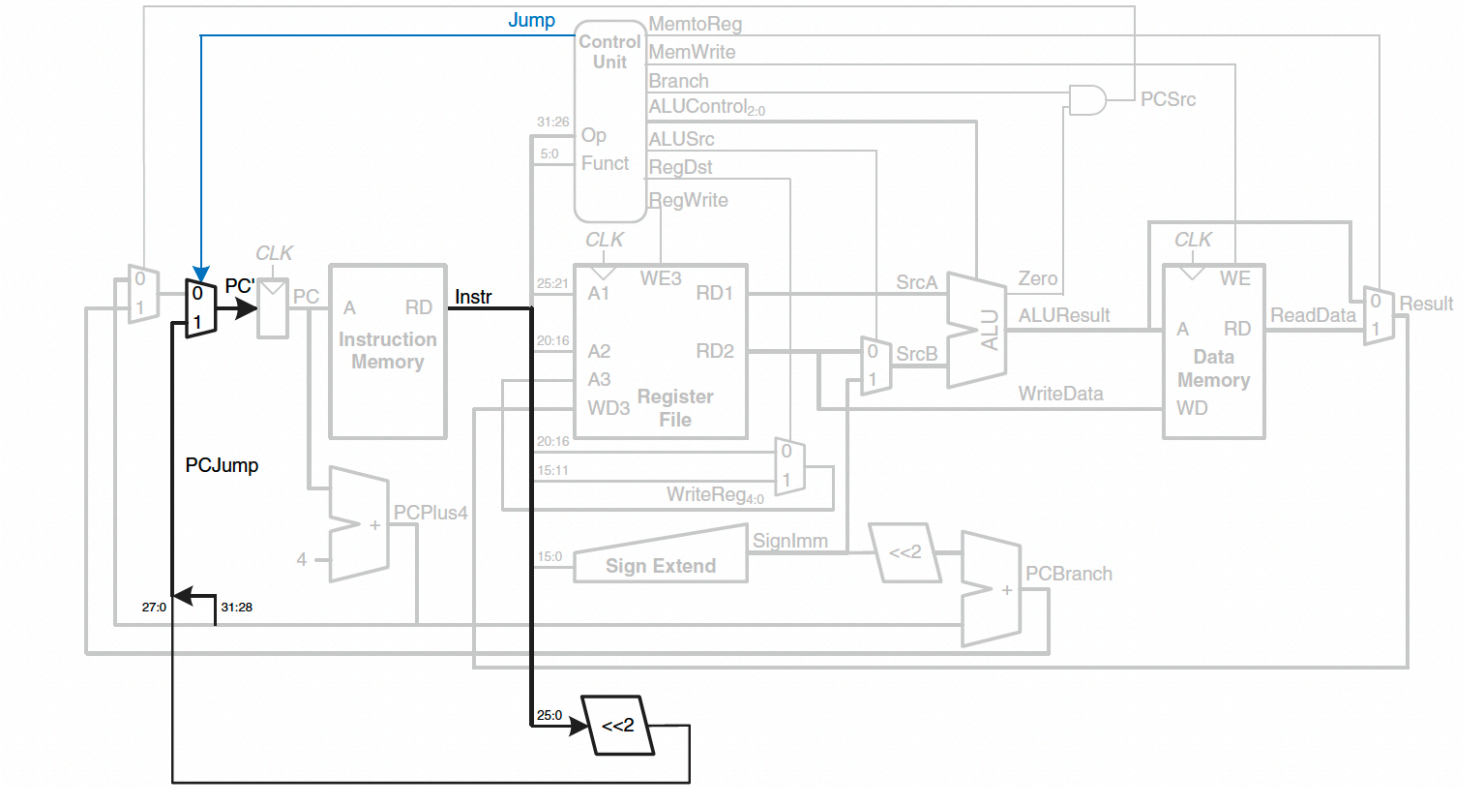
- The `addi` instruction adds the value in a register to an immediate and writes the result to another register.
  - The datapath for `addi` already exists. So, we don't need additional hardware!
  - We only have to determine the correct control signal values



Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
`addi`	001000	1	0	1	0	0	0	00

