The Processor: Datapath and Control

Introduction

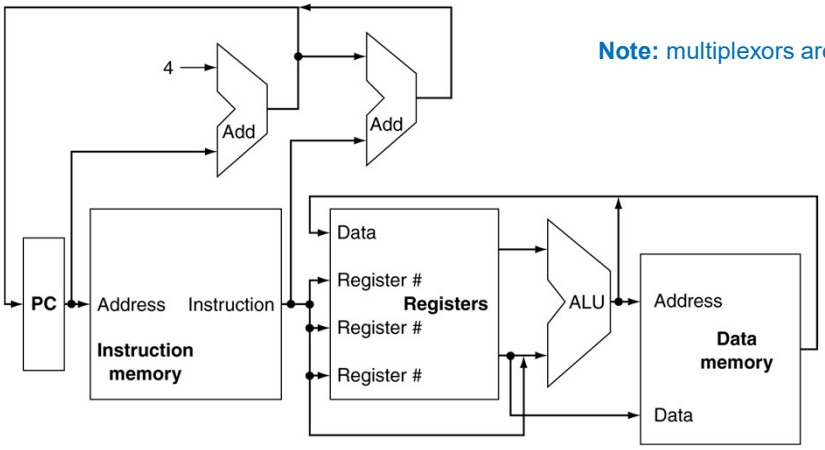
- CPU performance factors (CPU time = IC × CPI ×T_c)
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

- We will examine MIPS implementation supporting
  Memory reference: `lw`, `sw`
  Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  Control transfer: `beq`, `j`

Implementation Overview

- We need memory
  - to store instructions
  - to store data
  - for now, let's make them separate units
- We need registers, ALU, and a whole lot of control logic
- CPU operations common to all instructions:
  - use the program counter (PC) to pull instruction out of instruction memory
  - read register values

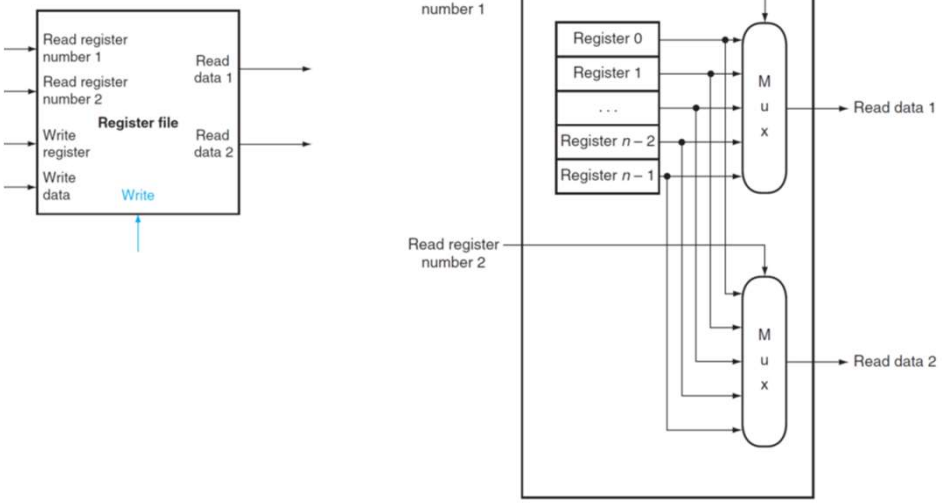
CPU (Single Cycle) Overview



Note: multiplexors are not shown

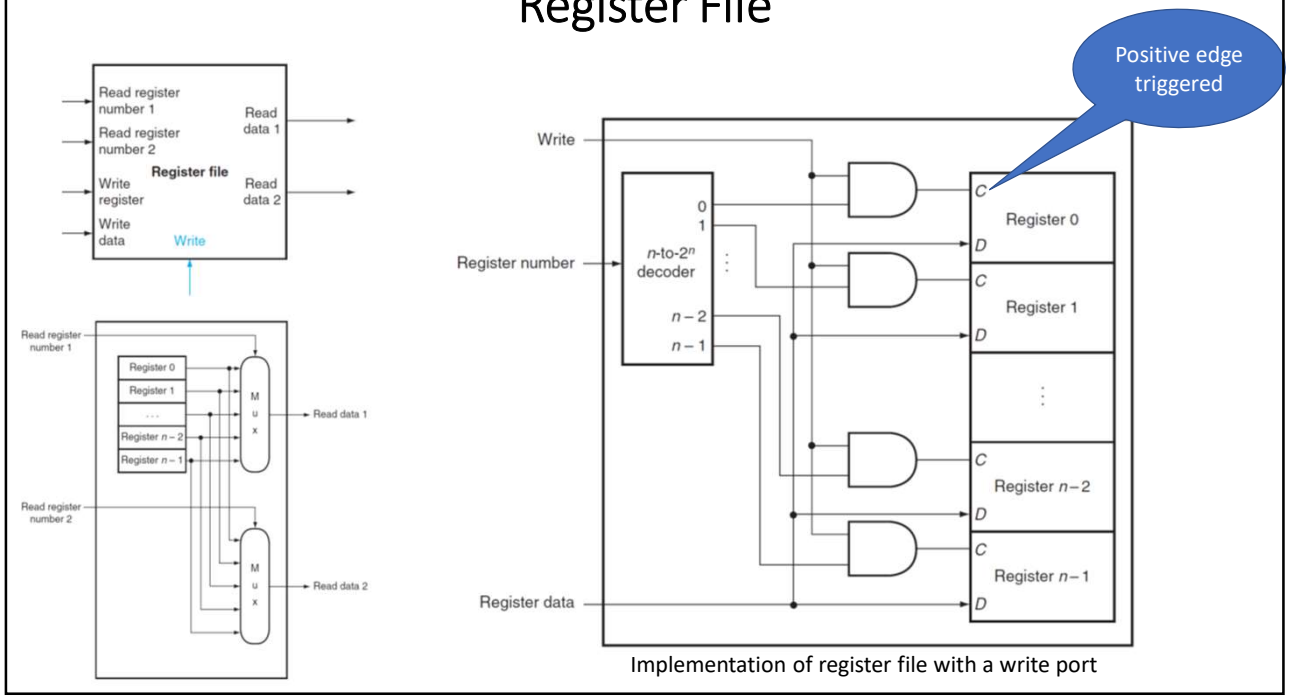
- What is the role of the Add units?
- Explain the inputs to the data memory unit
- Explain the inputs to the ALU
- Explain the inputs to the register unit
- Which of the above units need a clock?
- What is being saved (latched) on the rising edge of the clock? Keep in mind that the latched value remains there for an entire cycle

Register File



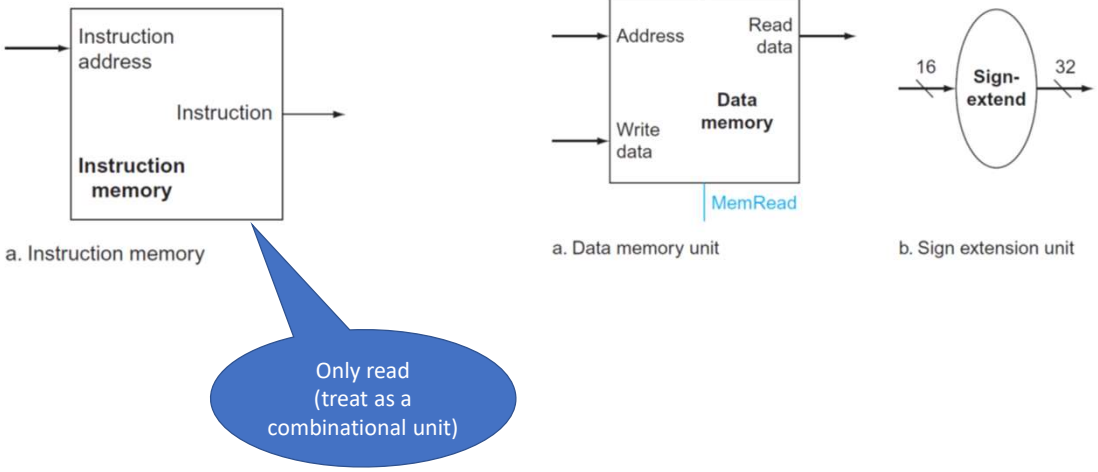
Implementation of register file with two read ports

