CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #12

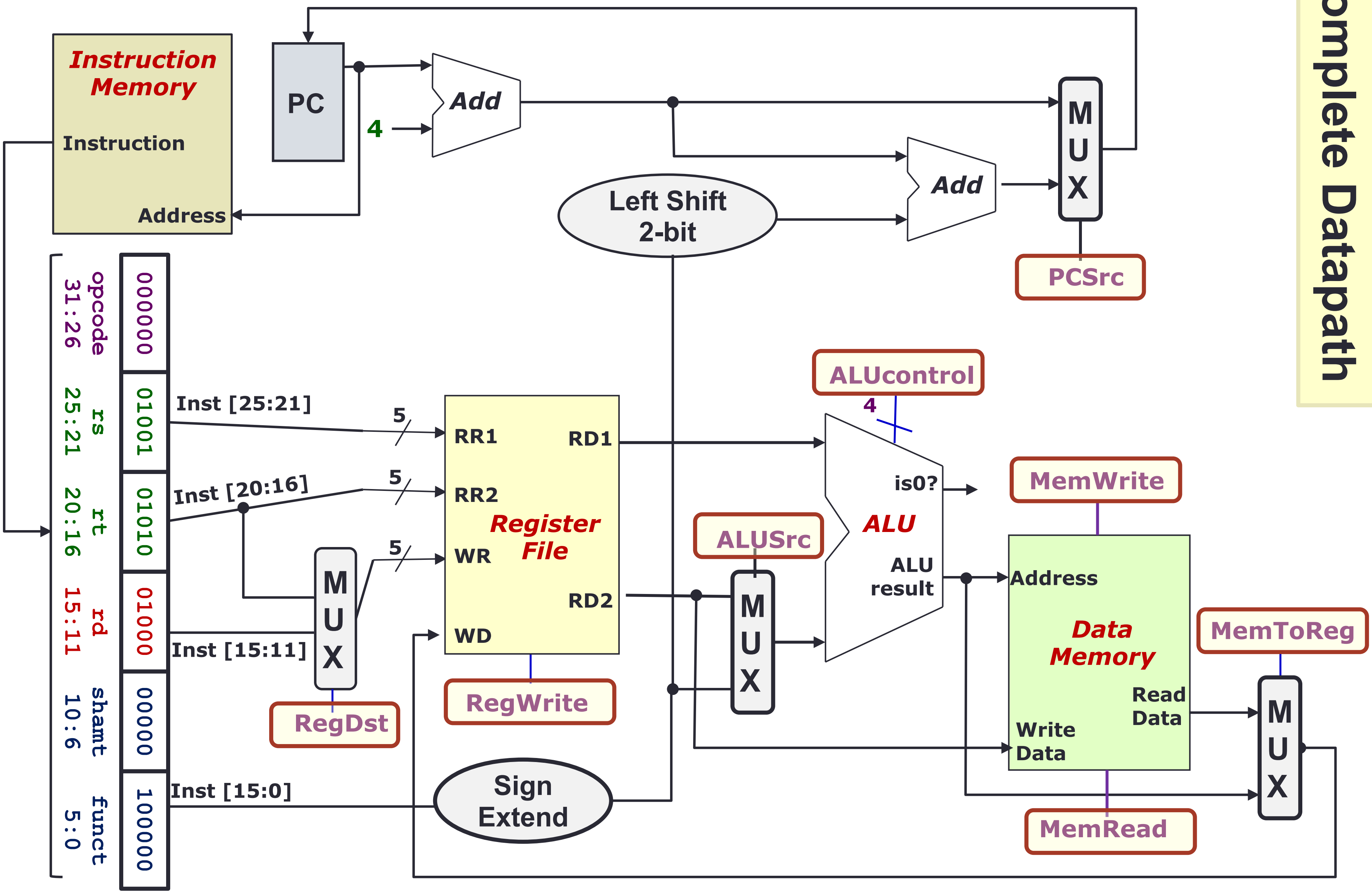
The Processor: Control

NUS National University of Singapore | School of Computing

Lecture #12: Processor: Control

1. Identified Control Signals
2. Generating Control Signals: Idea
3. The Control Unit
4. Control Signals
5. ALU Control Signal
6. Instruction Execution

Complete Datapath



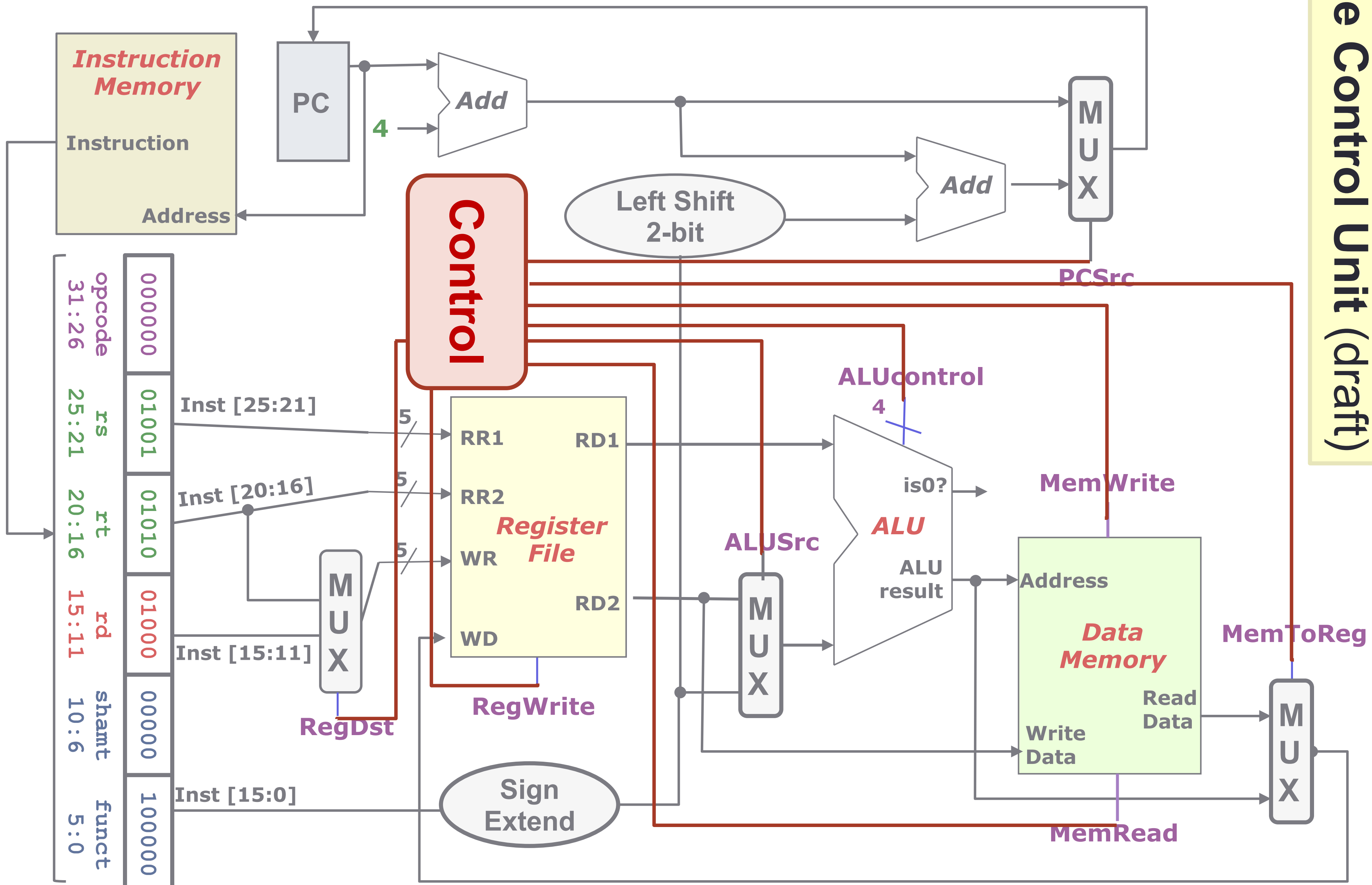
1. Identified Control Signals

Control Signal	Execution Stage	Purpose
`RegDst`	Decode/Operand Fetch	Select the destination register number
`RegWrite`	Decode/Operand Fetch RegWrite	Enable writing of register
`ALUSrc`	ALU	Select the 2nd operand for ALU
`ALUControl`	ALU	Select the operation to be performed
`MemRead` / **`MemWrite`**	Memory	Enable reading/writing of data memory
`MemToReg`	RegWrite	Select the result to be written back to register file
`PCSrc`	Memory/RegWrite	Select the next PC value

2. Generating Control Signals: Idea

- The control signals are generated based on the instruction to be executed:
  - Opcode ➔ Instruction Format
  - Example:
    - R-Format instruction ➔ `RegDst` = 1 (use `Inst[15:11]`))
  - R-Type instruction has additional information:
    - The 6-bit "`funct`" (function code, `Inst[5:0]`) field
- **Idea:**
  - Design a combinational circuit to generate these signals based on Opcode and possibly Function code
    - A **control unit** is needed (a draft design is shown next)

The Control Unit (draft)



3. Let's Implement the Control Unit!

- Approach:
  - Take note of the instruction subset to be implemented:
    - Opcode and Function Code (if applicable)
  - Go through each signal:
    - Observe how the signal is generated based on the instruction opcode and/or function code
  - Construct truth table
  - Design the control unit using logic gates

3. MIPS Instruction Subset (Review)

R-type

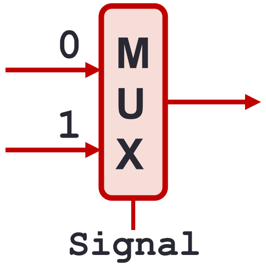
	opcode 31	25	20	15	shamt 10	funct 5
add	0_{16}	rs	rt	rd	0	20_{16}
sub	0_{16}	rs	rt	rd	0	22_{16}
and	0_{16}	rs	rt	rd	0	24_{16}
or	0_{16}	rs	rt	rd	0	25_{16}
slt	0_{16}	rs	rt	rd	0	$2A_{16}$