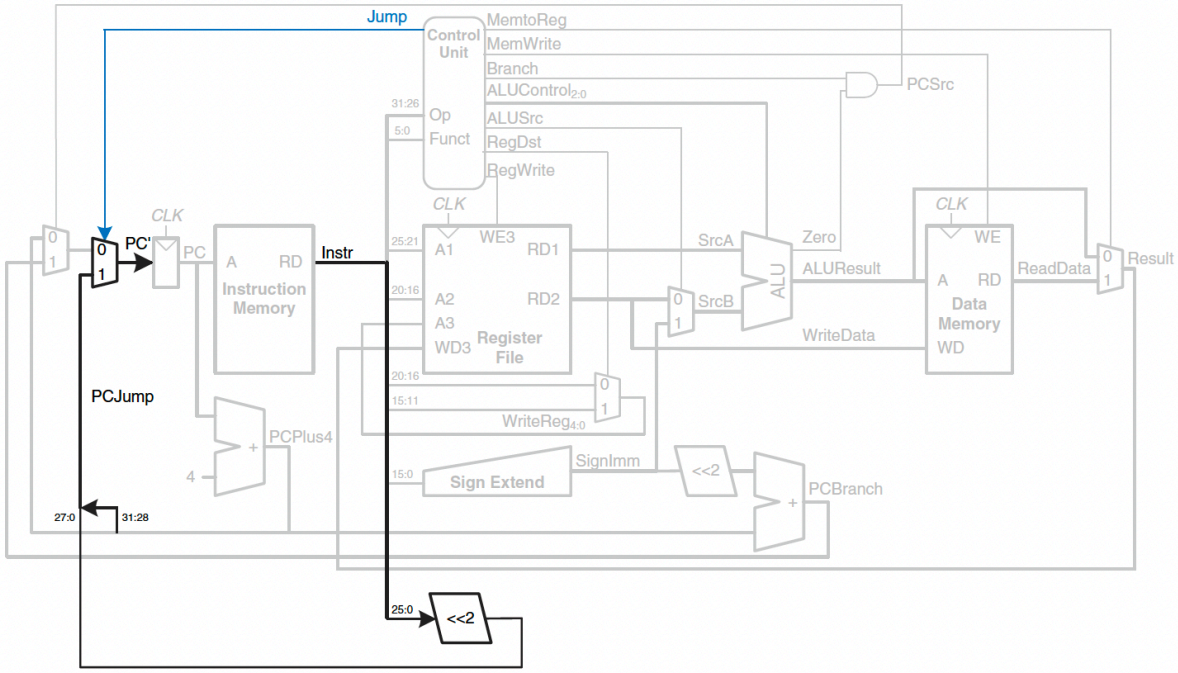
Extending The Single Cycle CPU

- The jump instruction, writes a new value to the PC
  - The 2 LSBs of the PC are always 0 (because word aligned, i.e., multiple of 4)
  - The next 26 bits are taken from the jump address field of the instruction
  - The upper 4 bits are taken from the old PC
- We don't have hardware to compute PC this way! What do we do?
  - Add hardware, specifically a multiplexer with a control signal *Jump*
  - *Jump = 1* for jump instructions and 0 for others



Extending The Single Cycle CPU

- Determine the control signals for jump



Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
j	000010	0	X	X	X	0	X	XX	1

Points To Note

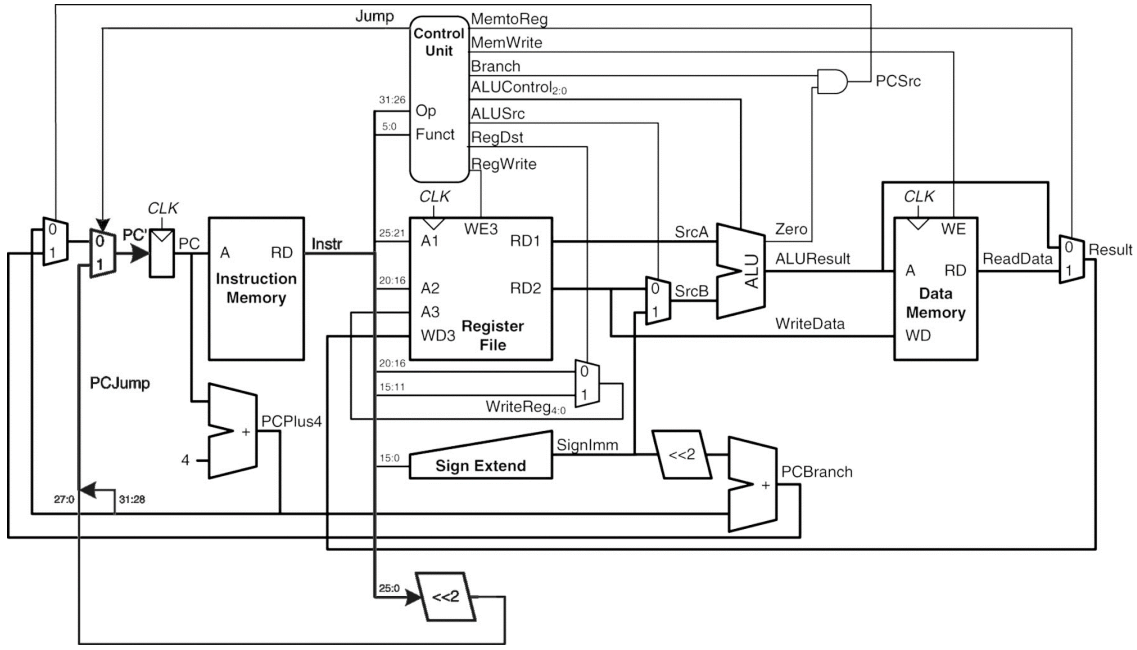
- When we build the hardware for a particular instruction we focus on additional hardware and the control signal values.
- This does not mean the datapath for existing instructions are switched off!
- Signals are always travelling thru the wires.
- It is the engineer/designer’s job to control the signals in a way that does not cause the CPU to malfunction.