Register File

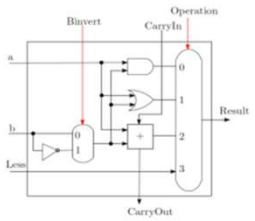


Implementation of register file with a write port

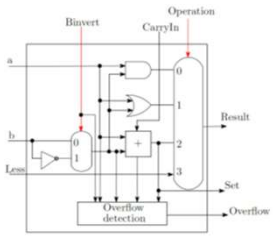
Data Memory Units



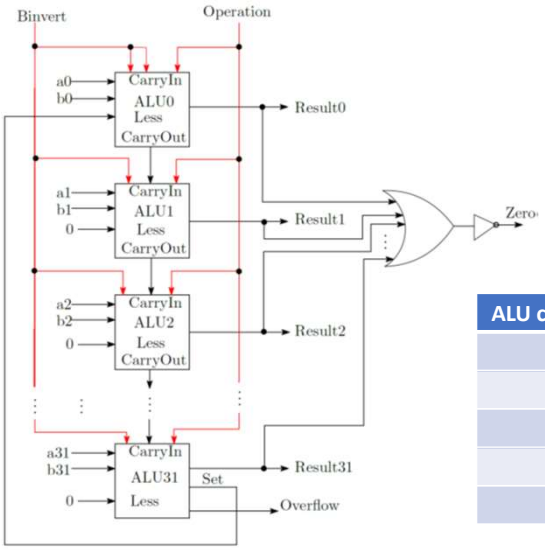
ALU

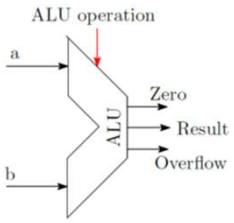


ALU 0 – ALU 30



ALU 31



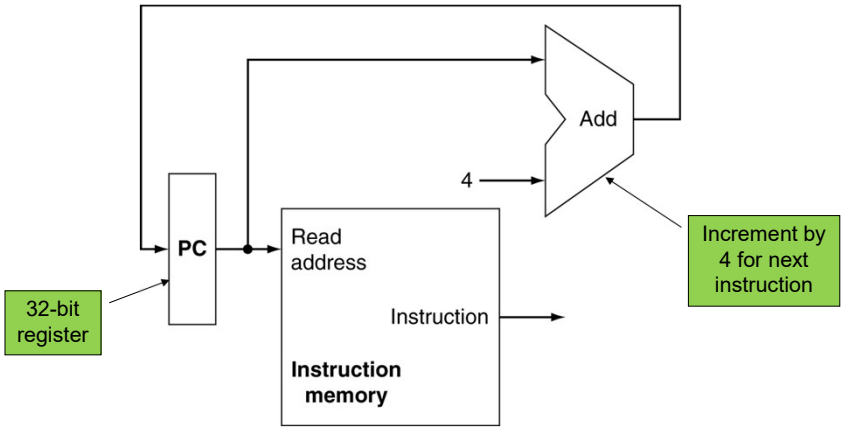


ALU control lines	Function
000	and
001	or
010	add
110	subtract
111	Set on less than

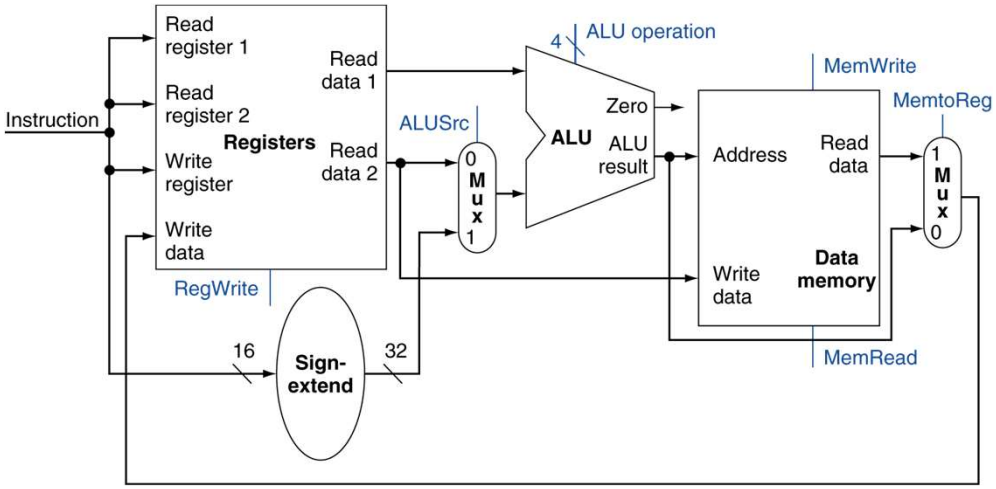
Supporting conditional branch: beq rs, rt, L1 -- if (rs == rt) branch to instruction labeled L1