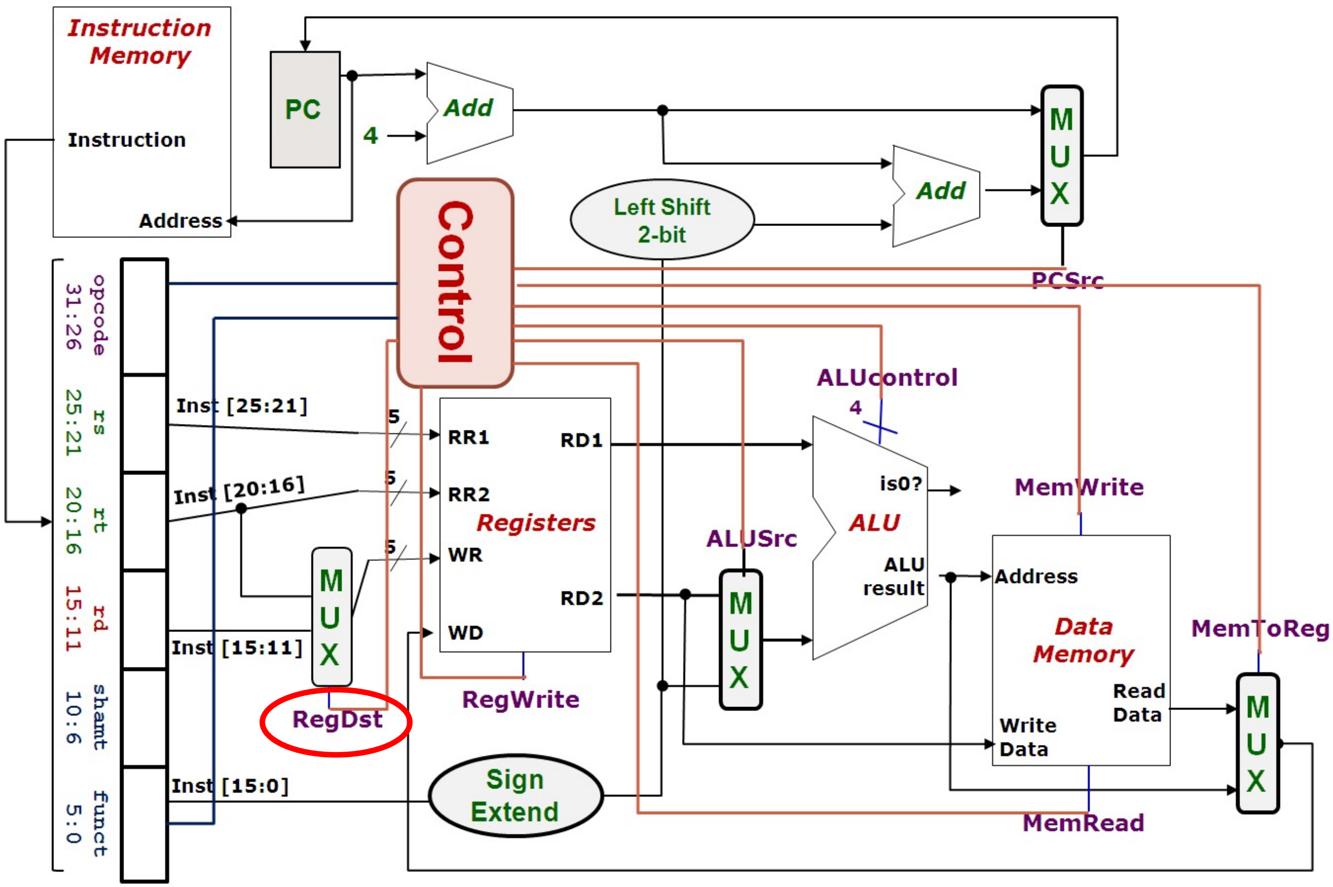
I-type

	31	25	20	15
lw	23_{16}	rs	rd	offset
sw	$2B_{16}$	rs	rd	offset
beq	4_{16}	rs	rd	offset

4. Control Signal: RegDst

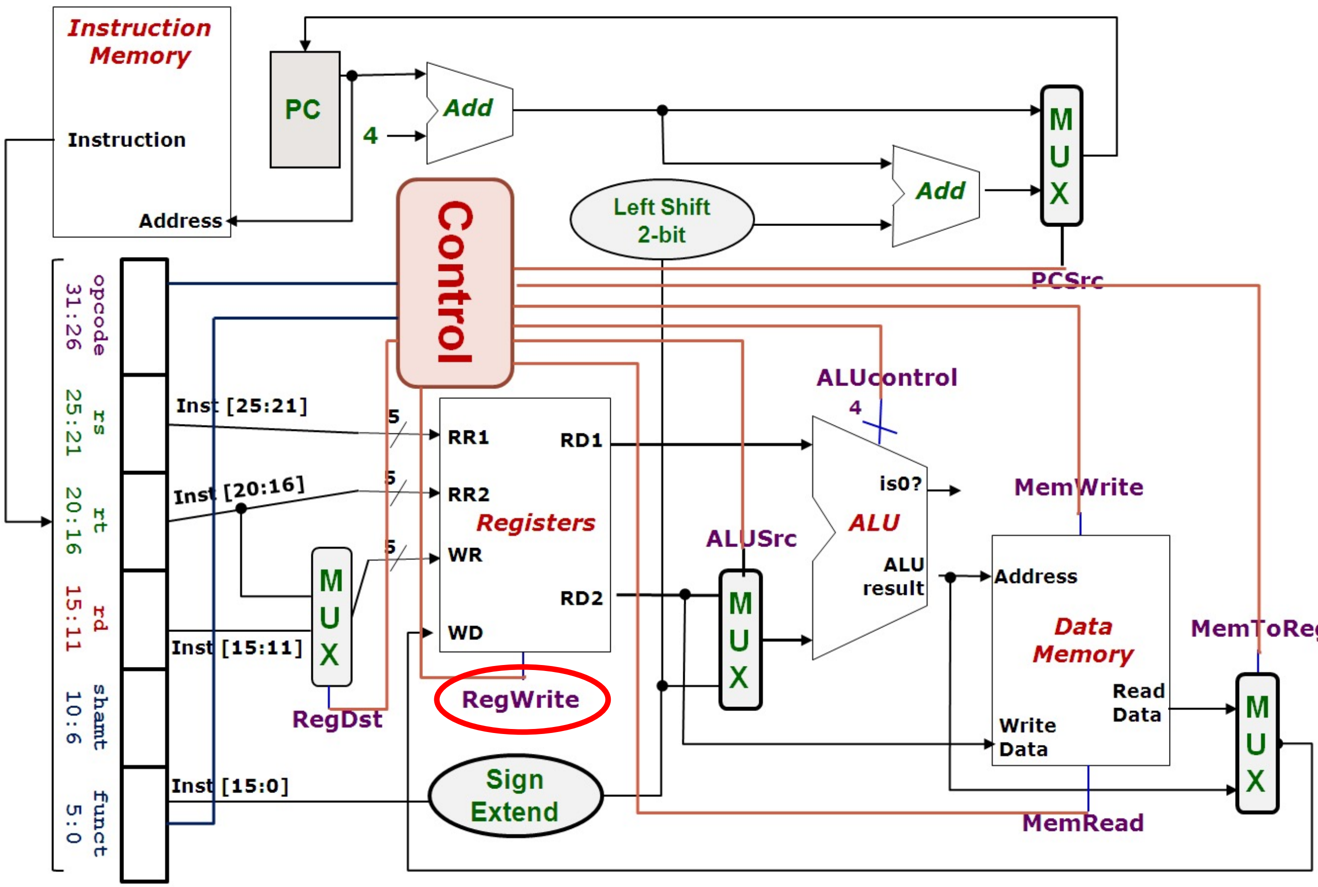
- **False (0)**: Write register = Inst[20:16]
- **True (1)**: Write register = Inst[15:11]