Recap

So far, we have built a single cycle processor for a few instructions

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
j	000010	0	X	X	X	0	X	XX	1

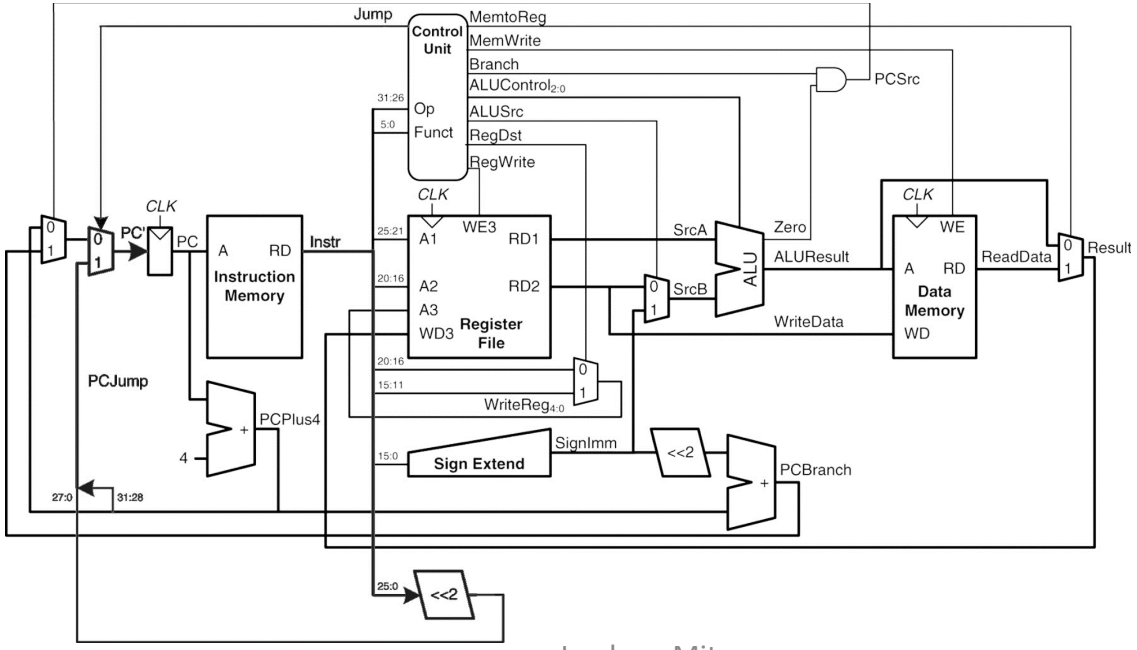


Performance Analysis

- Recall, we mentioned that different microarchitectures have different performance and cost tradeoffs
- There are many ways to measure performance; marketing departments infamously choose the measure that makes their computer look fastest!
- The only gimmick-free way to measure performance is to measure the execution of a program of interest to us or benchmarks
  - `Execution Time = (# instructions)(cycles/instruction)(seconds/cycle)`
- The no. of cycles per instruction is called the CPI
  - For single-cycle CPUs, CPI = 1
- The no. of seconds per cycle is called the clock period, T_c
  - Determined by critical path thru the logic on the processor

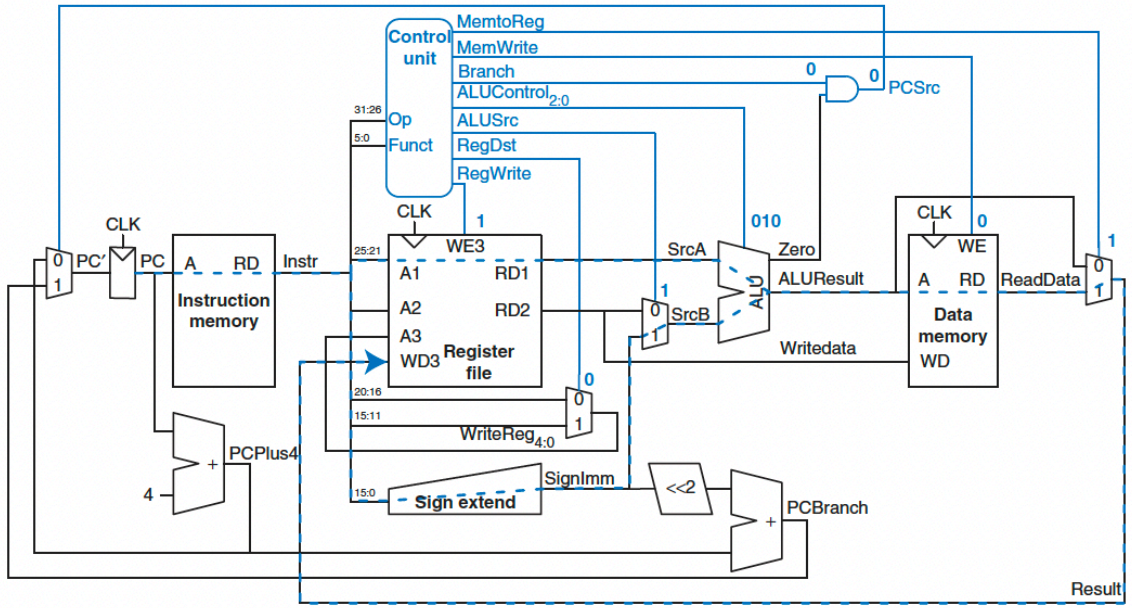
Performance Analysis

- Suppose we want to determine the clock period of the single cycle processor that we have been implementing
- We need to compute the delay along the critical path of the slowest instruction, which happens to be the `lw` instruction



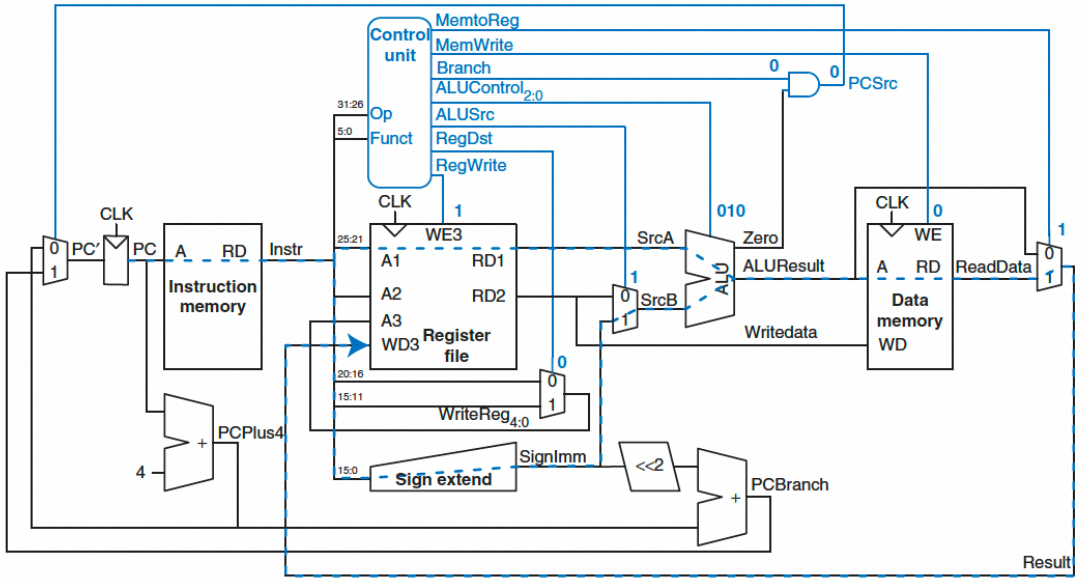
Critical Path of `lw` Instruction

- It starts with the PC loading a new address at the rising edge of the clock
- The instruction memory reads the next instruction
- The register file reads *SrcA*
- While register file is read, the immediate is sign extended and is available at the *ALUSrc* MUX to determine *SrcB*
- The ALU adds *SrcA* and *SrcB* to find effective address
- The data memory reads from this address
- The *MemtoReg* MUX selects *ReadData*
- Finally, *Result* setup at the register file before the rising edge of the clock