Idea: (a-b) = 0 → a = b

Datapath: Instruction Fetch



Datapath: R-Format and Load/Store



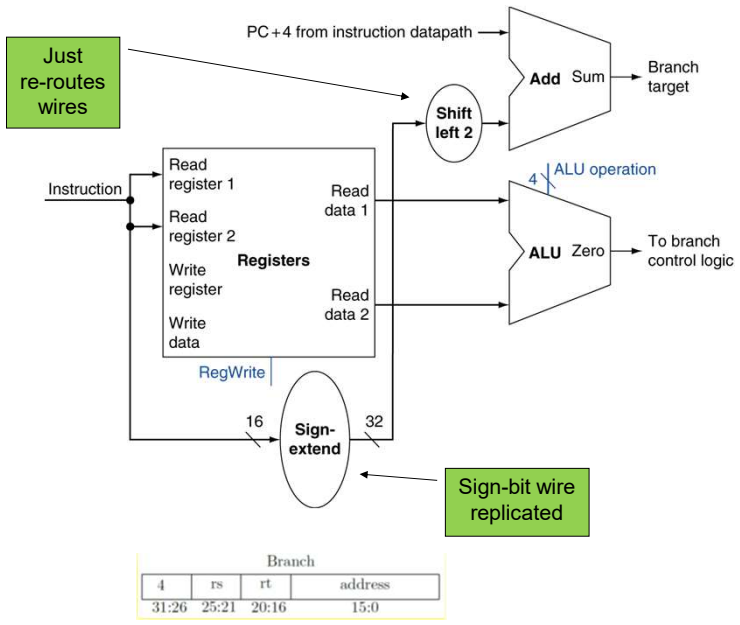
R-type

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

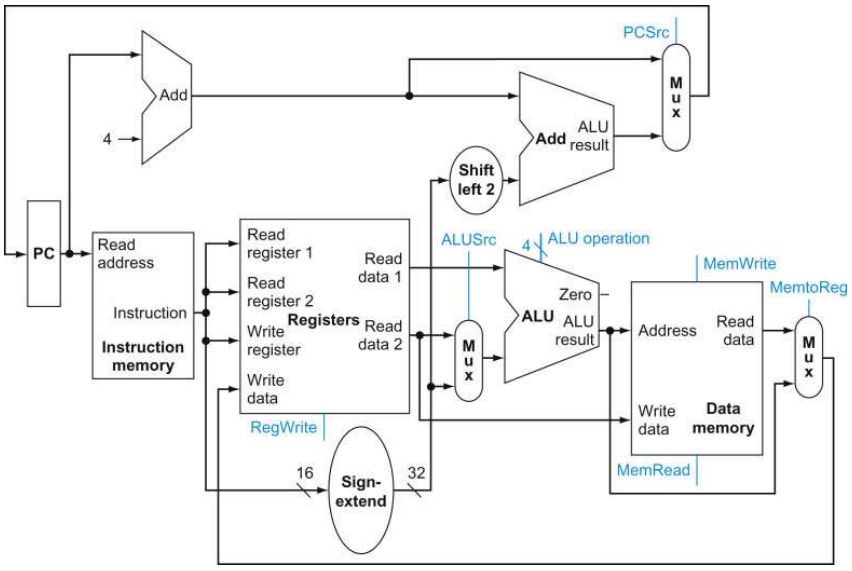
Load/Store

35 or 43	rs	rt	address
31:26	25:21	20:16	15:0

Datapath: Branch



Single Cycle Datapath



Datapath and the Clock

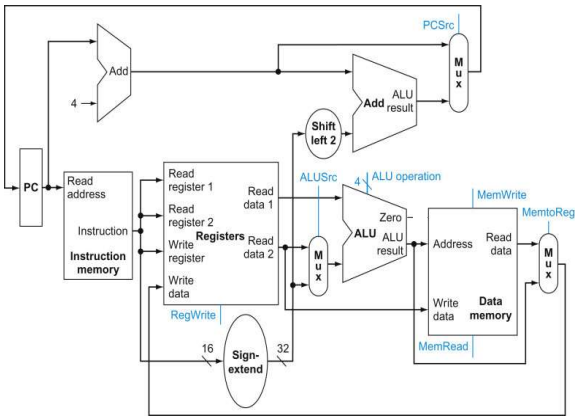
STEP 1: A new instruction is loaded from memory. The control unit sets the datapath signals appropriately so that

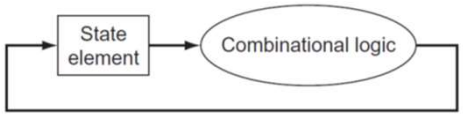
- registers are read,
- ALU output is generated,
- data memory is read and
- branch target addresses are computed.

STEP 2:

- The register file is updated for arithmetic or lw instructions.
- Data memory is written for a sw instruction.
- The PC is updated to point to the next instruction.

In a single-cycle datapath everything in STEP 1 must complete within one clock cycle.





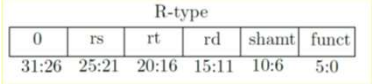
ALU Control

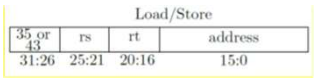
- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

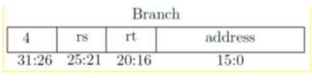
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

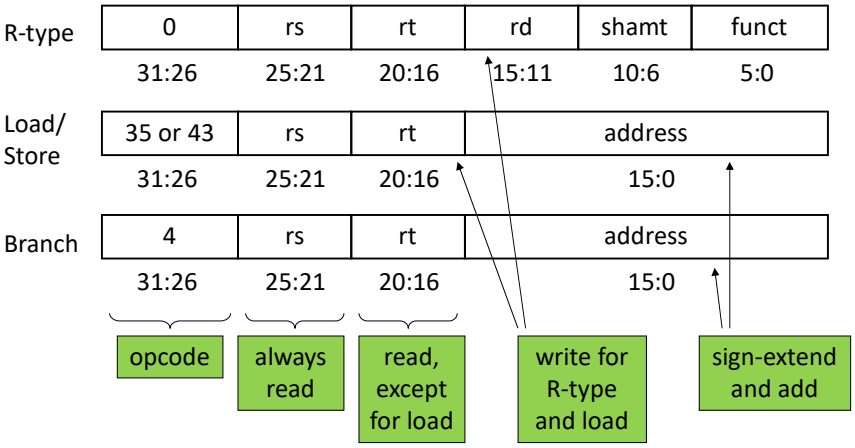




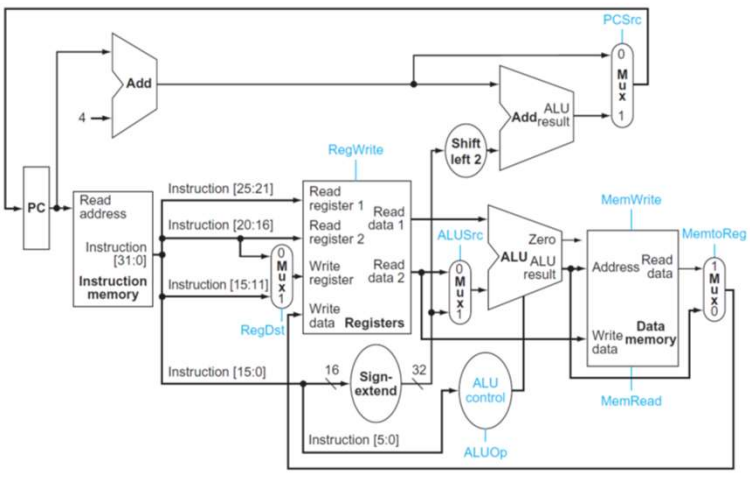


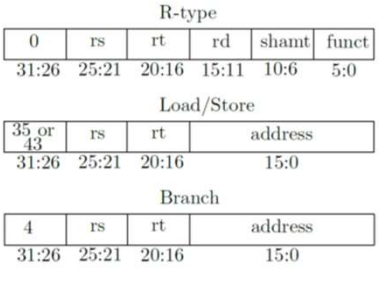
The Main Control Unit

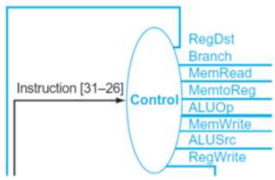
- Control signals derived from instruction



Single Cycle Datapath With Control



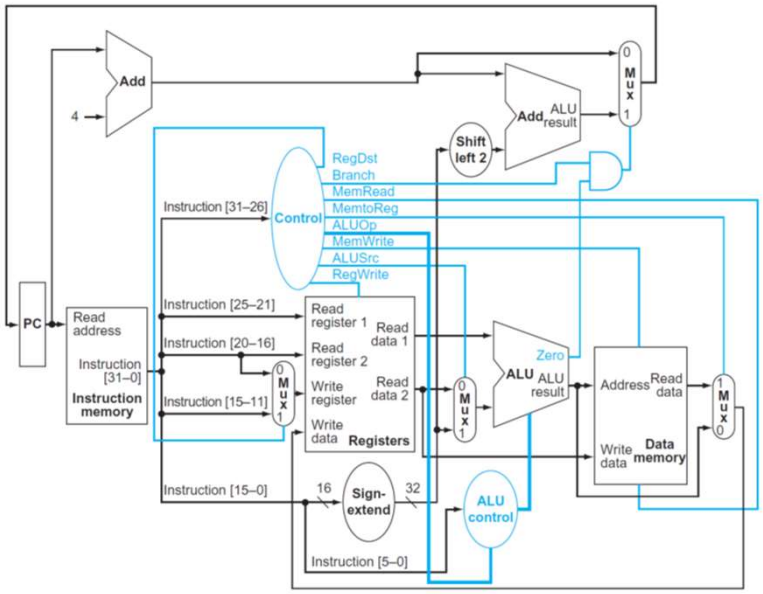




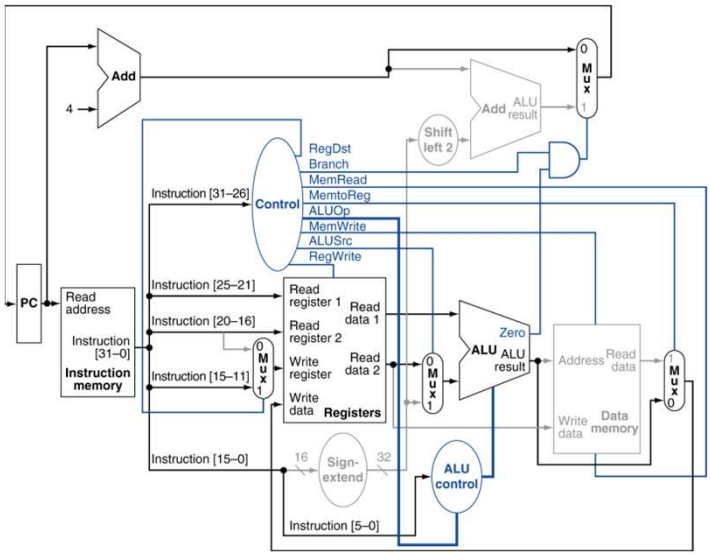
Q. How to augment datapath for conditional jump instruction e.g., beq?