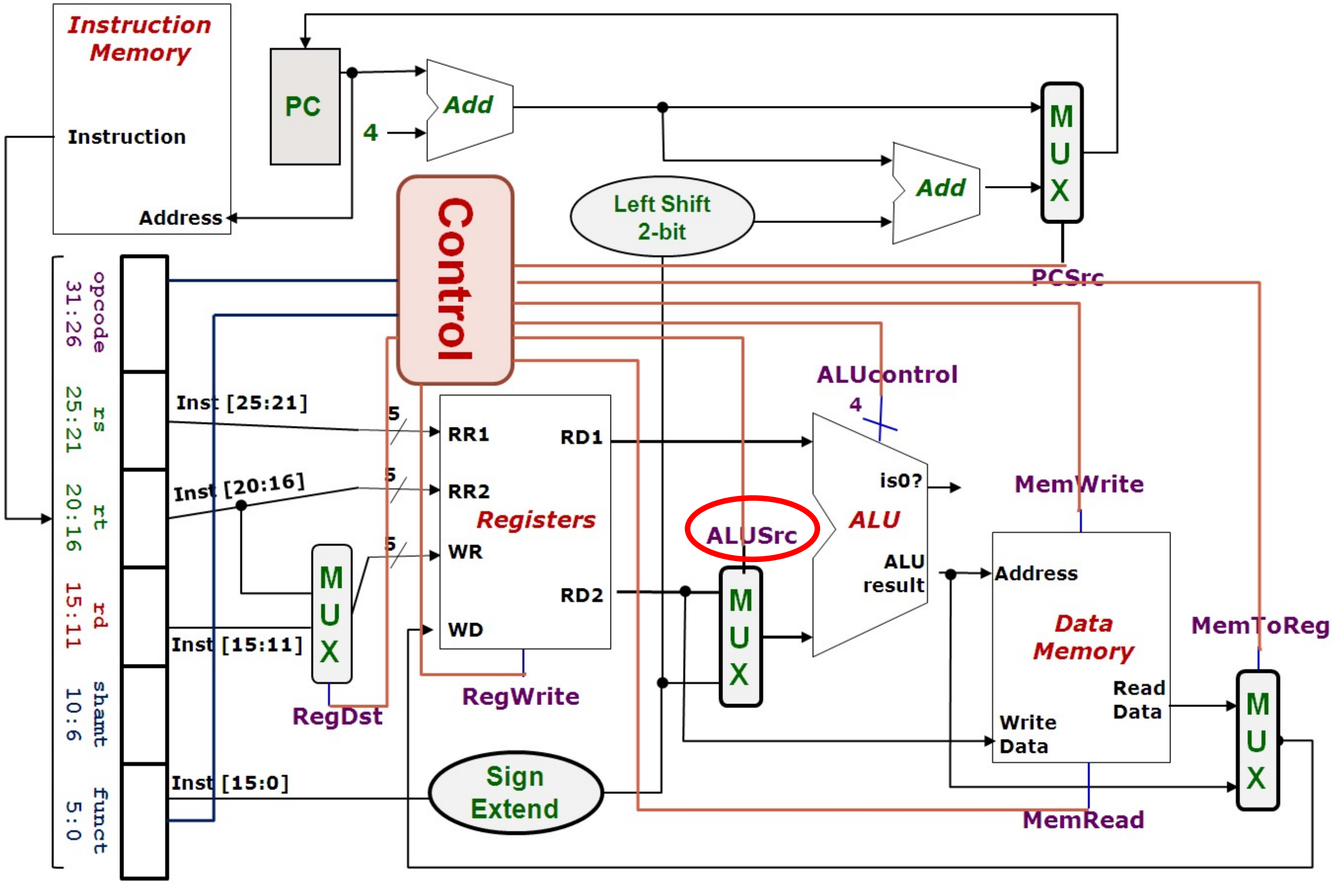
4. Control Signal: `RegWrite`

- **False (0):** No register write
- **True (1):** New value will be written



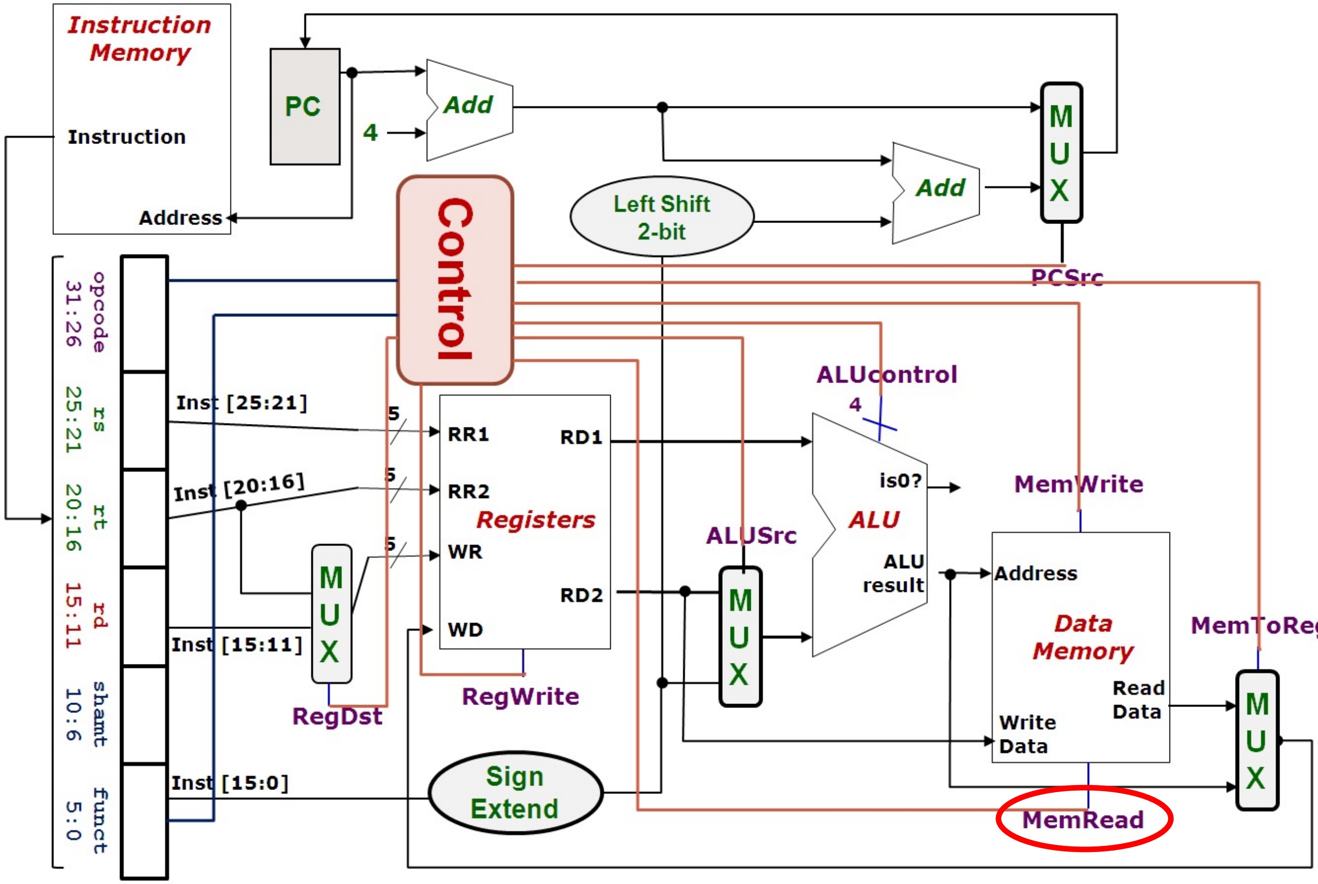
4. Control Signal: ALUSrc

- **False (0):** Operand2 = Register Read Data 2
- **True (1):** Operand2 = SignExt(Inst[15:0])



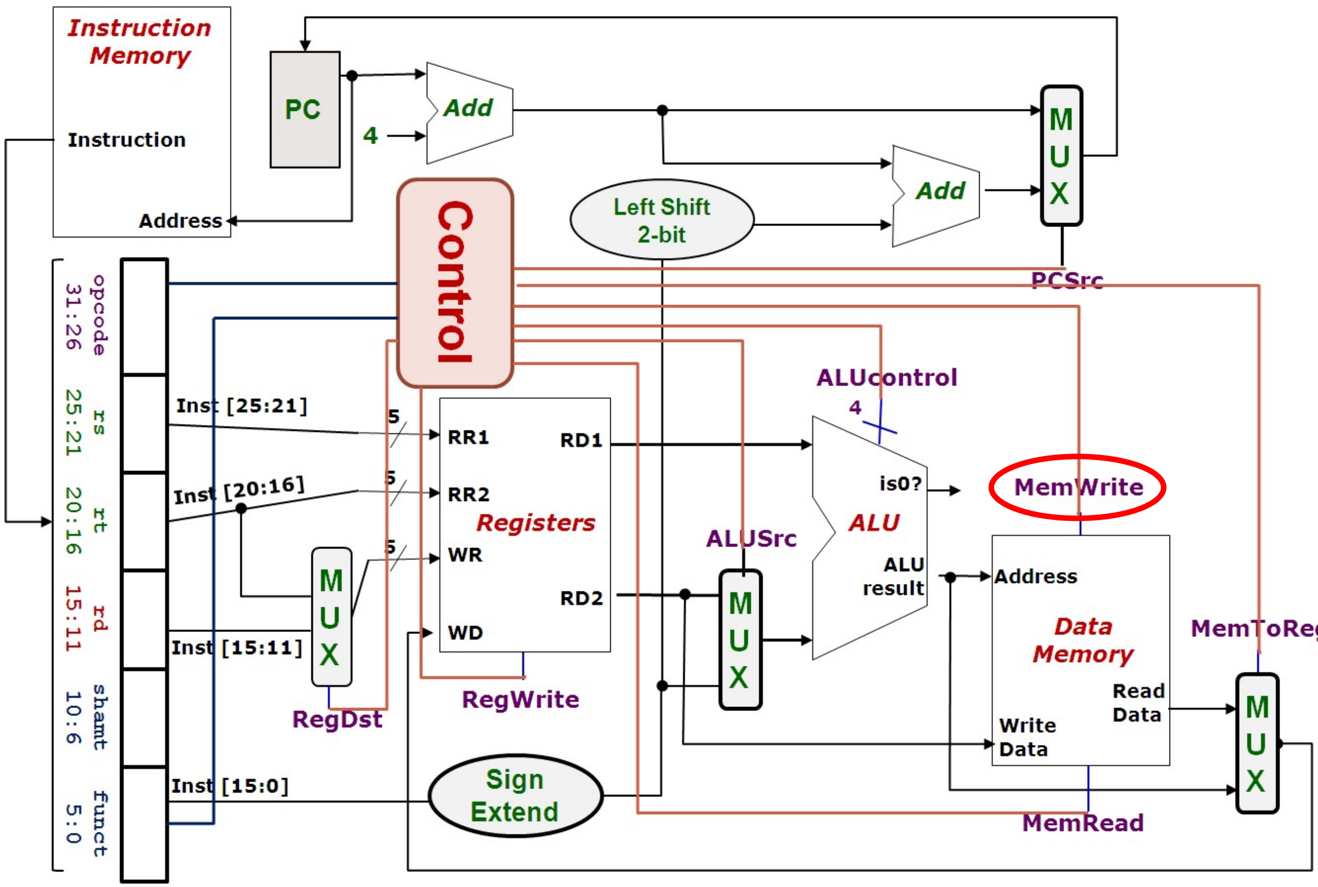
4. Control Signal: MemRead

- **False (0)**: Not performing memory read access
- **True (1)**: Read memory using ***Address***