Critical Path of `lw` Instruction

- It starts with the PC loading a new address at the rising edge of the clock ($\mathbf{t_{pcq\_PC}}$)
- The instruction memory reads the next instruction ($\mathbf{t_{mem}}$)
- The register file reads *SrcA* ($\mathbf{t_{RFread}}$)
- While register file is read, the immediate is sign extended and is available at the *ALUSrc* MUX to determine *SrcB* ($\mathbf{t_{sext} + t_{mux}}$)
- The ALU adds *SrcA* and *SrcB* to find effective address ($\mathbf{t_{ALU}}$)
- The data memory reads from this address ($\mathbf{t_{mem}}$)
- The *MemtoReg* MUX selects *ReadData* ($\mathbf{t_{mux}}$)
- Finally, *Result* setup at the register file before the rising edge of the clock ($\mathbf{t_{RFsetup}}$)



Critical Path of `lw` Instruction

- In most implementations, the ALU, memory, register file accesses are slower than other operations.
- Therefore, the cycle time simplifies:
  - $T_c = t_{pcq\_PC} + t_{mem} + t_{RFread} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
- We don't consider $\mathbf{t_{sext} + t_{mux}}$ and the time to compute the new PC. Why?
  - Because they happen in parallel with the other slower operations
  - Slower operations dominate Now, recall there are other instructions for which we can come up with a formula
- But they have shorter cycle times. E.g., R-type instructions do not access memory
- The clock period must be long enough to accommodate the slowest instruction

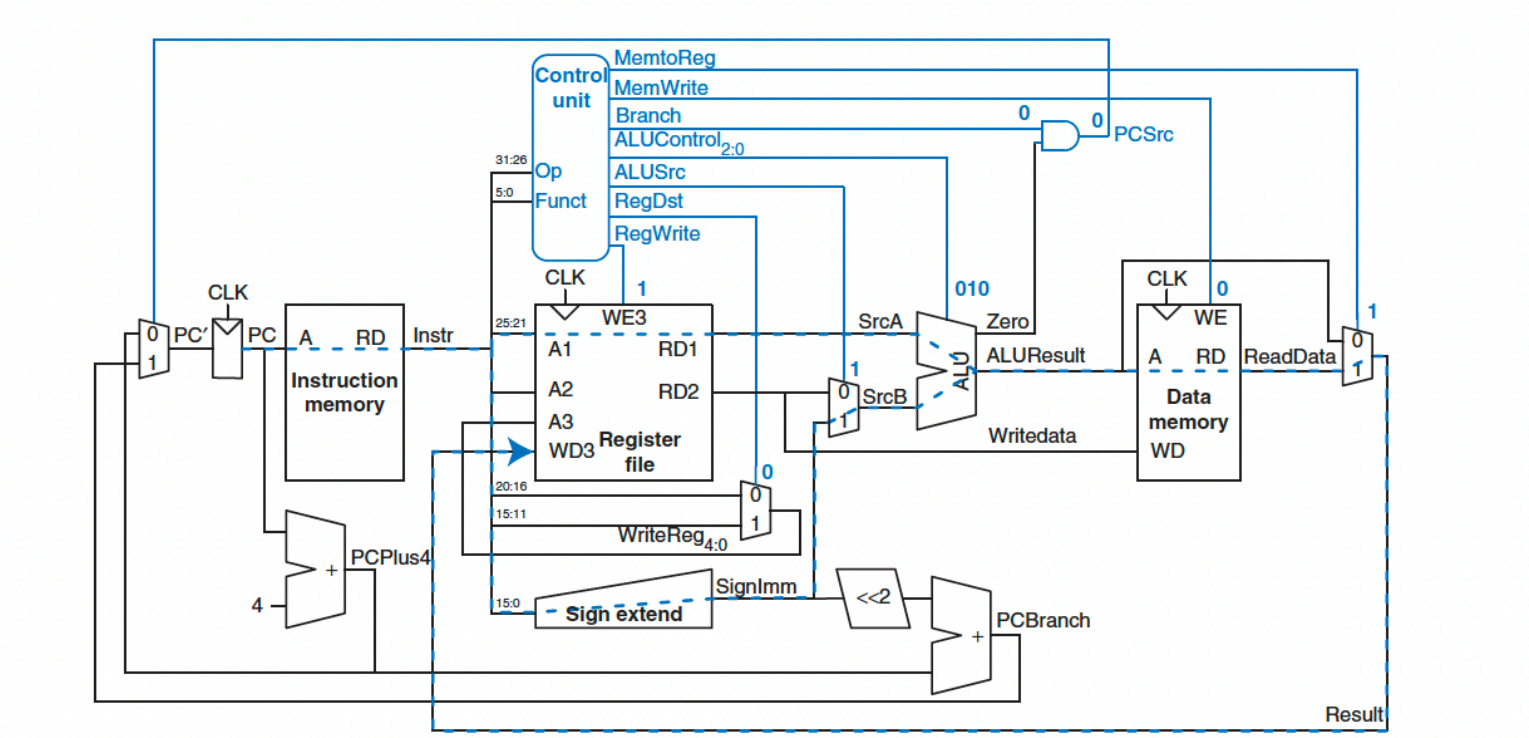
Critical Path of `lw` Instruction

- Generally, you can assume that the delays for each of the circuit elements in our processor is known
- Hence, we can concretize the clock cycle now that we know the values

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	200
memory read	t_{mem}	250
register file read	$t_{RF\text{read}}$	150
register file setup	$t_{RF\text{setup}}$	20

Critical Path of `lw` Instruction

- We can use the formula we came up with.