Hint: What should be the control logic for PCSrc?

Single Cycle Datapath With Control

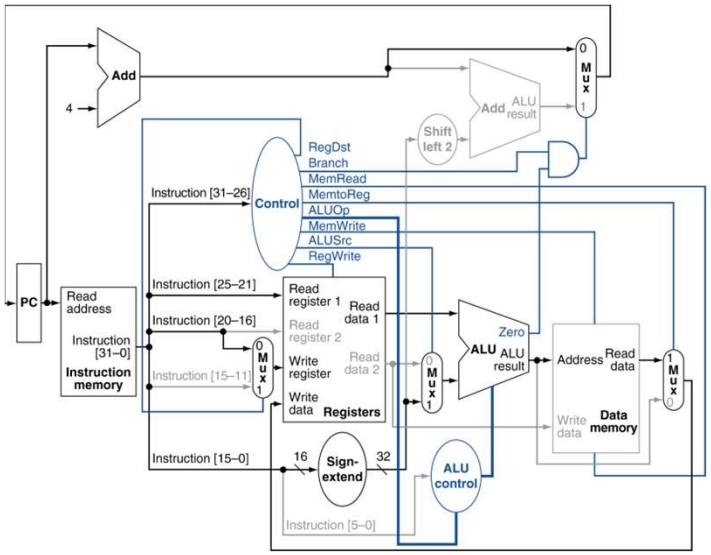


R-Type Instruction



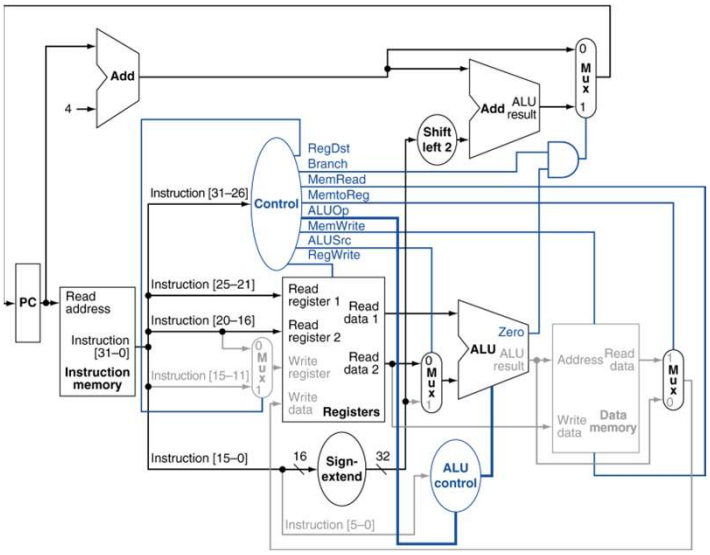
Input or output	Signal name	R-format
Inputs	Op5	0
	Op4	0
	Op3	0
	Op2	0
	Op1	0
	Op0	0
Outputs	RegDst	1
	ALUSrc	0
	MemtoReg	0
	RegWrite	1
	MemRead	0
	MemWrite	0
	Branch	0
	ALUOp1	1
	ALUOp0	0

Load Instruction



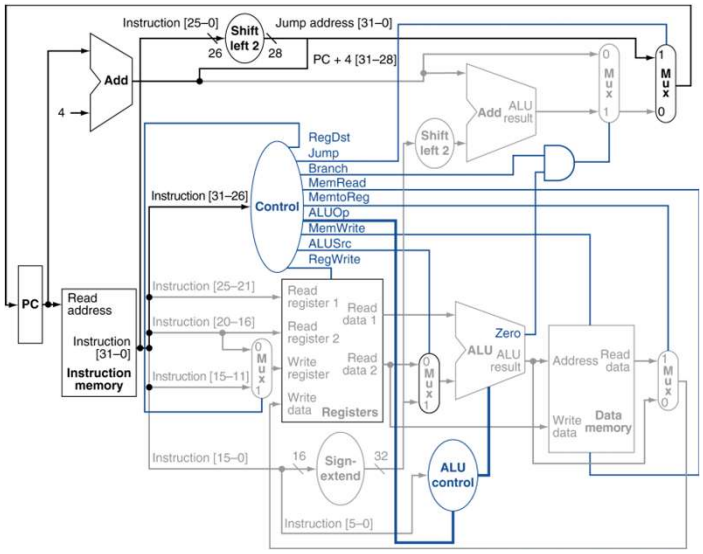
Input or output	Signal name	lw	sw
Inputs	Op5	1	1
	Op4	0	0
	Op3	0	1
	Op2	0	0
	Op1	1	1
	Op0	1	1
Outputs	RegDst	0	X
	ALUSrc	1	1
	MemtoReg	1	X
	RegWrite	1	0
	MemRead	1	0
	MemWrite	0	1
	Branch	0	0
	ALUOp1	0	0
	ALUOp0	0	0

Branch-on-Equal Instruction



Input or output	Signal name	beq
Inputs	Op5	0
	Op4	0
	Op3	0
	Op2	1
	Op1	0
	Op0	0
Outputs	RegDst	X
	ALUSrc	0
	MemtoReg	X
	RegWrite	0
	MemRead	0
	MemWrite	0
	Branch	1
	ALUOp1	0
	ALUOp0	1

Datapath With Unconditional Jumps



2	address
31:26	25:0

Performance Issues

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast

Overview of Multicycle Implementation

Single cycle processor's problems:

- The cycle time has to be long enough for the slowest instruction

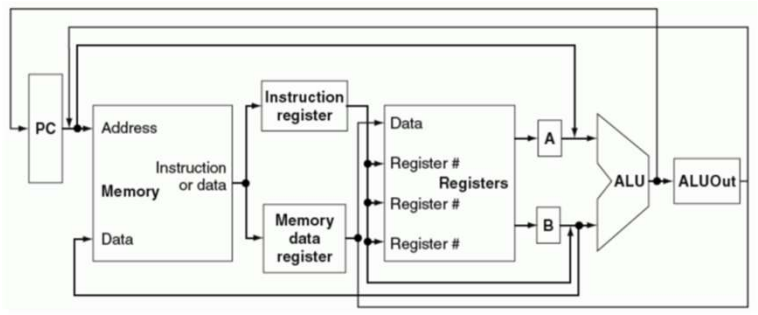
Solution:

- Break the instruction into smaller steps
- Execute each step (instead of the entire instruction) in one cycle
  - ✓ Cycle time: time it takes to execute the longest step
  - ✓ Keep all the steps to have similar length
- This is the essence of the multiple cycle processor

The advantages of the multiple cycle processor:

- Cycle time is much shorter
- Different instructions take different number of cycles to complete
  - ✓ Load takes five cycles
  - ✓ Branch only takes three cycles
- Allows a functional unit to be used more than once per instruction

Multicycle Datapath



High level view of the multicycle datapath

Assumption: In a clock cycle, we can do any one of the following operations

- ✓ A memory access
- ✓ A register file access (two reads or one write)
- ✓ ALU operation

Note: Only IR needs write control signal. All other temporary registers hold data between a pair of adjacent clock cycles.

