

CSE 4305
Computer Organization and Architecture

Lecture 4

The Processor

Sabrina Islam
Lecturer, CSE, IUT
Contact: +8801832239897
E-mail: sabrinaislam22@iut-dhaka.edu

Background

The performance of a computer is determined by three key factors:

- **Instruction count**
 - The number of instructions executed by the program
- **Clock cycle time**
 - The time for one clock period, in which the smallest unit of activity of a processor is executed and it runs at a constant rate.
- **Clock cycles per instruction (CPI)**
 - Average number of clock cycles per instruction for a program or program fragment.

Background

Previous Lecture ->


- Instruction Set Architecture
- ISA determines the instruction count

This Lecture ->

- Implementation of the Processor
- It determines both the clock cycle time and CPI

Basic implementation of RISC-V ISA

A subset of the core RISC-V instruction set:


- **The memory-reference instructions:**
 - load doubleword (ld) and store doubleword (sd)
 - **The arithmetic-logical instructions**
 - add, sub, and, and or
 - **The conditional branch instruction**
 - branch if equal (beq)
- 

This subset of instructions illustrates the key principles used in creating a datapath and designing the control.

Overview of the implementation

Steps to implement the instructions:

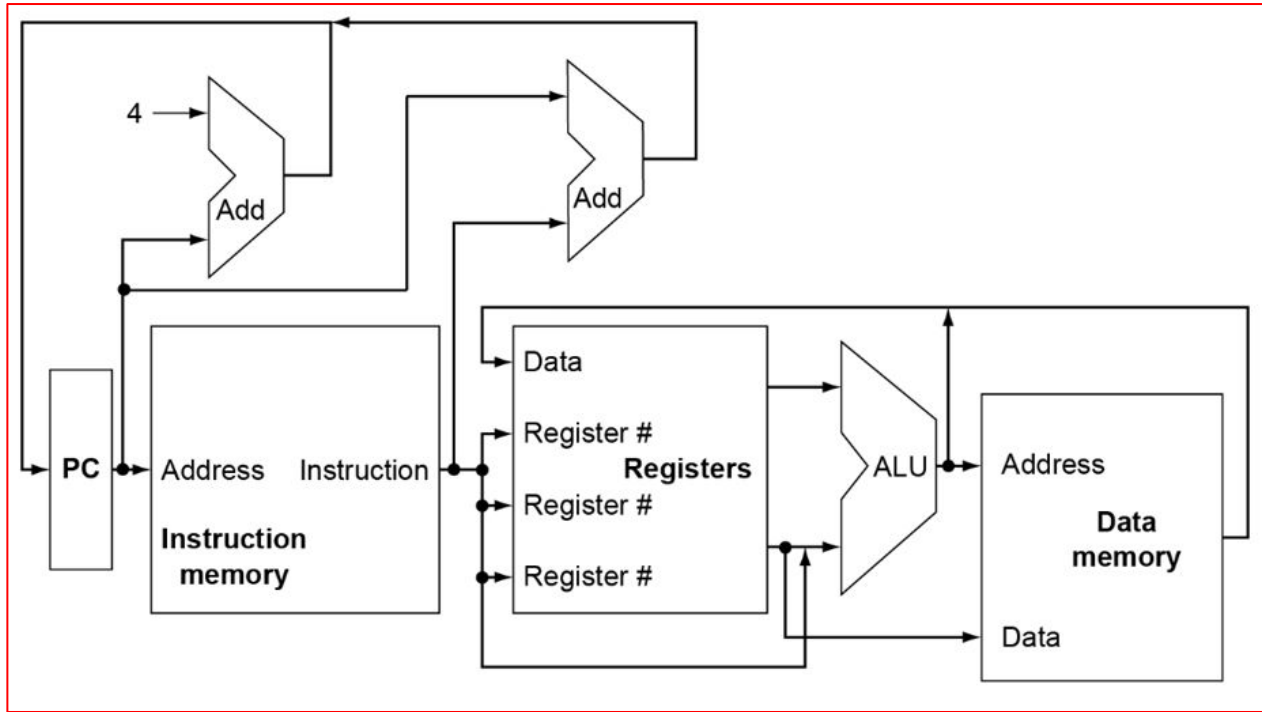
1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read.
3. Even though the remaining steps depend on the instruction class, they are largely the same.



The first 2 steps are the same for all instructions.