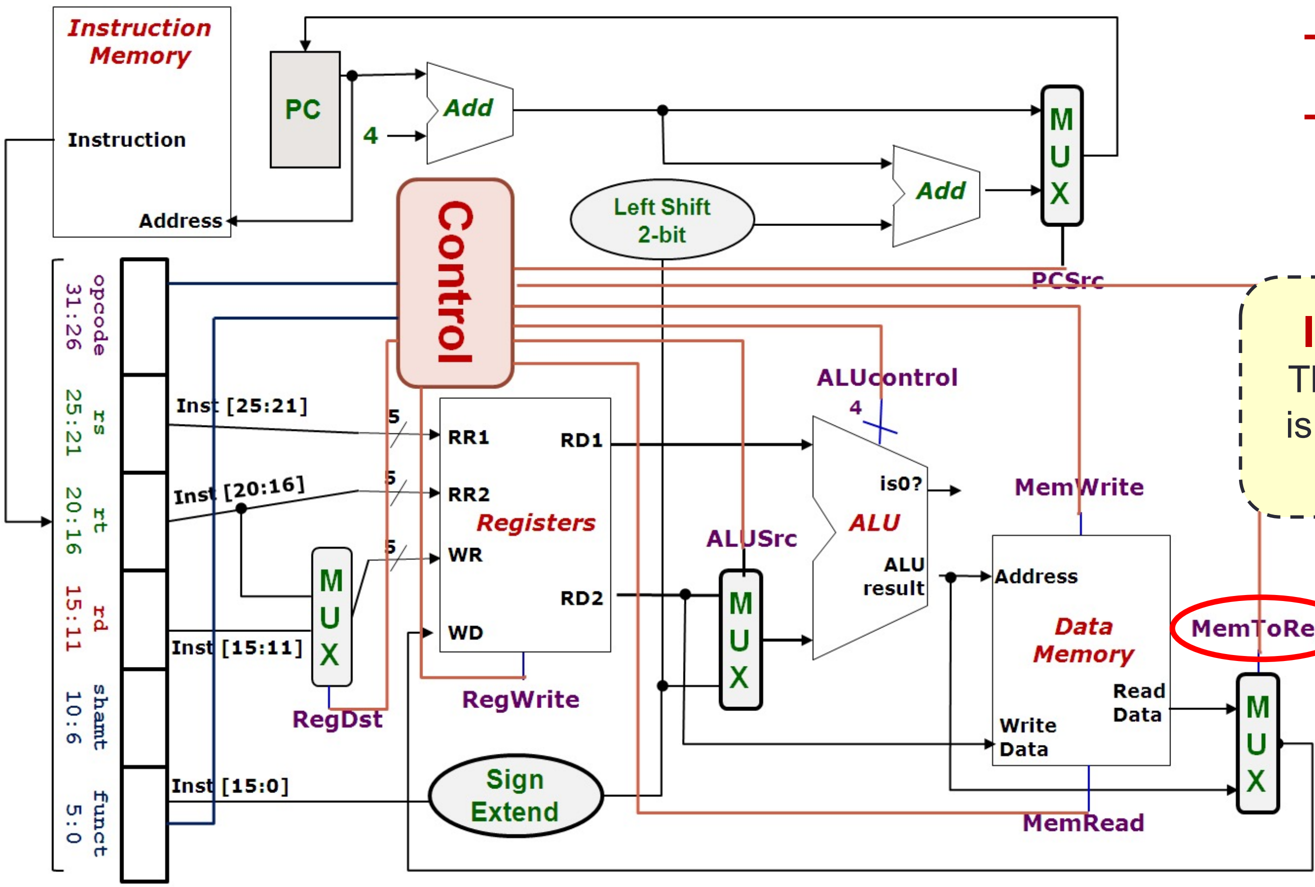
4. Control Signal: MemWrite

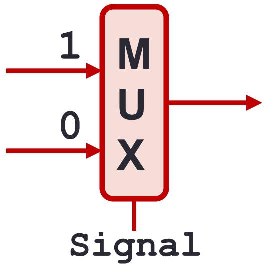
- **False (0):** Not performing memory write operation
- **True (1):** memory[*Address*] ← Register Read Data 2



4. Control Signal: MemToReg

- **True (1):** Register write data = Memory read data
- **False (0):** Register write data = ALU result