$T_c = 30 + 250 + 150 + 200 + 250 + 25 + 20$
$= 925\ ps$

$$T_c = 30 + 250 + 150 + 200 + 250 + 25 + 20$$
$$= 925\ ps$$

Critical Path of `lw` Instruction

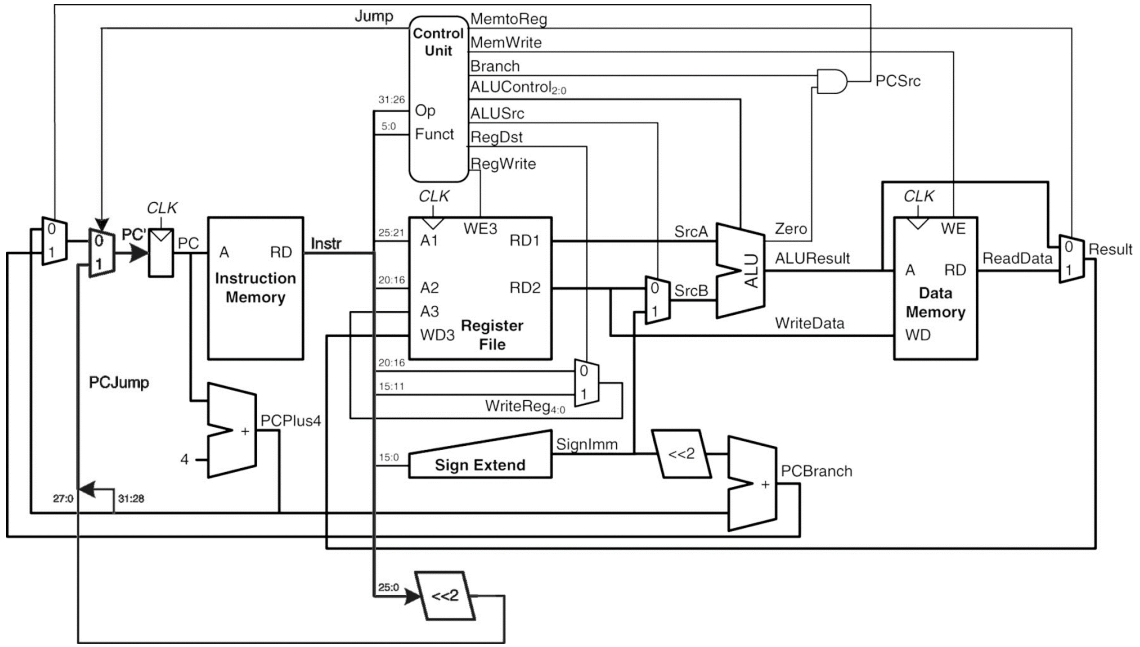
- Suppose we want to compute the execution time (T_1) of a program with 100 billion instructions
- Assuming that `lw` is the slowest instruction, we get:

$T_c = 30 + 250 + 150 + 200 + 250 + 25 + 20 = 925\ ps$

$T_1 = (100 * 10^9)(1\ \text{cycle/instruction})(925 * 10^{-12}\text{s/cycle}) = 92.5\ \text{seconds}$

Critical Path of R-Type Instruction

- Let's compute the critical path delay (propagation delay) for R-type instructions in the datapath



t_{pcq}	I-Mem Read	Adder	MUX	ALU	RegFile Read	RegFile Write	D-Mem Read	D-Mem Write	SignExt	Shifter
30	40	5	3	20	30	45	50	40	4	6

Critical Path of R-Type Instruction

- Determine he cycle time formula:
  - $T_c = t_{pcq\_PC} + t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{mux} + t_{RFwrite}$
- Ignore $\mathbf{t_{sext} + t_{mux}}$ and the time to compute the new PC because negligible
- $T_c = 30 + 40 + 30 + 3 + 20 + 3 + 45 = 171$

Custom Instructions

- We will now extend our single cycle datapath with an instruction that we have invented!
- `xcond $rs ,$rt, immediate` is an I-type instruction defined as follows:

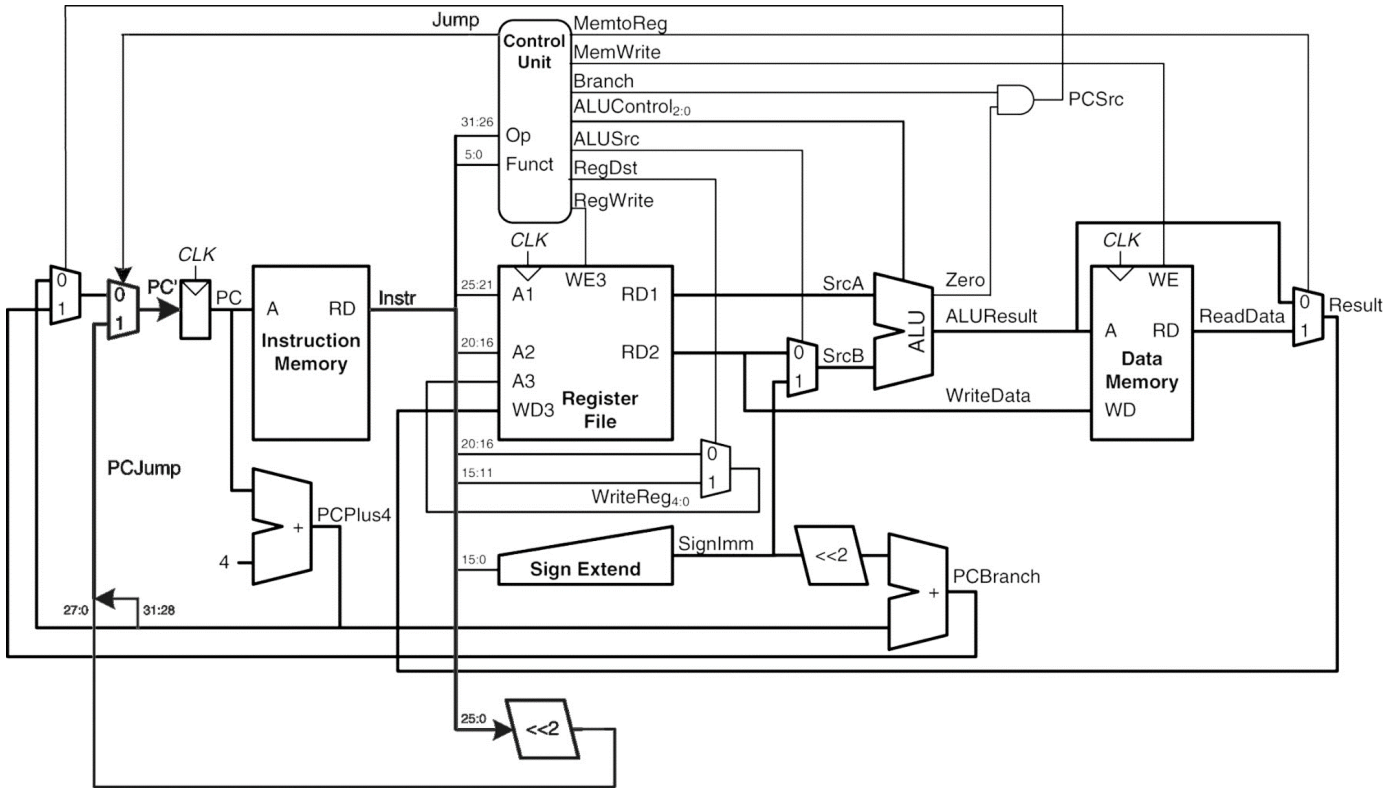
```
if Reg[$rt] >= 0 then
     Mem[SignImm] = Reg[$rs]
else
     do nothing
PC' = PC + 4
```