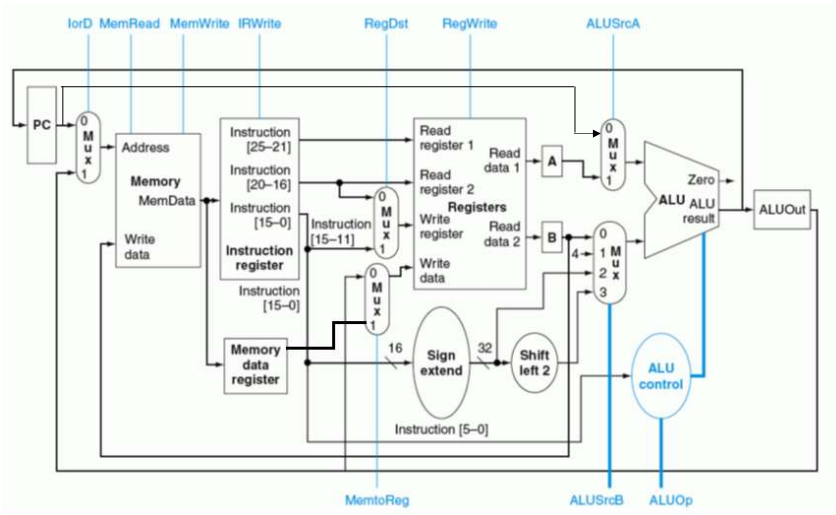
Sharing of functional units on different clock cycles

- ✓ Single memory is used for both data and instructions
- ✓ Single ALU

Temporary registers are required to hold data between clock cycles of the same instruction

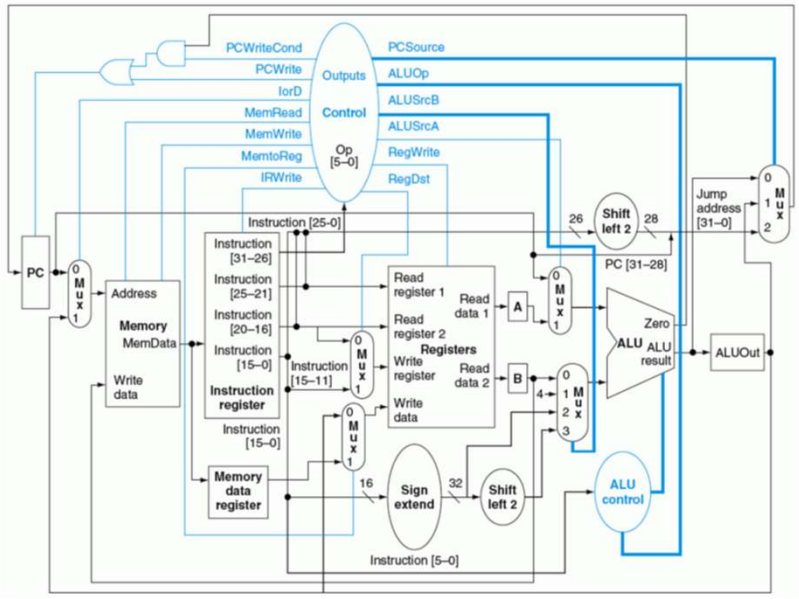
- ✓ Instruction register
- ✓ Memory data register (MDR)
- ✓ A, B, ALUOut

Multicycle Datapath of MIPS



Explain the ALU inputs and control lines

Multicycle Datapath and Control of MIPS

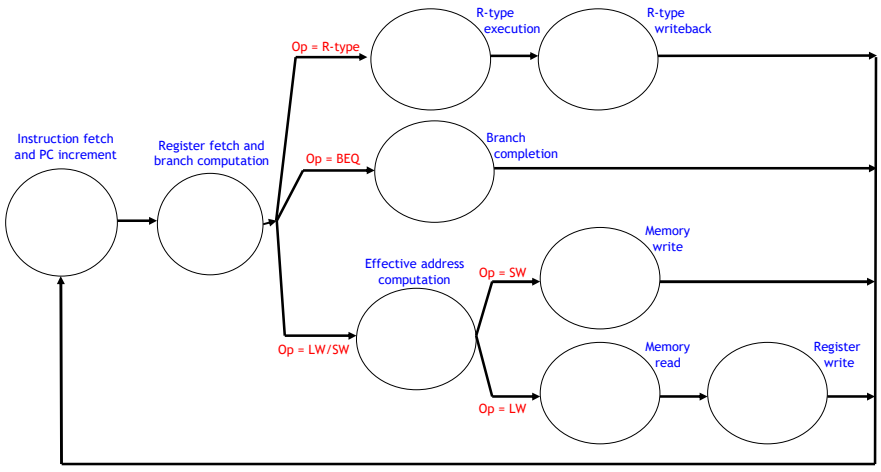


Explain: PCSource, PCWriteCond, PCWrite, IorD

Multicycle control unit

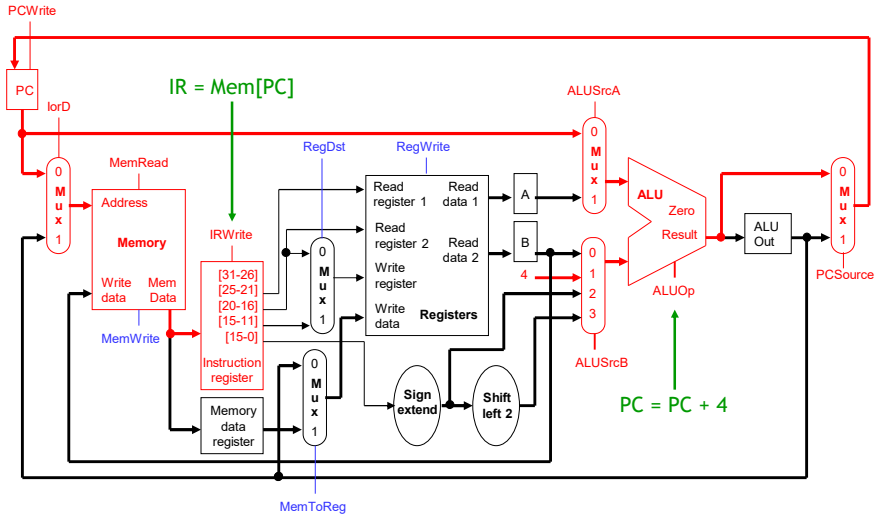
- The control unit is responsible for producing all of the control signals
- Each instruction requires a *sequence* of control signals, generated over multiple clock cycles
  - ✓ This implies that we need a state machine
  - ✓ The datapath control signals will be outputs of the state machine
- Different instructions require different sequences of steps
  - ✓ This implies the instruction word is an input to the state machine
  - ✓ The next state depends upon the exact instruction being executed
- After we finish executing one instruction, we'll have to repeat the entire process again to execute the next instruction

Finite-state machine for the control unit



- Each bubble is a state
  - Holds the control signals for a single cycle
  - Note: All instructions do the same things during the first two cycles

Stage 1: Instruction fetch and PC increment



Stage 1 includes two actions which use two separate functional units: the memory and the ALU.