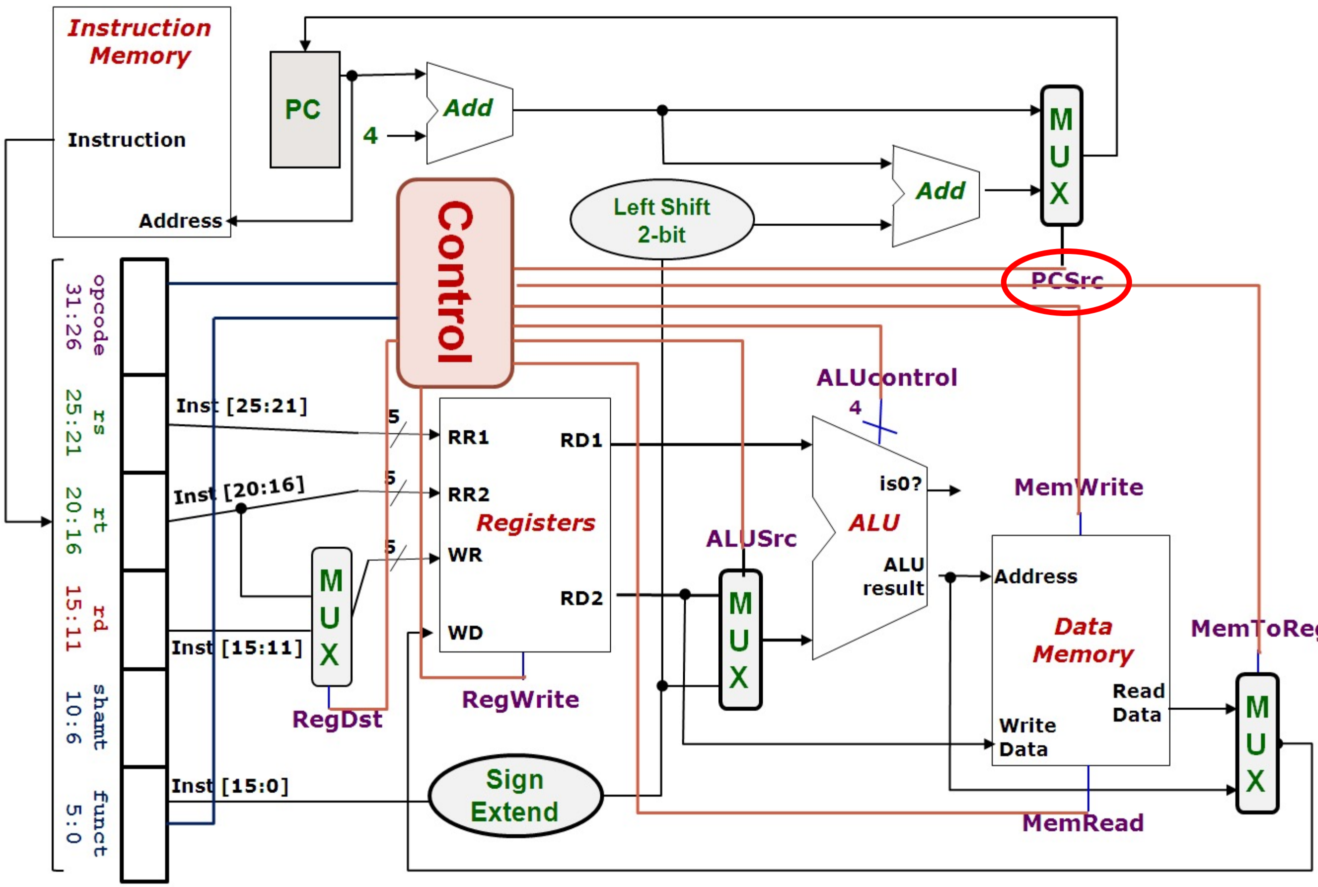




IMPORTANT: The input of MUX is swapped in this case

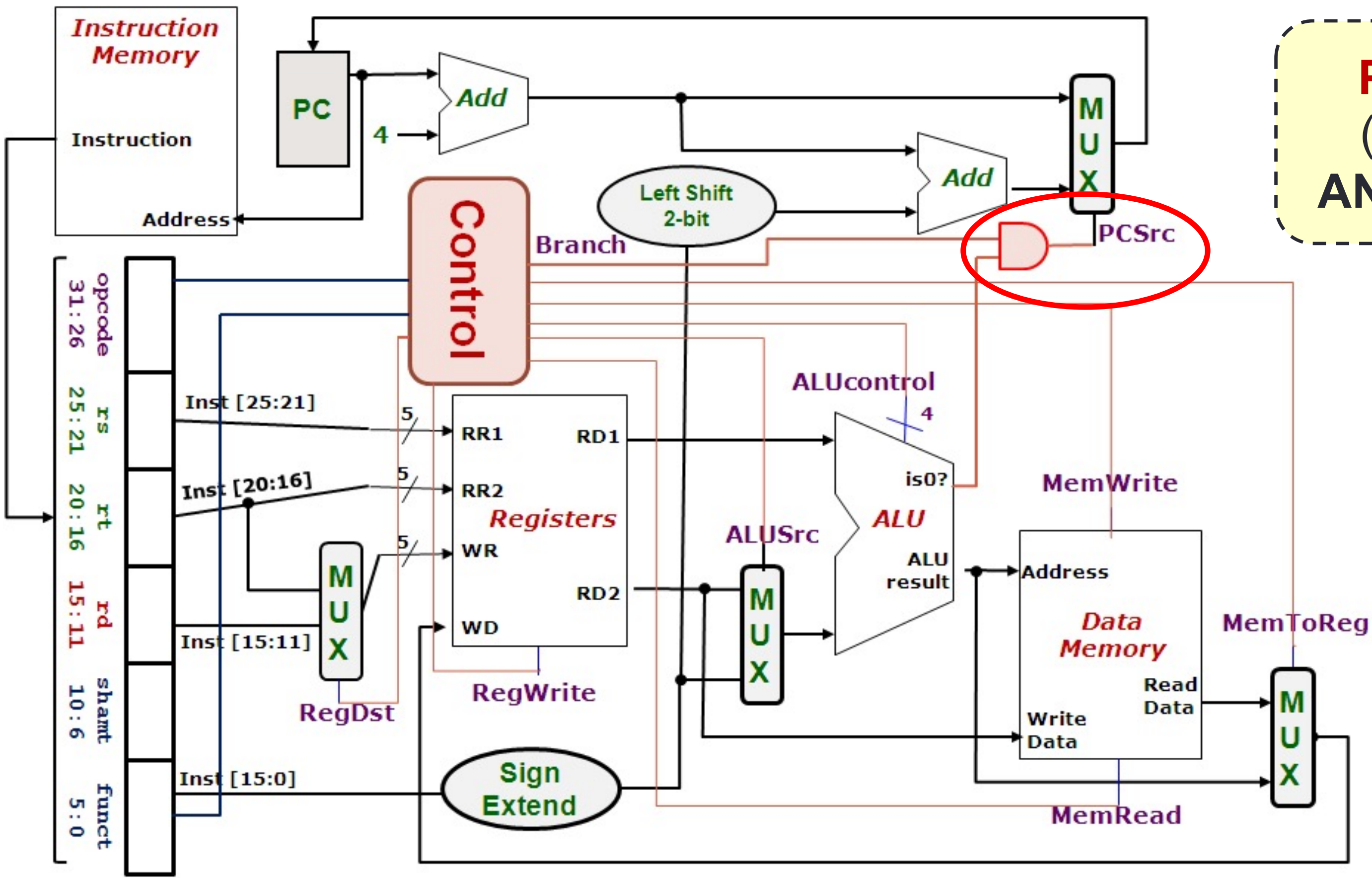
4. Control Signal: `PCSrc` (1/2)

- The "**`isZero?`**" signal from the ALU gives us the actual branch outcome (taken/not taken)
- **Idea:** "If instruction is a branch **AND** taken, then…"



4. Control Signal: PCSrc (2/2)

- **False (0)**: Next PC = PC + 4
- **True (1)**: Next PC = SignExt(`Inst[15:0]`) << 2 + (PC + 4)



PCSrc = (Branch AND isZero)

4. Midpoint Check

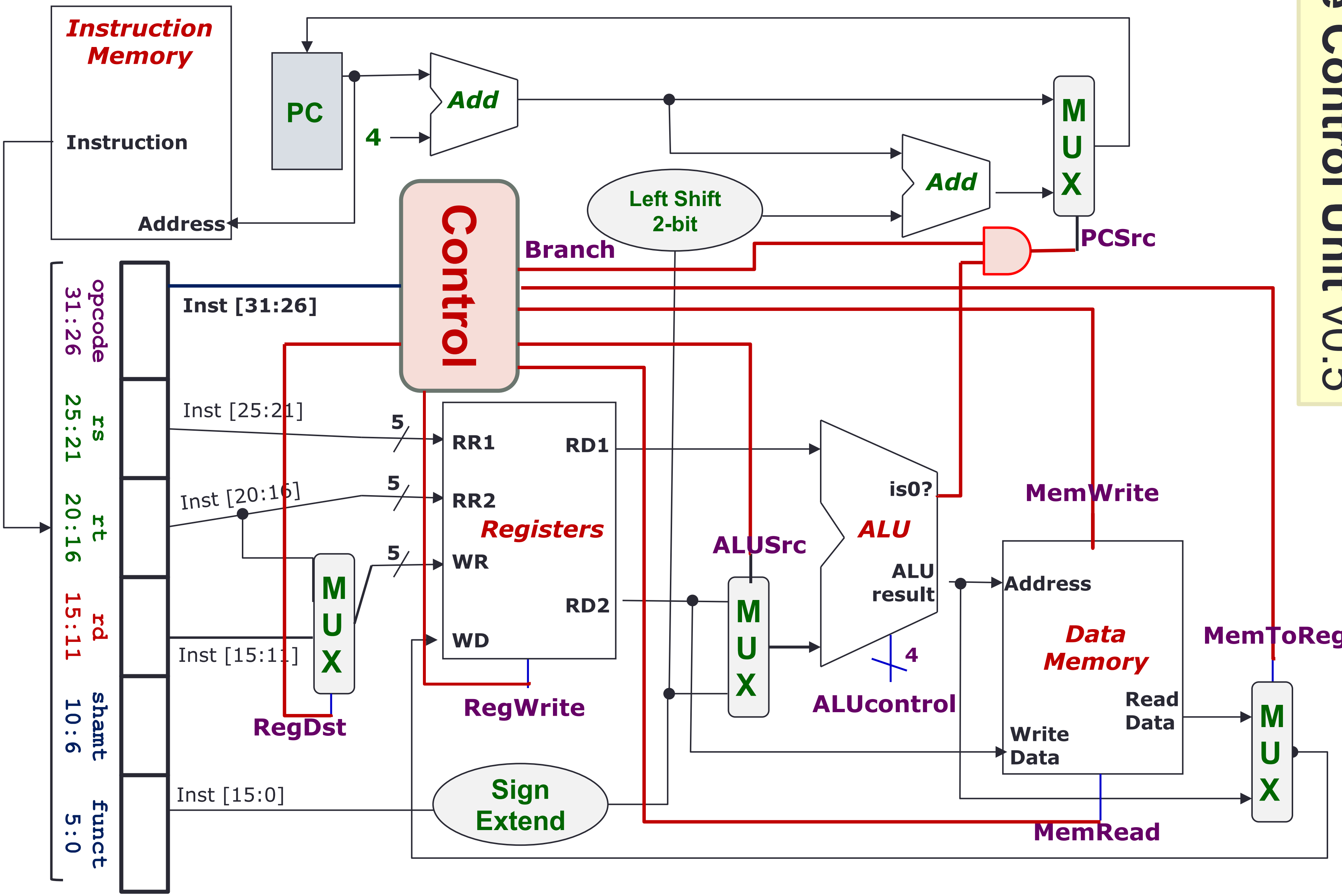
- We have gone through almost all of the signals:
  - Left with the more challenging **ALUControl** signal

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / **MemWrite**	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

- Observation so far:
  - The signals discussed so far can be generated by *opcode* directly
    - Function code is not needed up to this point

→ A major part of the controller can be built based on *opcode* alone

The Control Unit v0.5



5. Closer Look at ALU

Note: We will cover combinational circuits after the recess.

- The ALU is a combinational circuit:
  - Capable of performing several arithmetic operations
- In Lecture #11:
  - We noted the required operations for the MIPS subset

- Question:
  - How is the `ALUcontrol` signal designed?

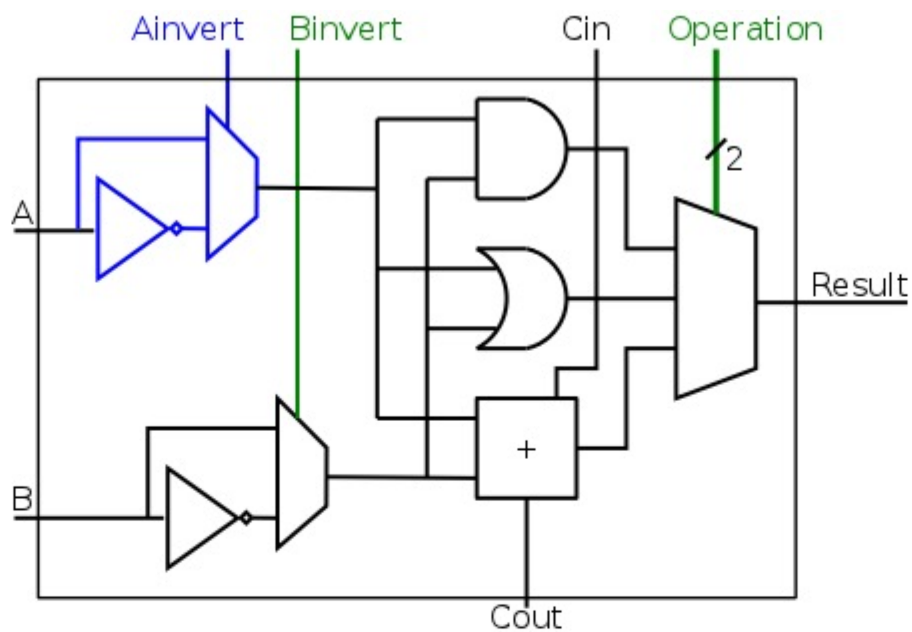
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

5. One Bit At A Time

Note: We will revisit this when we cover combinational circuits later.