- How should we implement this?
  - Notice that `xcond` is a conditional memory write
  - This means that write to memory happens only when the condition true
  - Hence, we need to optionally switch on or switch off the MemWrite signal
  - We also need to specify the write address and the data that will be written
- Hardware assumptions:
  - Assume that we can add at most one logical gate and three 2:1 multiplexers

Custom Instructions

- `xcond $rs ,$rt, immediate` is an I-type instruction defined as follows:

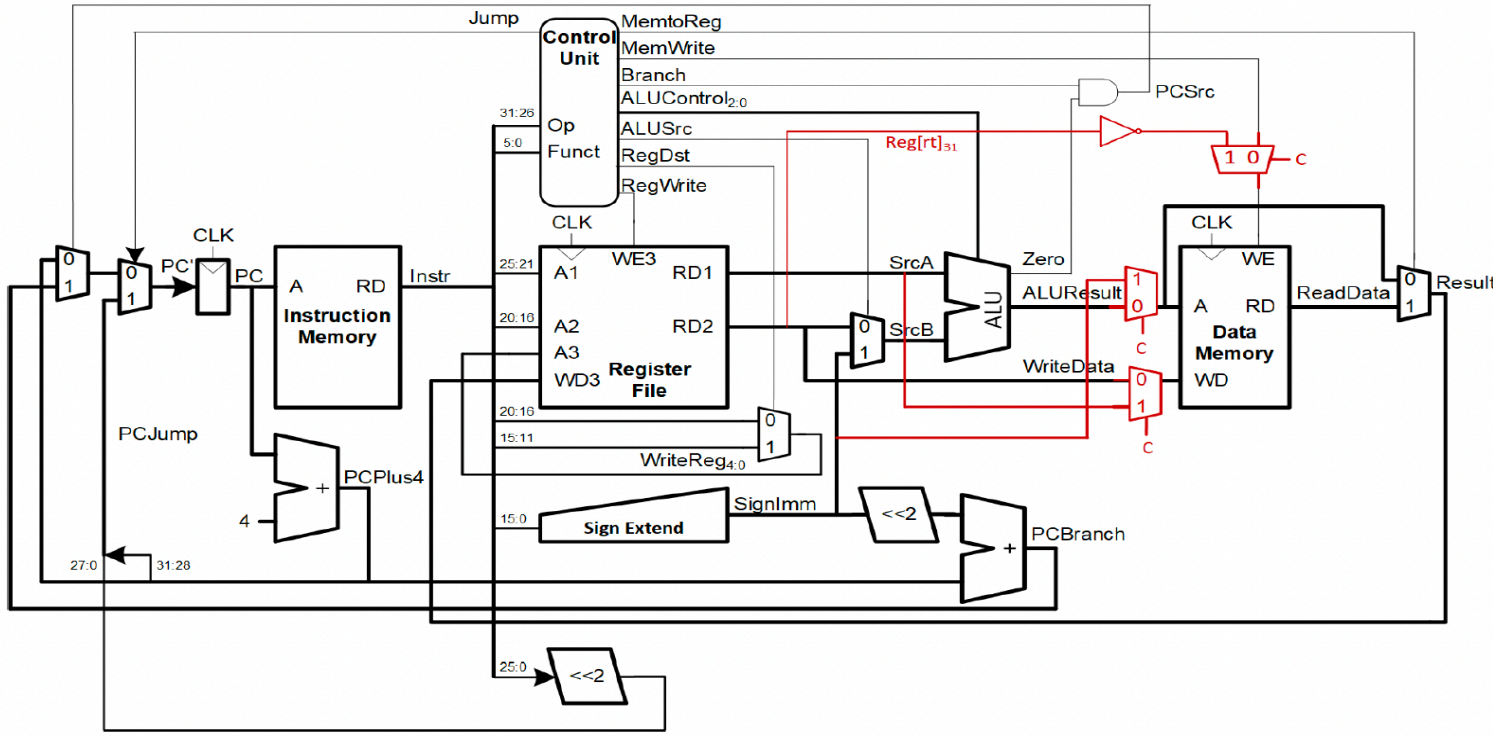
```
if Reg[$rt] >= 0 then
     Mem[SignImm] = Reg[$rs]
else
     do nothing
PC’ = PC + 4
```



Custom Instructions

- `xcond $rs ,$rt, immediate` is an I-type instruction defined as follows:

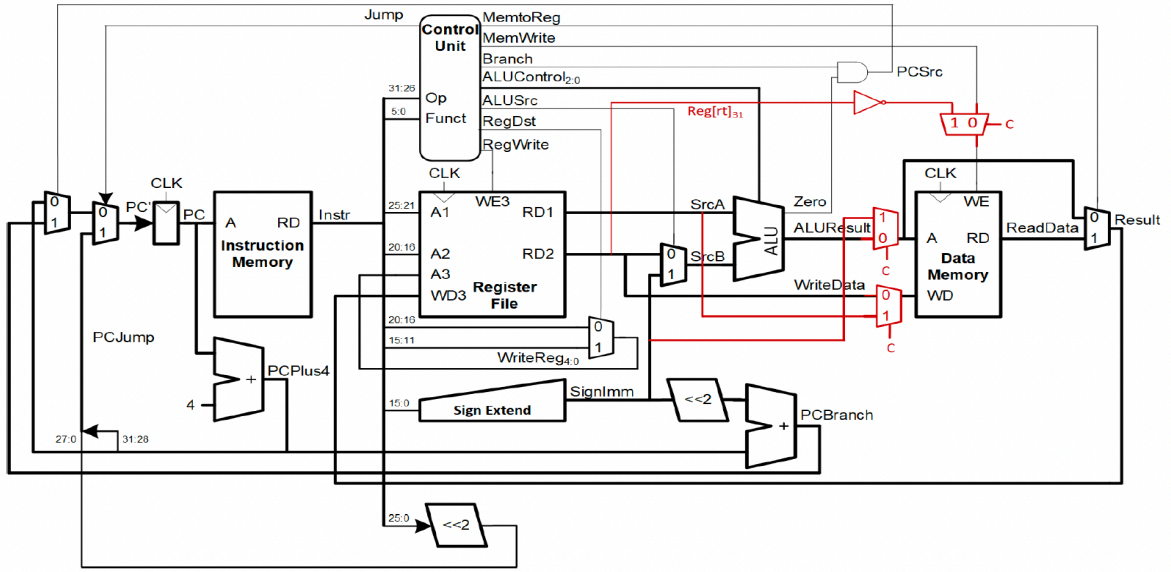
```
if Reg[rt] >= 0 then
      Mem[SignImm] = Reg[$rs]
else
      do nothing
PC' = PC + 4
```



Custom Instructions

- `xcond $rs ,$rt, immediate` is an I-type instruction defined as follows:

```
if Reg[rt] >= 0 then
      Mem[SignImm] = Reg[$rs]
else
      do nothing
PC' = PC + 4
```



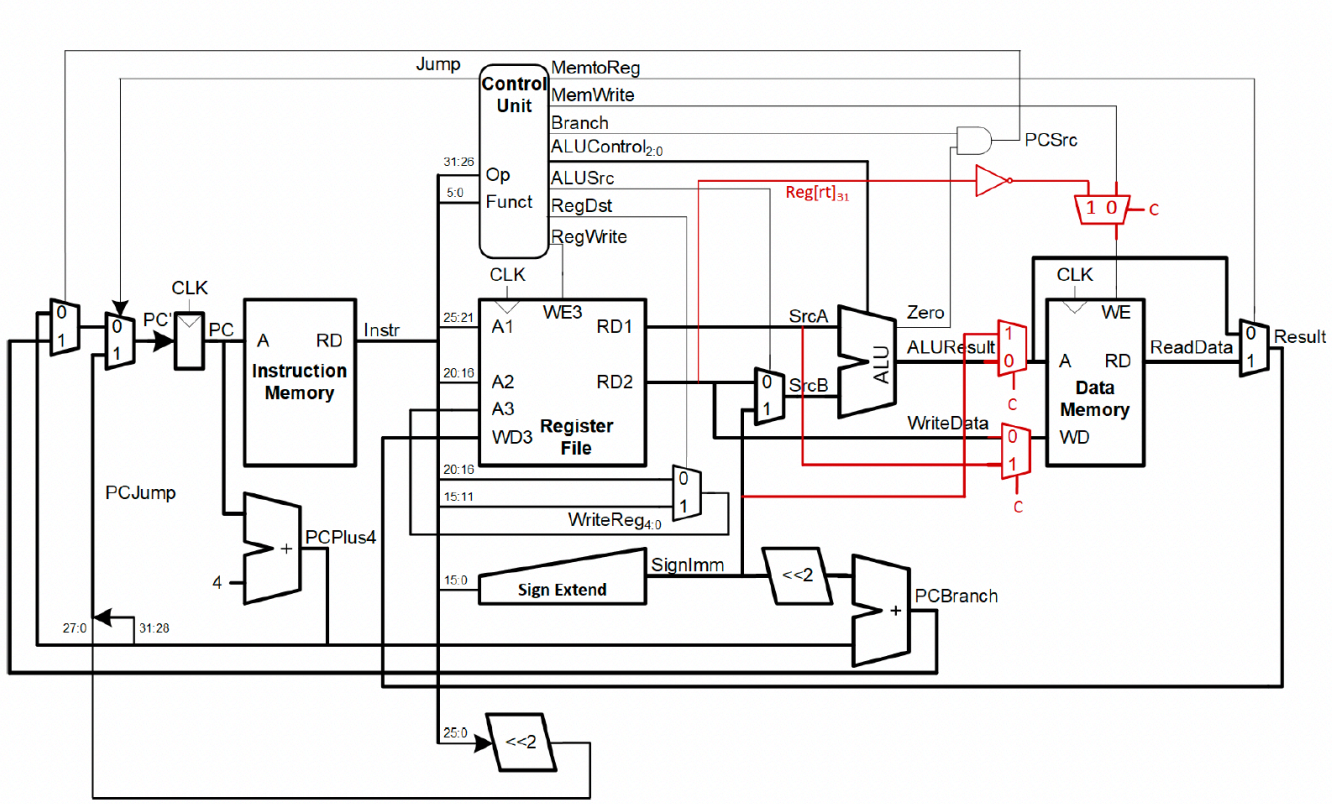
Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	Branch	ALUOp	Jump	C
R-type	1	0	0	1	0	0	10	0	0
lw	0	1	1	1	0	0	00	0	0
sw	X	1	X	0	1	0	00	0	0
addi	0	1	0	1	0	0	00	0	0
beq	X	0	X	0	0	1	01	0	0
j	X	X	X	0	0	X	XX	1	0
xcond									