- Fetch the instruction from memory and store it in IR.
  IR = Mem[PC]
- Use the ALU to increment the PC by 4.
  PC = PC + 4

Stage 1 control signals

- **Instruction fetch: IR = Mem[PC]**

Signal	Value	Description
MemRead	1	Read from memory
IorD	0	Use PC as the memory read address
IRWrite	1	Save memory contents to instruction register

- **Increment the PC: PC = PC + 4**

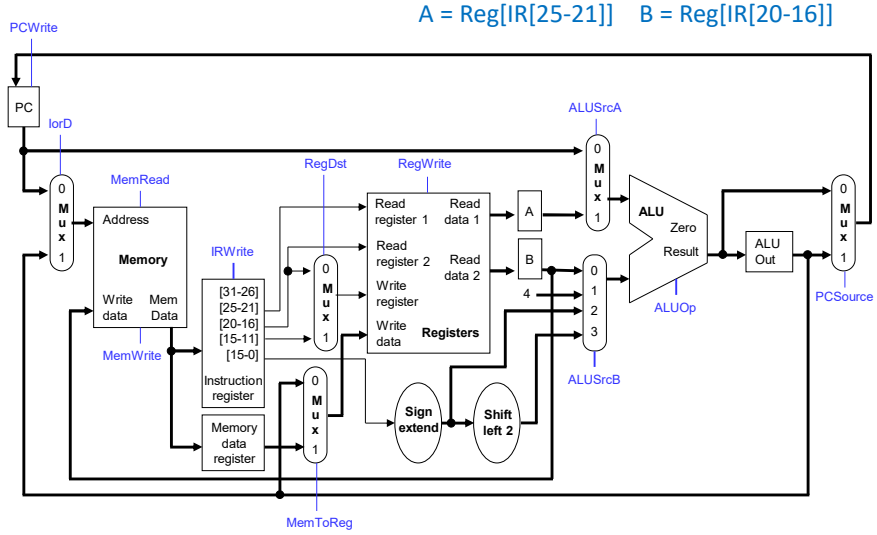
Signal	Value	Description
ALUSrcA	0	Use PC as the first ALU operand
ALUSrcB	01	Use constant 4 as the second ALU operand
ALUOp	ADD	Perform addition
PCWrite	1	Change PC
PCSource	0	Update PC from the ALU output

Note: We'll assume that all control signals not listed are implicitly set to 0

Stage 2: Read registers

Read the contents of source registers rs and rt, and store them in the intermediate registers A and B. (Remember the rs and rt fields come from the instruction register IR.)

A = Reg[IR[25-21]] B = Reg[IR[20-16]]



Control signals:

No control signals need to be set for the register reading operations

- IR[25-21] and IR[20-16] are already applied to the register file.
- Registers A and B are already written on every clock cycle.

Executing Arithmetic Instructions: Stages 3 & 4

- We'll start with R-type instructions like add $t1, $t1, $t2.
- Stage 3 for an arithmetic instruction is simply ALU computation.

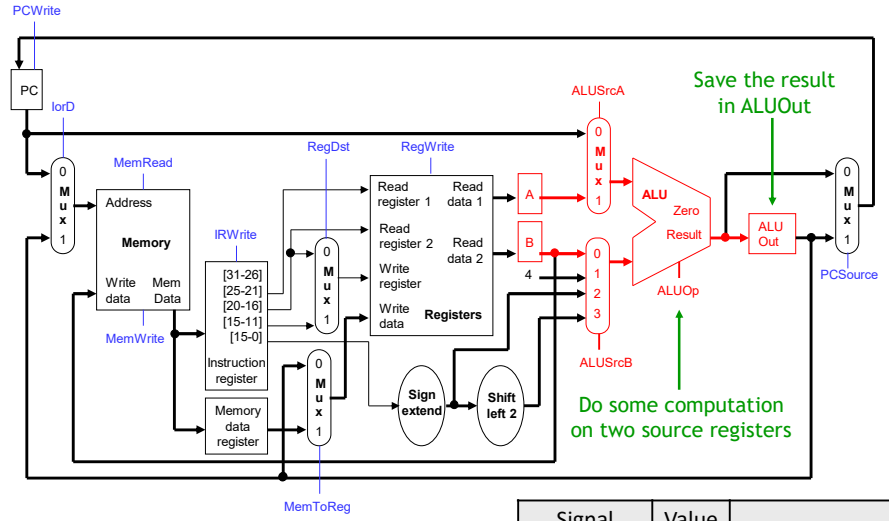
ALUOut = A op B

  - A and B are the intermediate registers holding the source operands.
  - The ALU operation is determined by the instruction's "func" field and could be one of add, sub, and, or, slt.

- Stage 4, the final R-type stage, is to store the ALU result generated in the *previous* cycle into the destination register rd.

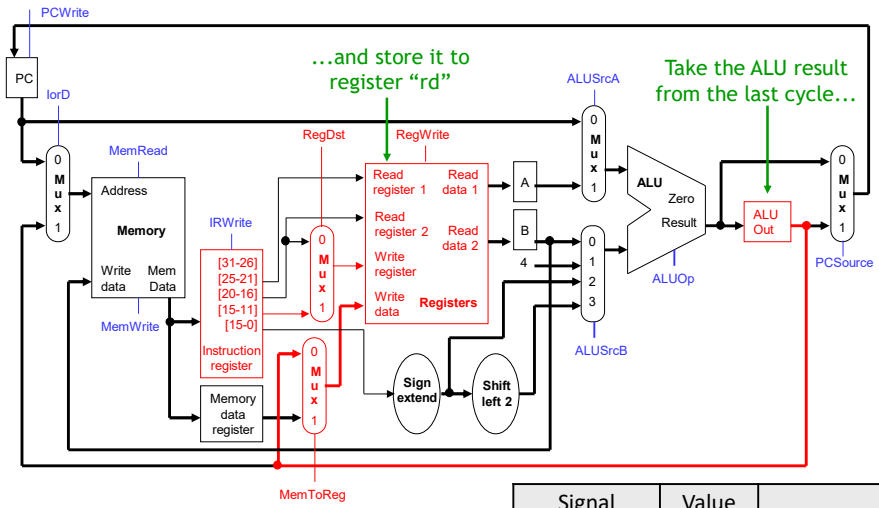
Reg[IR[15-11]] = ALUOut

Stage 3 (R-type): instruction execution



Signal	Value	Description
ALUSrcA	1	Use A as the first ALU operand
ALUSrcB	00	Use B as the second ALU operand
ALUOp	func	Do the operation specified in the “func” field

Stage 4 (R-type): write back



Signal	Value	Description
RegWrite	1	Write to the register file
RegDst	1	Use field rd as the destination register
MemToReg	0	ALUOut contains the data to write

Executing a beq Instruction

- We can execute a branch instruction in three stages or clock cycles.
  - But it requires a little cleverness…

— Stage 1 involves instruction fetch and PC increment.

IR = Mem[PC]
PC = PC + 4

— Stage 2 is register fetch and branch target computation.

A = Reg[IR[25-21]]
B = Reg[IR[20-16]]

- Stage 3 is the final cycle needed for executing a branch instruction.
  - Assuming we have the branch target available

```
if (A == B) then
  PC = branch_target
```

When Should We Compute the Branch Target?

- We need the ALU to do the computation.
  - When is the ALU not busy?

Cycle	ALU
1	PC = PC + 4
2	Here
3	Comparing A & B

- But, we don't know whether or not the branch is taken in cycle 2!!
- That's okay…. we can still go ahead and compute the branch target first.
  - The ALU is otherwise free during this clock cycle.
  - Nothing is harmed by doing the computation early. If the branch is not taken, we can just ignore the ALU result.
- This idea is also used in more advanced CPU design techniques.
  - Modern CPUs perform branch prediction, which will be discussed in the context of pipelining.