- A simplified 1-bit MIPS ALU can be implemented as follows:

- 4 control bits are needed:
  - **`Ainvert`**:
    - 1 to invert input A
  - **`Binvert:`**
    - 1 to invert input B
  - **`Operation`** (2-bit)
    - To select one of the 3 results

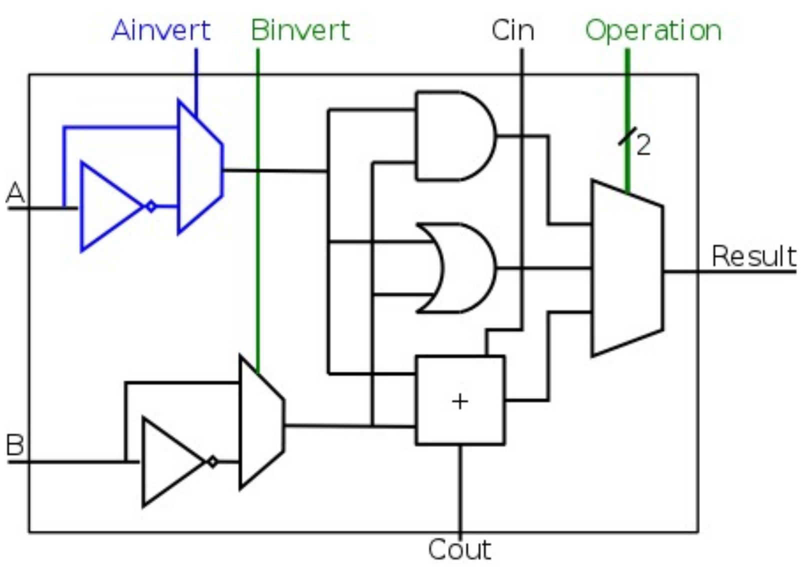


Acknowledgement: Image taken from NYU Course CSCI-UA.0436

5. One Bit At A Time (Aha!)

- Can you see how the **`ALUcontrol`** (4-bit) signal controls the ALU?
  - Note: implementation for **`slt`** not shown

ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR



Acknowledgement: Image taken from NYU Course CSCI-UA.0436

5. Multilevel Decoding

- Now we can start to design for `ALUcontrol` signal, which depends on:
  - Opcode (6-bit) field **and** Function Code (6-bit) field
- **Brute Force approach:**
  - Use Opcode and Function Code directly, i.e. finding expressions with 12 variables
- **Multilevel Decoding approach:**
  - Use some of the input to reduce the cases, then generate the full output
  - Simplify the design process, reduce the size of the main controller, potentially speedup the circuit

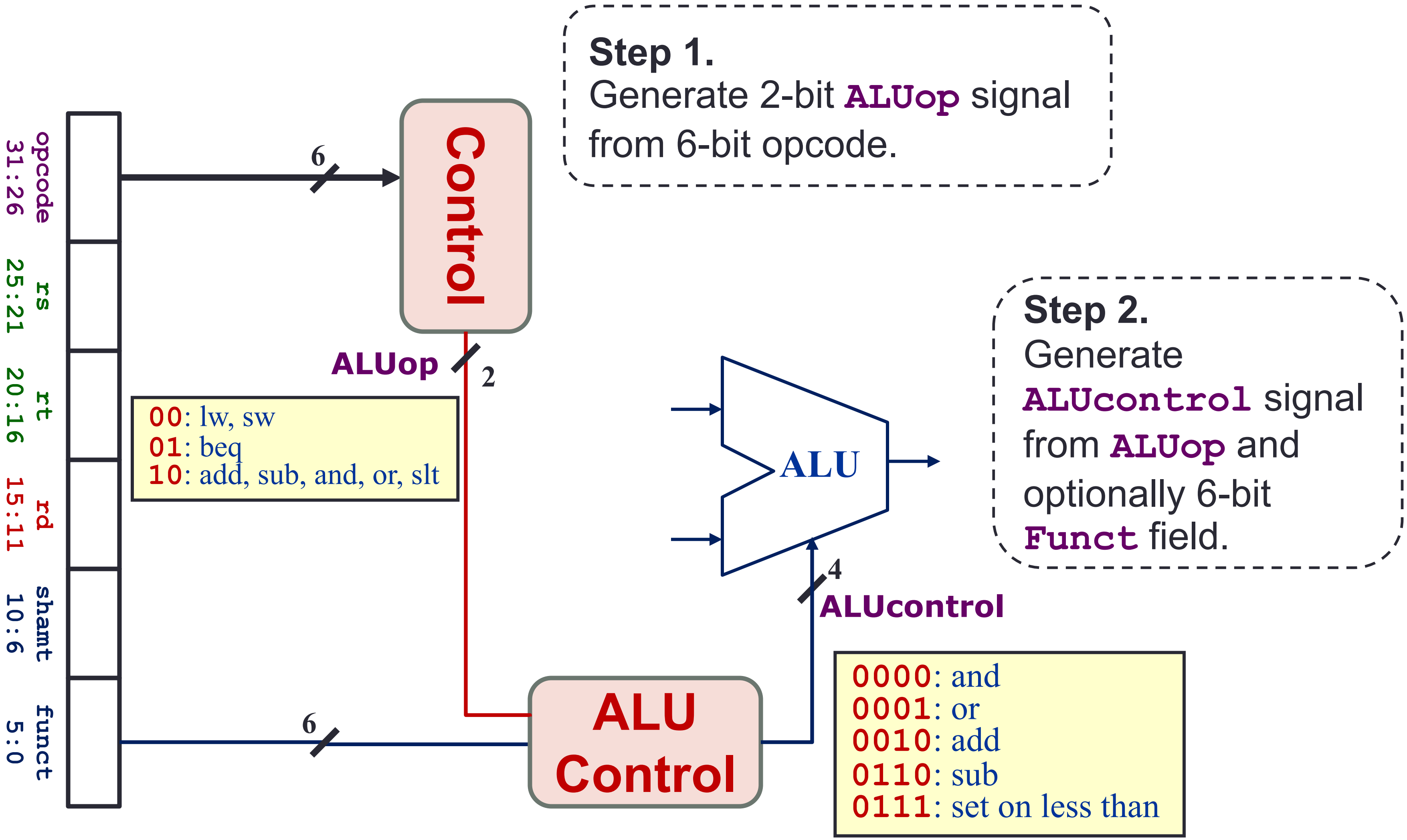
5. Intermediate Signal: `ALUop`

- **Basic Idea:**
  1. Use Opcode to generate a 2-bit **`ALUop`** signal
     - Represents classification of the instructions:

Instruction type	ALUop
lw / sw	00
beq	01
R-type	10

  2. Use **`ALUop`** signal and Function Code field (for R-type instructions) to generate the 4-bit **`ALUcontrol`** signal

5. Two-level Implementation



5. Generating ALUcontrol Signal

Opcode	ALUop	Instruction Operation	Funct field	ALU action	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	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Generation of 2-bit ALUop signal will be discussed later

5. Design of ALU Control Unit (1/2)

- Input: 6-bit **`Funct`** field and 2-bit **`ALUop`**
- Output: 4-bit **`ALUcontrol`**
- Find the simplified expressions

ALUcontrol3 = 0

ALUcontrol2 = ?