Custom Instructions

- `xcond $rs ,$rt, immediate` is an I-type instruction defined as follows:

```
if Reg[rt] >= 0 then
      Mem[SignImm] = Reg[$rs]
else
      do nothing
PC' = PC + 4
```

Type	RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	Branch	$ALUOp_{1:0}$	Jump	c
R-type	1	0	0	1	0	0	10	0	0
lw	0	1	1	1	0	0	00	0	0
sw	X	1	X	0	1	0	00	0	0
addi	0	1	0	1	0	0	00	0	0
beq	X	0	X	0	0	1	01	0	0
j	X	X	X	0	0	X	XX	1	0
xcond	X	X	X	0	X	0	XX	0	1



Rules For Adding Instructions

- Determine the type of instruction (R-type, I-type, or J-type).
- Compare the new instructions with the existing instructions in the datapath.
- Ask yourself if the existing datapath is enough or additional hardware is required.
- If existing datapath is enough, then analyze the control signal values needed to implement the instruction.
- If additional hardware is needed, then try and use minimal hardware.
- Ensure that the new datapath still works for existing instructions!

Custom Instructions

- Imagine a MIPS instruction that would take a branch if the value in **`rs`** were not a multiple of 4: `bnmul4 rs, label`

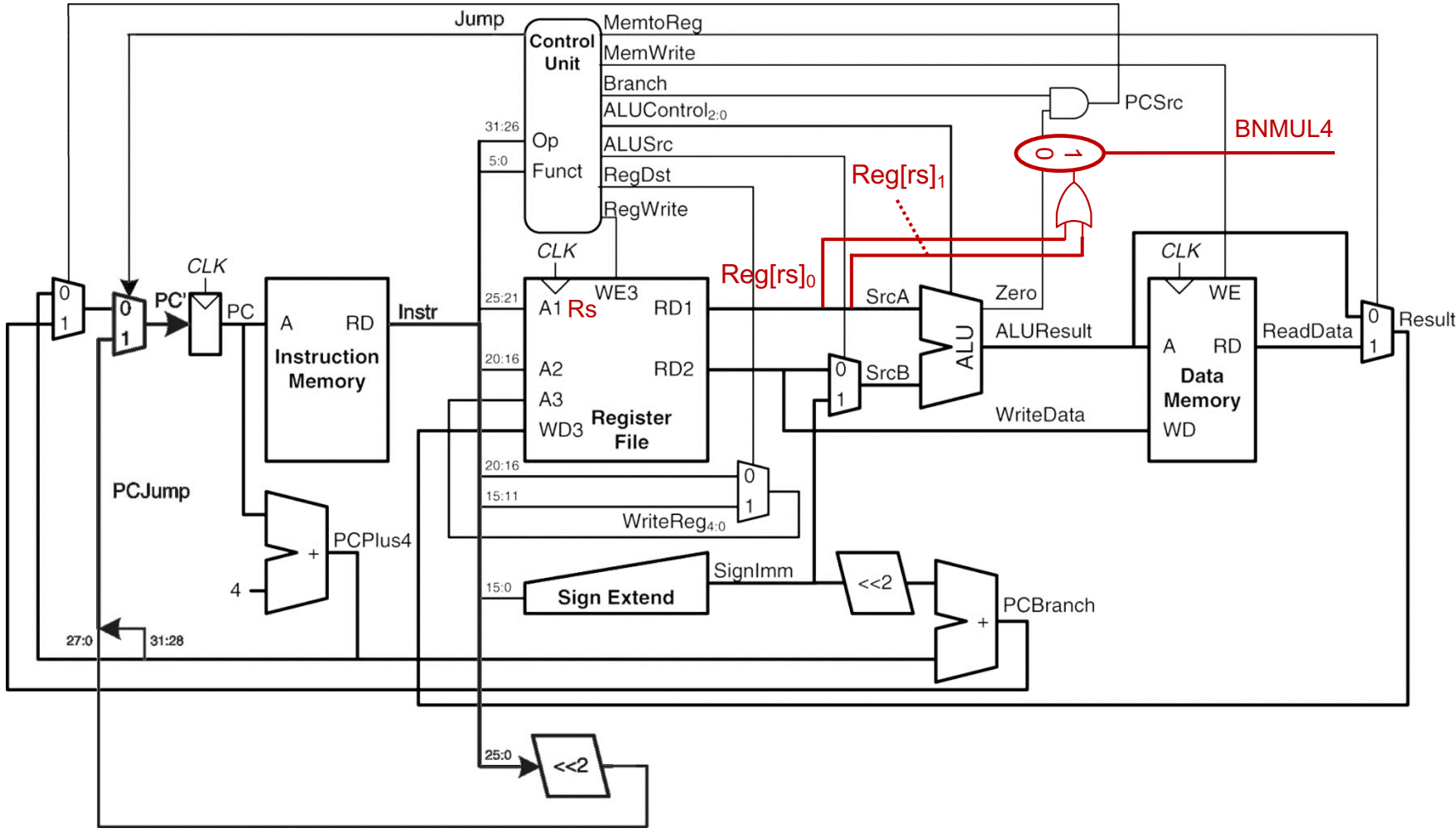
```
if Reg[rs] is NOT a multiple of 4 then
   PC' = PC+4 + SignImm×4
else
   PC'= PC+4
```

- How could we implement this? What hardware and logic would we need to add to the single-cycle datapath and control unit?
- The binary representation of an integer (positive or negative) that is a multiple of four will end in 00
- So, we will branch if either or both of the two least significant bits are 1s: **`Reg[rs][0] = 1 or Reg[rs][1] = 1`**

Custom Instructions

```
bnmul4 rs, label:

if Reg[rs] is NOT a multiple of
then
    PC' = PC+4 + SignImm×4
else
    PC'= PC+4
```



Custom Instructions

- Control signals