Stage 2 Revisited: Compute the Branch Target

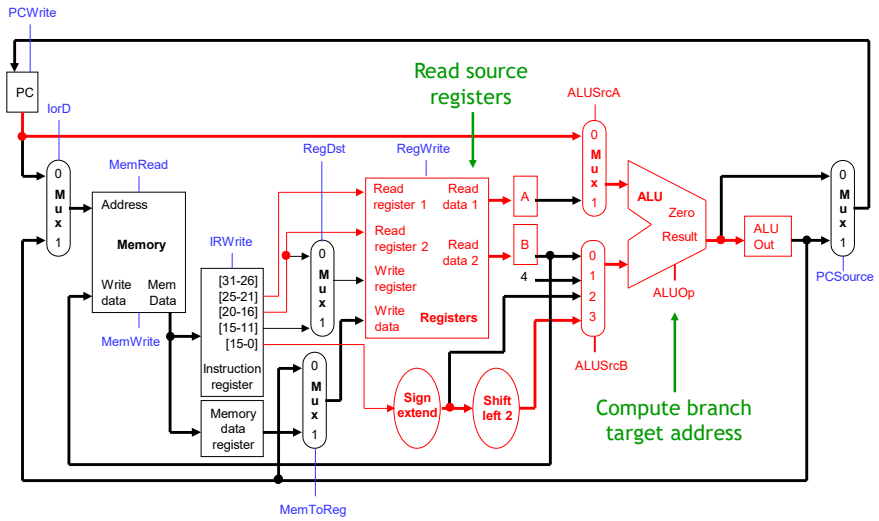
- To Stage 2, we'll add the computation of the branch target.
  - Compute the branch target address by adding the new PC (the original PC + 4) to the sign-extended, shifted constant from IR.

    ALUOut = PC + (sign-extend(IR[15-0]) << 2)

    We save the target address in ALUOut for now, since we don't know yet if the branch should be taken.

  - What about R-type instructions that always go to PC+4

Stage 2(revisited): Register Fetch & Branch Target Computation



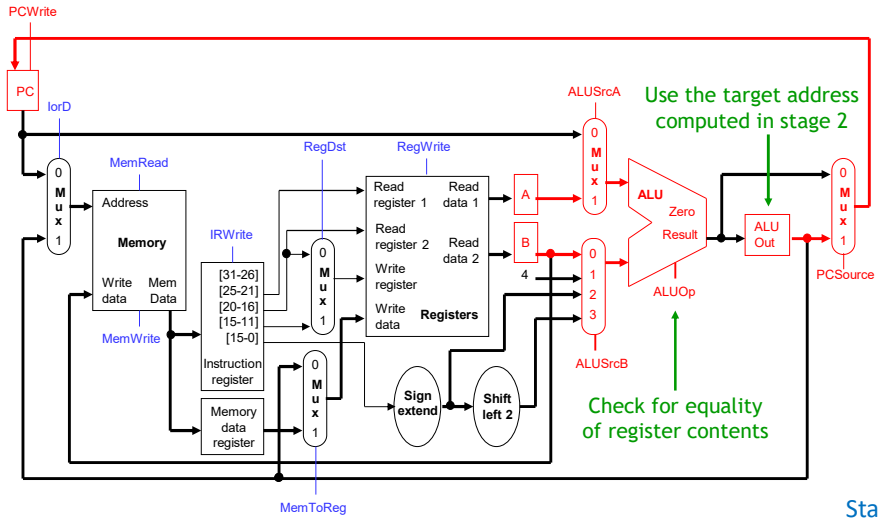
Stage 2 Control Signals

- No control signals need to be set for the register reading operations A = Reg[IR[25-21]] and B = Reg[IR[20-16]].
  - IR[25-21] and IR[20-16] are already applied to the register file.
  - Registers A and B are already written on every clock cycle.
- Branch target computation: ALUOut = PC + (sign-extend(IR[15-0]) << 2)

Signal	Value	Description
ALUSrcA	0	Use PC as the first ALU operand
ALUSrcB	11	Use (sign-extend(IR[15-0]) << 2) as second operand
ALUOp	ADD	Add and save the result in ALUOut

Note: ALUOut is also written automatically on each clock cycle.

Stage 3 (BEQ): Branch Completion



Stage 3 is the final cycle needed for executing a branch instruction.

if (A == B) then
PC = ALUOut

Note: Remember that A and B are compared by subtracting and testing for a result of 0, so we must use the ALU again in this stage.

Stage 3 (beq) Control Signals

- Comparison: if (A == B) ...

Signal	Value	Description
ALUSrcA	1	Use A as the first ALU operand
ALUSrcB	00	Use B as the second ALU operand
ALUOp	SUB	Subtract, so Zero will be set if A = B

- Branch: ...then PC = ALUOut

Signal	Value	Description
PCWrite	Zero	Change PC only if Zero is true (i.e., A = B)
PCSource	1	Update PC from the ALUOut register

- ALUOut contains the ALU result from the *previous* cycle, which would be the branch target. We can write that to the PC, even though the ALU is doing something different (comparing A and B) during the *current* cycle.

Executing a sw Instruction

- A store instruction, like sw $a0, 16($sp), also shares the same first two stages as the other instructions.
  - Stage 1: instruction fetch and PC increment.
  - Stage 2: register fetch and branch target computation.

- Stage 3 computes the effective memory address using the ALU.

  ALUOut = A + sign-extend(IR[15-0])

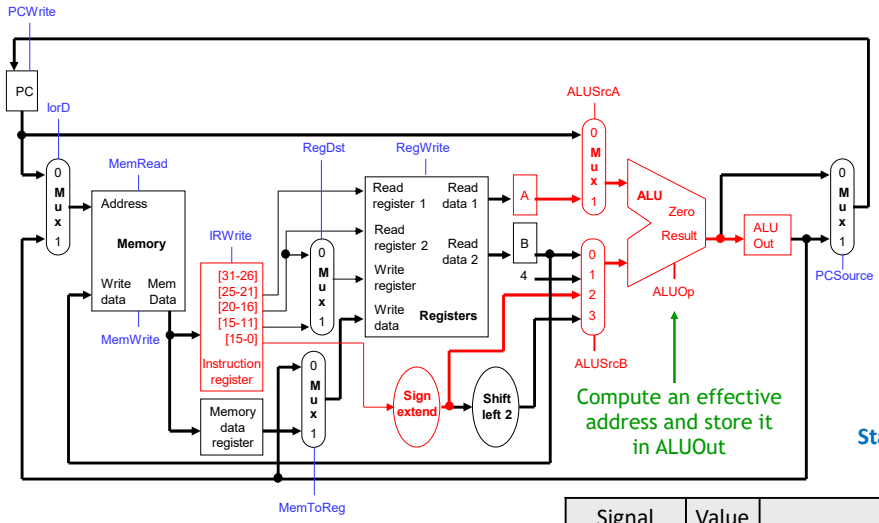
  A contains the base register (like $sp), and IR[15-0] is the 16-bit constant offset from the instruction word, which is *not* shifted.

- Stage 4 saves the register contents (here, $a0) into memory.

  Mem[ALUOut] = B

  Remember that the second source register rt was already read in Stage 2 (and again in Stage 3), and its contents are in intermediate register B.

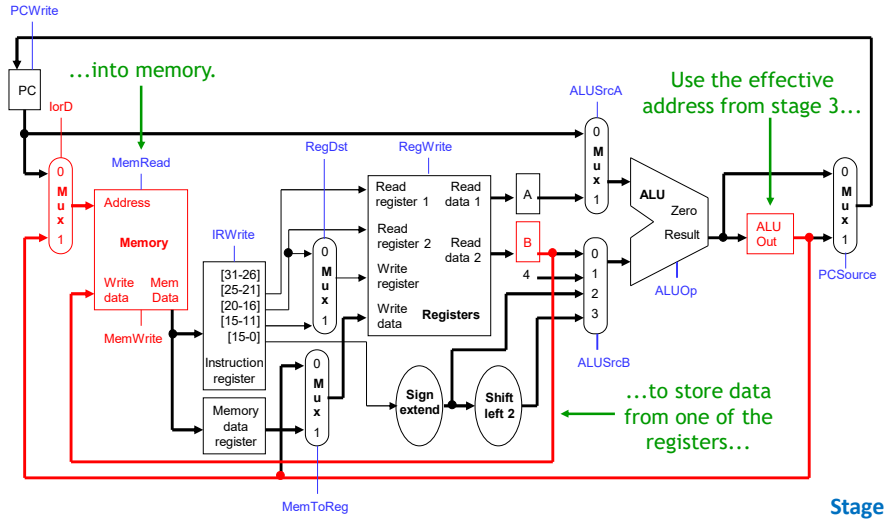
Stage 3 (sw): Effective Address Computation



Stage 3 (address computation):
ALUOut = A + sign-extend(IR[15-0])

Signal	Value	Description
ALUSrcA	1	Use A as the first ALU operand
ALUSrcB	10	Use sign-extend(IR[15-0]) as the second operand
ALUOp	010	Add and store the resulting address in ALUOut

Stage 4 (sw): Memory Write



Stage 4 (memory write): Mem[ALUOut] = B

Note: The memory's "Write data" input *always* comes from the B intermediate register, so no selection is needed.