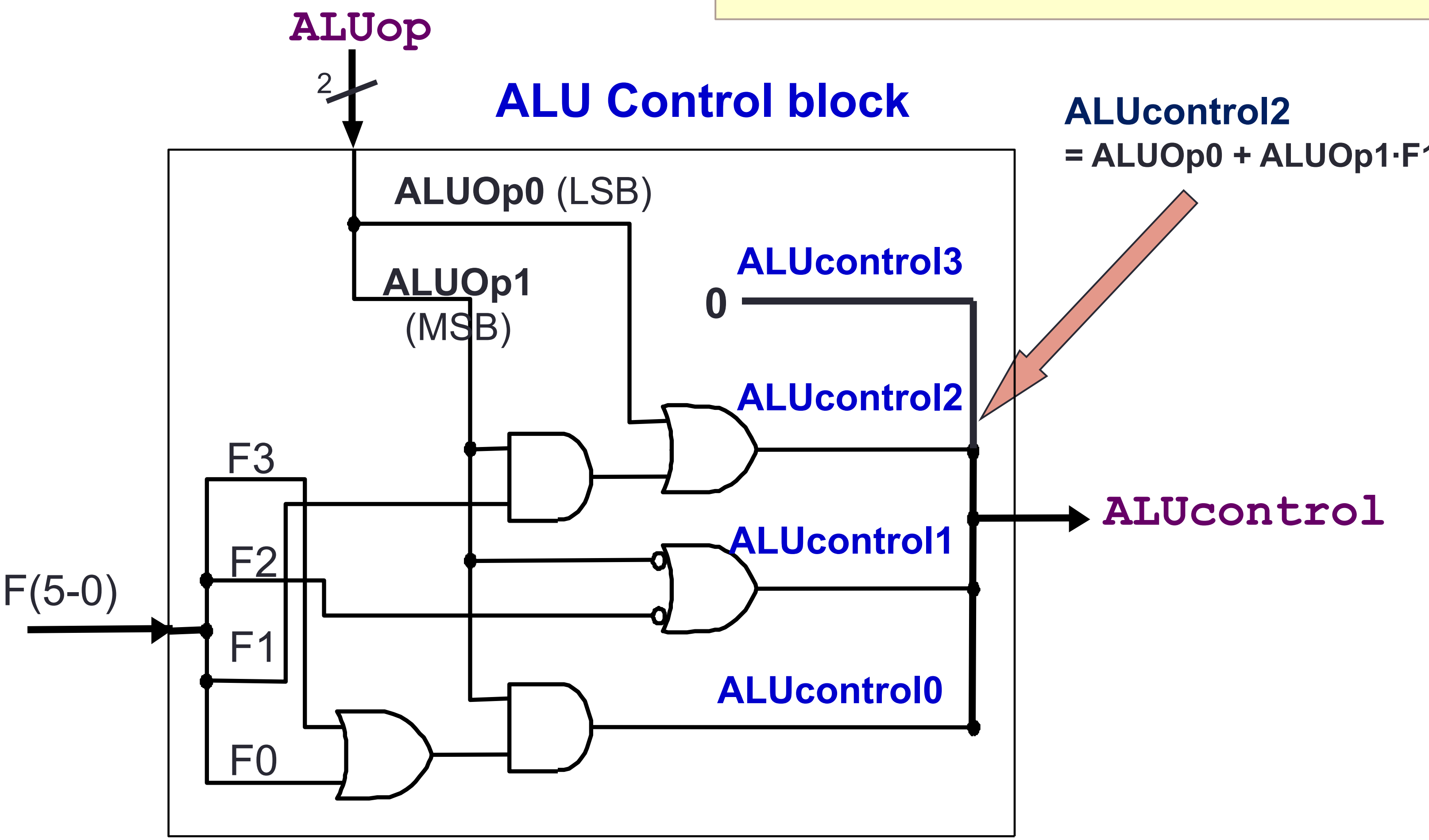
ALUop0 + ALUop1· F1

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 1 1 1

5. Design of ALU Control Unit (2/2)

- Simple combinational logic

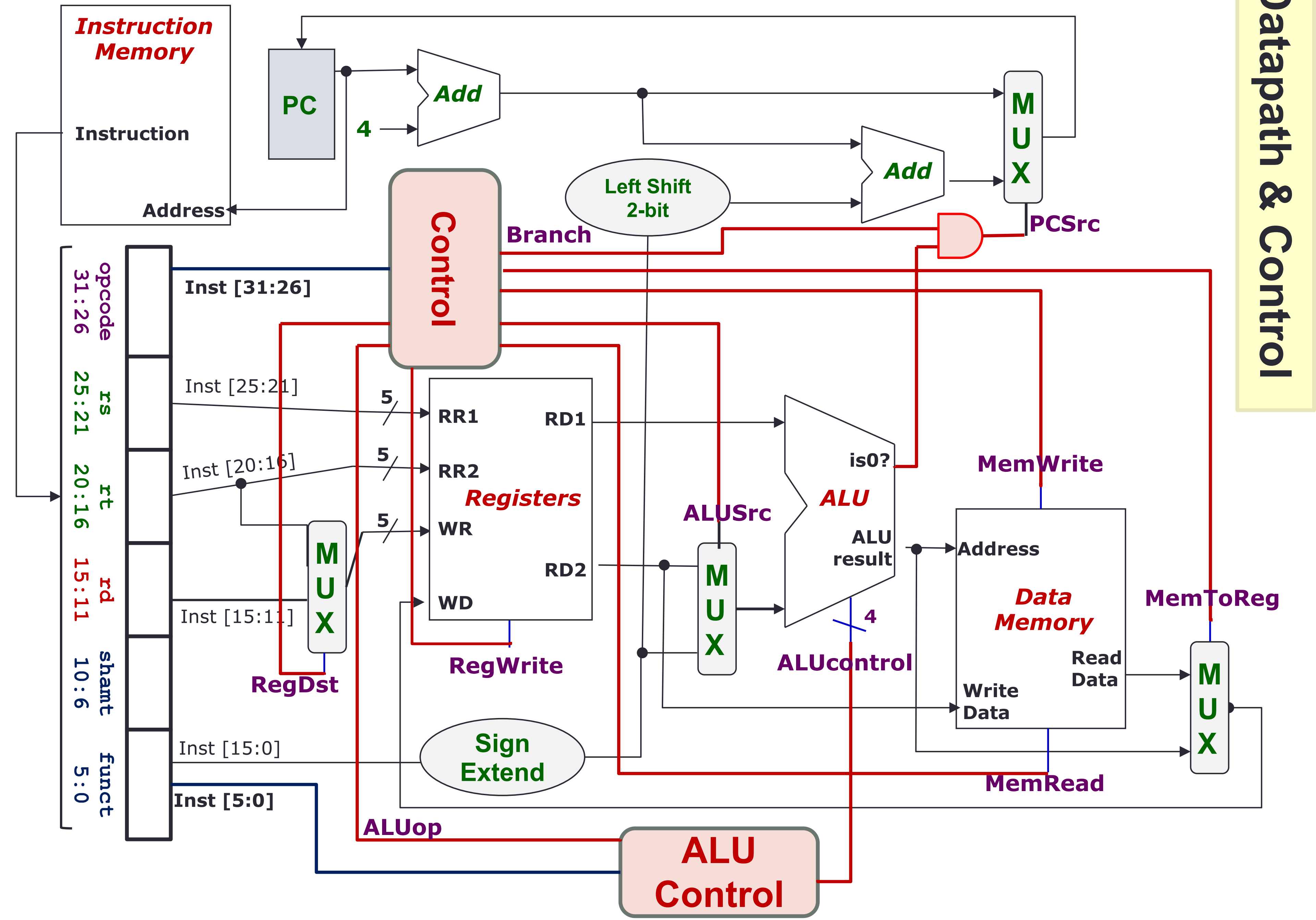
Note: We will revisit this when we cover combinational circuits later.



5. Finale: Control Design

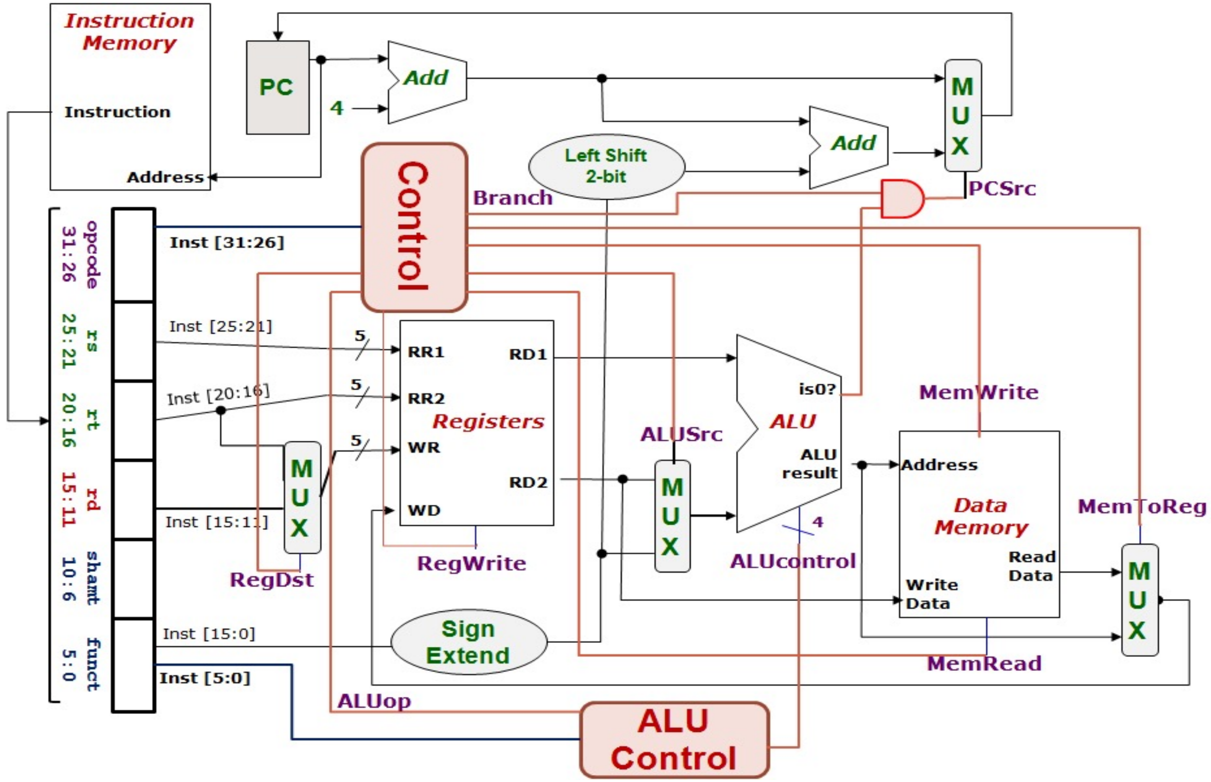
- We have now considered all individual signals and their expected values
  - ➔ Ready to design the controller itself
- Typical digital design steps:
  - Fill in truth table
    - **Input:** Opcode
    - **Output:** Various control signals as discussed
  - Derive simplified expression for each signal

Datapath & Control



5. Control Design: **Outputs**

	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



5. Control Design: **Inputs**

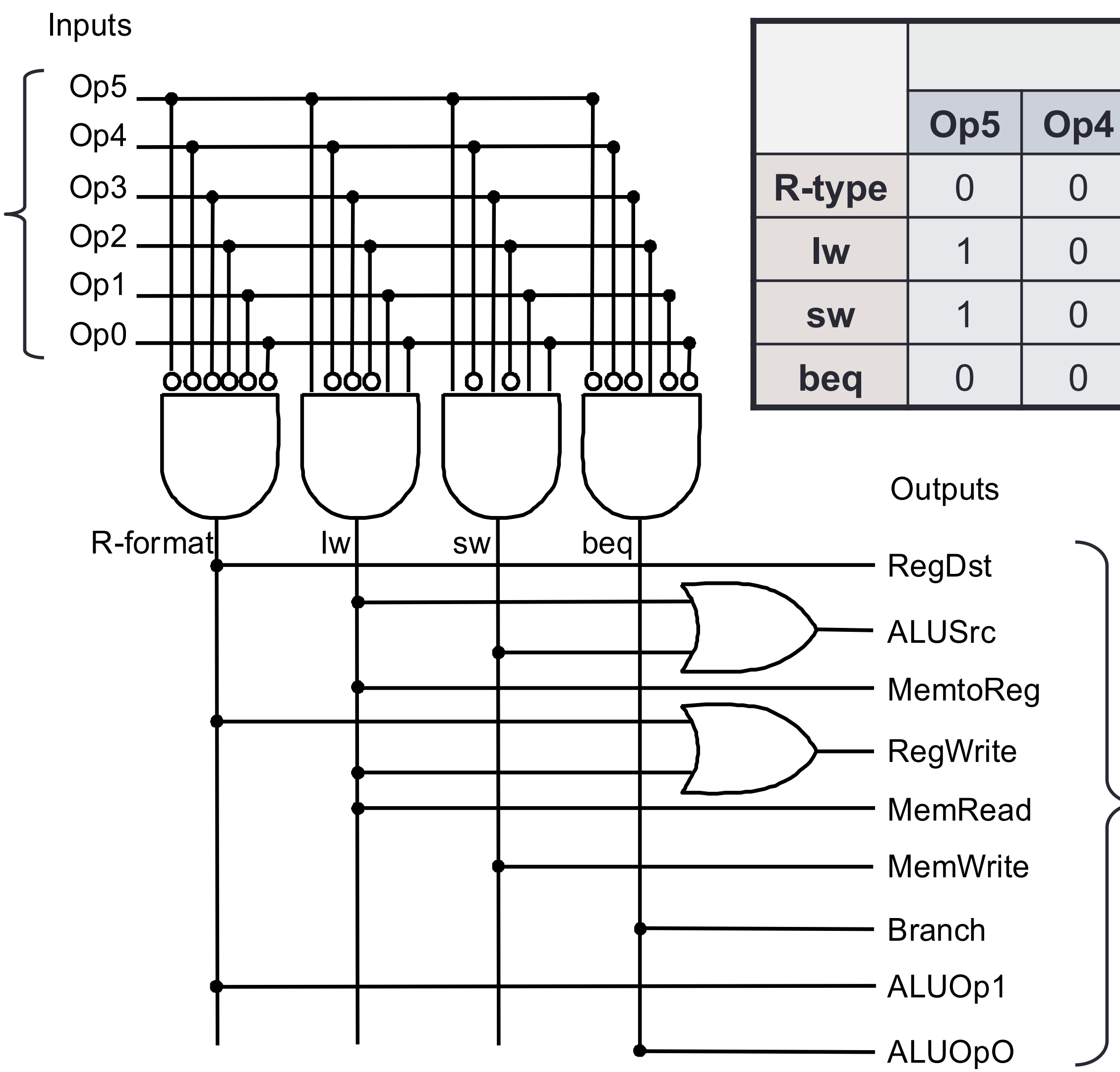
	Opcode (`Op[5:0] == Inst[31:26]`)						
	Op5	**Op4**	**Op3**	**Op2**	**Op1**	**Op0**	Value in Hexadecimal
R-type	**0**	**0**	**0**	**0**	**0**	**0**	**0**
lw	**1**	**0**	**0**	**0**	**1**	**1**	**23**
sw	**1**	**0**	**1**	**0**	**1**	**1**	**2B**
beq	**0**	**0**	**0**	**1**	**0**	**0**	**4**

- With the input (opcode) and output (control signals), let's design the circuit

5. Combinational Circuit Implementation

	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

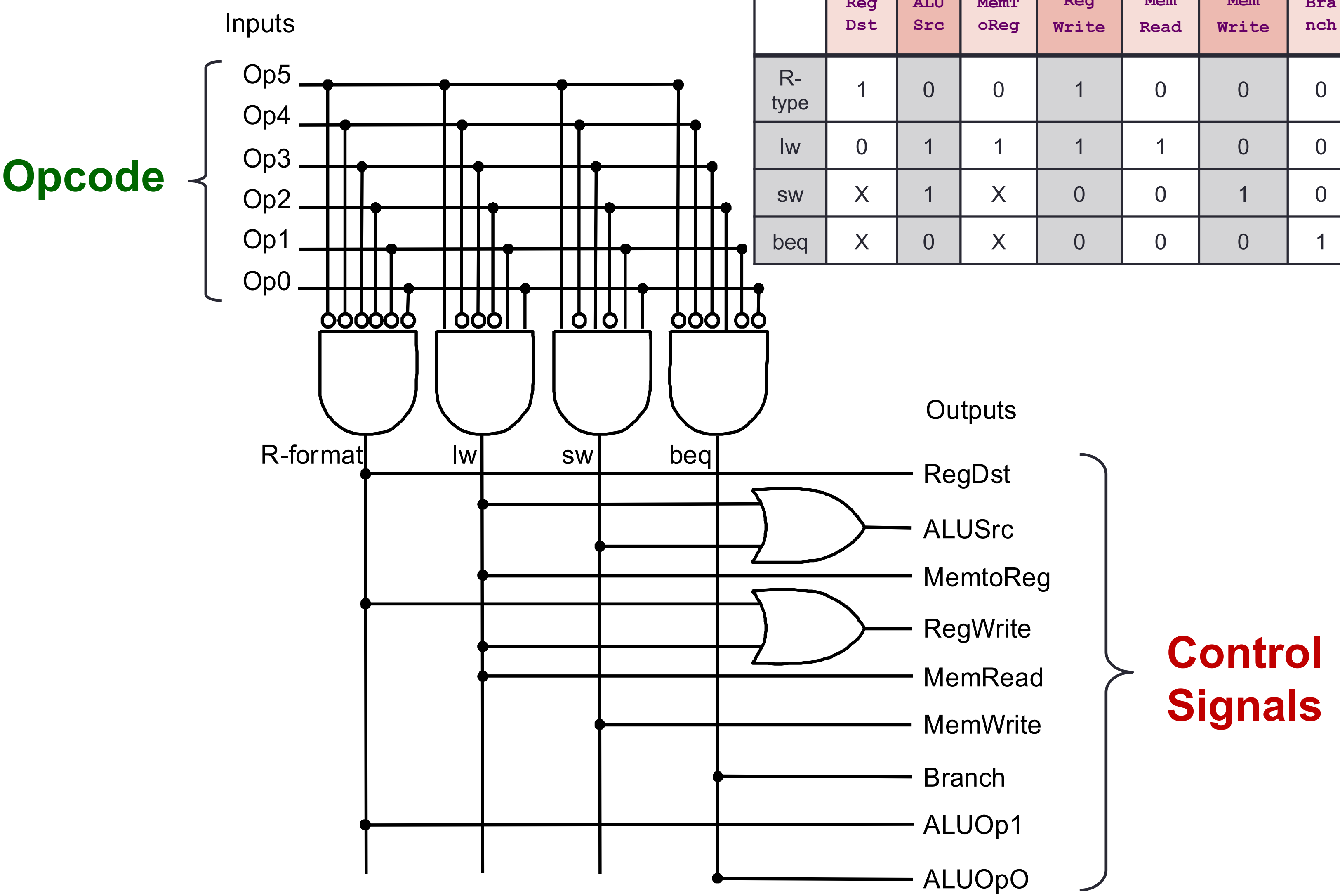
Opcode



Control Signals

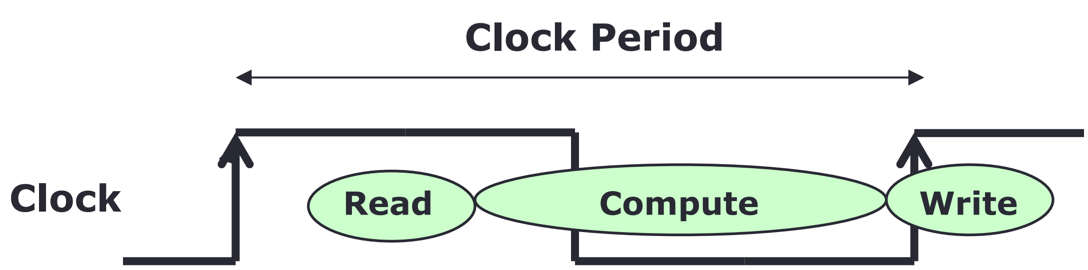
5. Combinational Circuit Implementation

	Reg Dst	ALU Src	MemT oReg	Reg Write	Mem Read	Mem Write	Bra nch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



6. Big Picture: Instruction Execution

- Instruction Execution =
  1. Read contents of one or more storage elements (register/memory)
  2. Perform computation through some combinational logic
  3. Write results to one or more storage elements (register/memory)
- All these performed **within a clock period**



Don't want to read a storage element when it is being written.

6. Single Cycle Implementation: Shortcoming

- Calculate cycle time assuming negligible delays: memory (2ns), ALU/adders (2ns), register file access (1ns)

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	**6**
lw	2	1	2	2	1	**8**
sw	2	1	2	2		**7**
beq	2	1	2			**5**

- All instructions take as much time as the slowest one (i.e., load)

  ➔ Long cycle time for each instruction

6. Solution #1: Multicycle Implementation

- Break up the instructions into execution steps:
  1. Instruction fetch
  2. Instruction decode and register read
  3. ALU operation
  4. Memory read/write
  5. Register write
- Each execution step **takes one clock cycle**

  ➔ **Cycle time is much shorter, i.e., clock frequency is much higher**
- Instructions take <u>variable number of clock cycles</u> to complete execution
- Not covered in class:
  - See Section 5.5 of COD if interested

6. Solution #2: Pipelining

- Break up the instructions into execution steps one per clock cycle
- Allow <u>different instructions to be in different execution steps simultaneously</u>
- Covered in a later lecture

Summary

- A very simple implementation of MIPS datapath and control for a subset of its instructions
- Concepts:
  - An instruction executes in a single clock cycle
  - Read storage elements, compute, write to storage elements
  - Datapath is shared among different instructions types using MUXs and control signals
  - Control signals are generated from the machine language encoding of instructions

Reading

- The Processor: Datapath and Control
  - COD Chapter 5 Sections 5.4 (3rd edition)
  - COD Chapter 4 Sections 4.4 (4th edition)

- Exploration:
  - ALU design and implementation:
    - 4th edition (MIPS): Appendix C
    - http://cs.nyu.edu/courses/fall11/CSCI-UA.0436-001/class-notes.html

End of File