Signal	Value	Description
MemWrite	1	Write to the memory
IorD	1	Use ALUOut as the memory address

Executing a lw Instruction

- Finally, lw is the most complex instruction, requiring five stages.
- The first two are like all the other instructions.
  - Stage 1: instruction fetch and PC increment.
  - Stage 2: register fetch and branch target computation.
- The third stage is the same as for sw, since we have to compute an effective memory address in both cases.
  - Stage 3: compute the effective memory address.
- Stage 4 is to read from the effective memory address, and to store the value in the intermediate register MDR (memory data register).

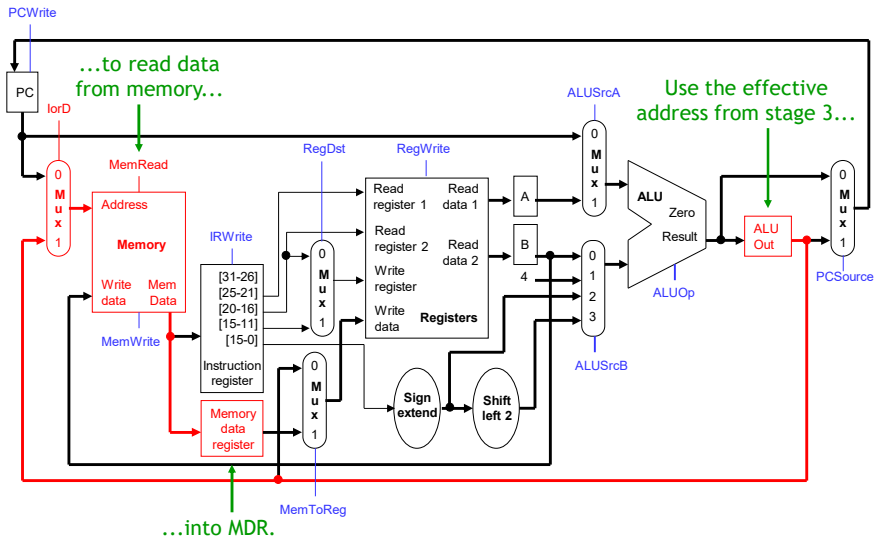
MDR = Mem[ALUOut]

- Stage 5 stores the contents of MDR into the destination register.

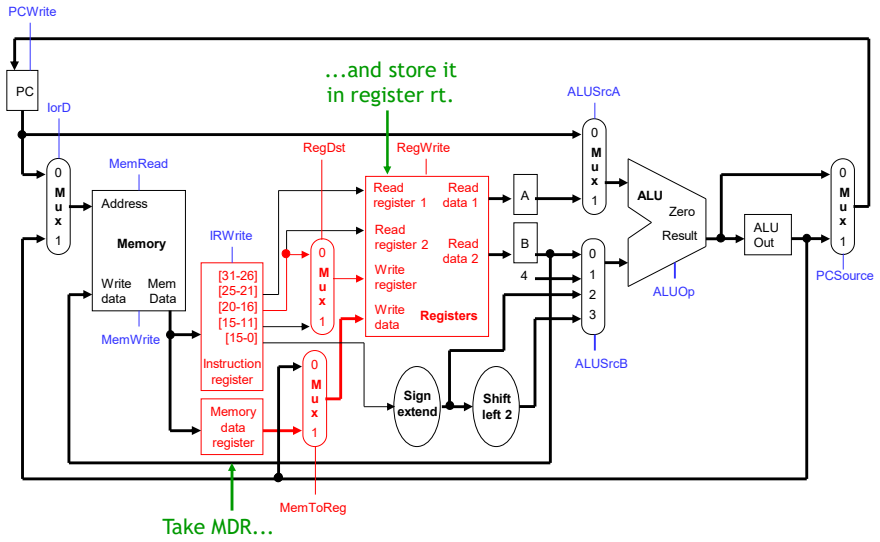
Reg[IR[20-16]] = MDR

Remember that the destination register for lw is field rt (bits 20-16) and *not* field rd (bits 15-11).

Stage 4 (lw): Memory Read



Stage 5 (lw): register write



Stages 4-5 (lw) control signals

- Stage 4 (memory read): MDR = Mem[ALUOut]

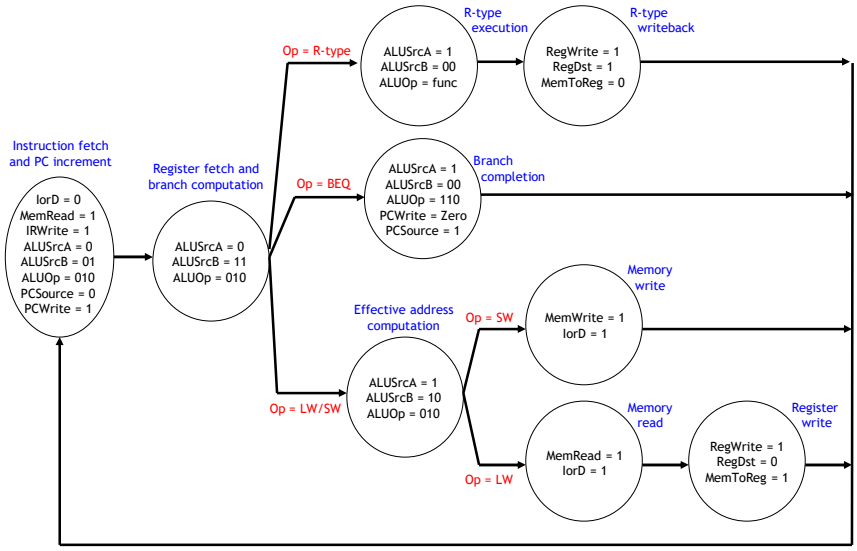
Signal	Value	Description
MemRead	1	Read from memory
IorD	1	Use ALUOut as the memory address

The memory contents will be automatically written to MDR.

- Stage 5 (writeback): Reg[IR[20-16]] = MDR

Signal	Value	Description
RegWrite	1	Store new data in the register file
RegDst	0	Use field rt as the destination register
MemToReg	1	Write data from MDR (from memory)

Finite-state Machine for the Control Unit



Implementing the FSM

- This can be translated into a state table; here are the first two states.

Current State	Input (Op)	Next State	Output (Control signals)											
			PC Write	IorD	MemRead	Mem Write	IR Write	Reg Dst	MemToReg	Reg Write	ALU SrcA	ALU SrcB	ALU Op	PC Source
Instr Fetch	X	Reg Fetch	1	0	1	0	1	X	X	0	0	01	010	0
Reg Fetch	BEQ	Branch completion	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	R-type	R-type execute	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	LW/SW	Compute eff addr	0	X	0	0	0	X	X	0	0	11	010	X

- You can implement this the hard way.
  - Represent the current state using flip-flops or a register.
  - Find equations for the next state and (control signal) outputs in terms of the current state and input (instruction word).
- Or you can use the easy way.
  - Stick the whole state table into a memory, like a ROM.
  - This would be much easier, since you don't have to derive equations.

Summary

- Now you know how to build a multicycle controller!
  - Each instruction takes several cycles to execute.
  - Different instructions require different control signals and a different number of cycles.
  - We have to provide the control signals in the right sequence.