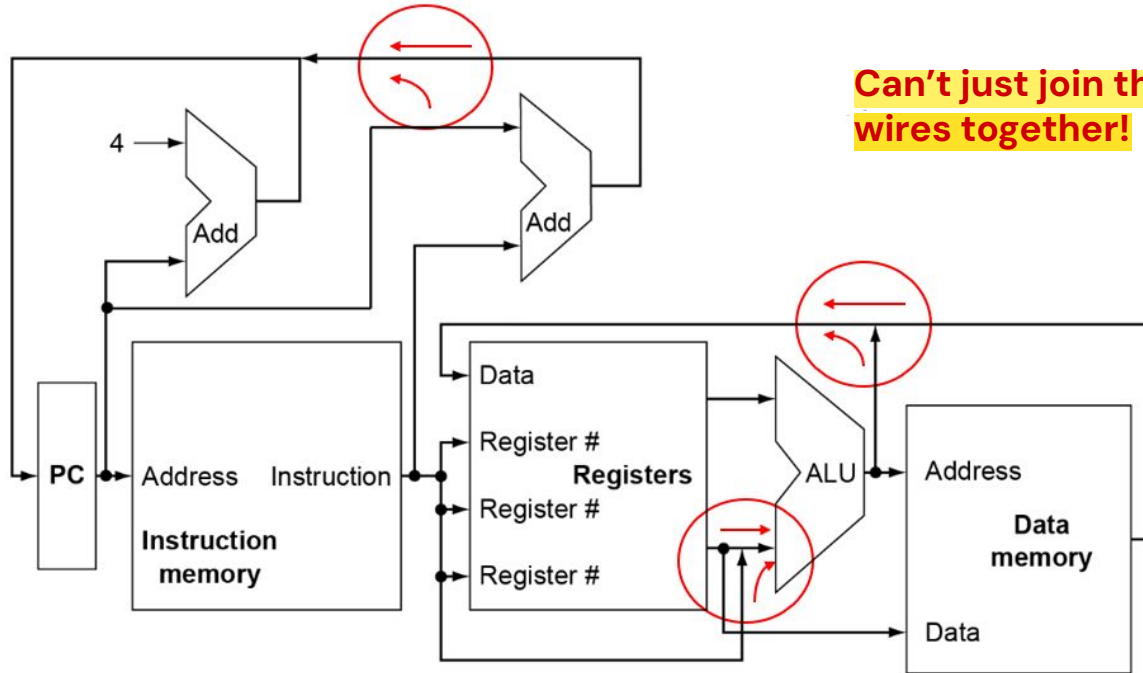
Depends on the function of the instruction

Overview of the implementation



An abstract view of the implementation of the RISC-V subset showing the major functional units and the major connections between them.

Overview of the implementation

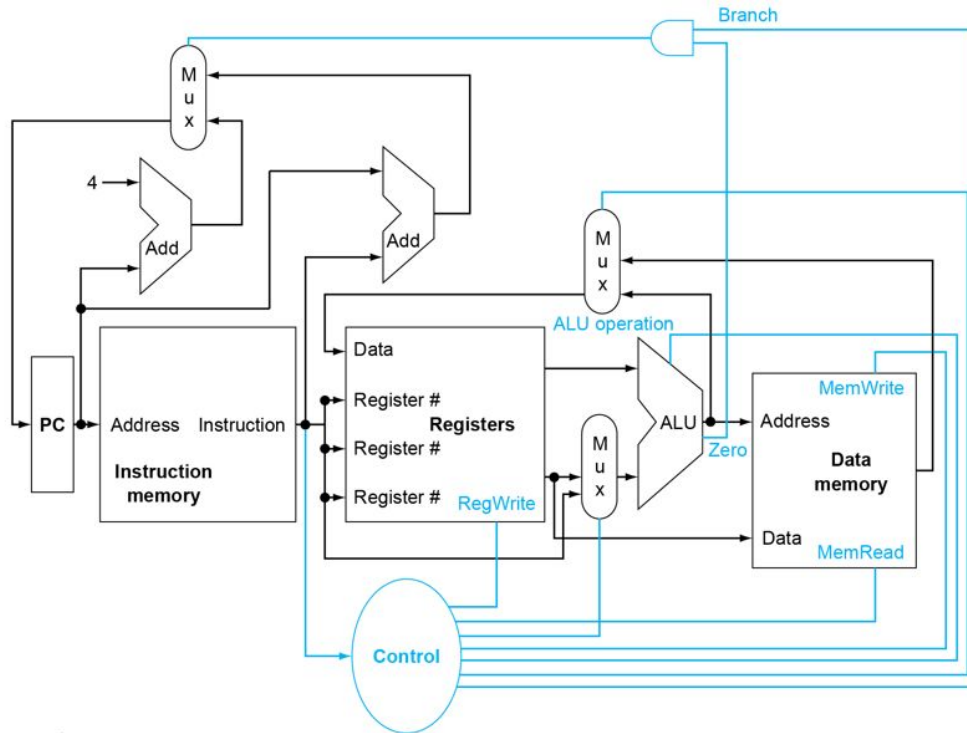


Can't just join the wires together!

WHY???

Which operation is performed by the wires in data memory : read or write?

Overview of the implementation



The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines.

Overview of the implementation

- Multiplexor
 - A logic element that selects from among several inputs based on the setting of its control lines.
- Control Unit
 - It has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors.

Understanding the Hardware

To design a computer, at first we must get a clear idea about:

- The hardware logic
 - How it will operate.
- Clocking methodology
 - How the computer is clocked.

Logic Design Conventions

- Information encoded in binary
 - Low voltage = 0, High voltage = 1
 - One wire per bit
 - Multi-bit data encoded on multi-wire buses
- Two different types of logic elements:
 - Combinational elements
 - State (sequential) elements

Combinational Elements

- Outputs depend only on the current inputs.
- Given the same input, a combinational element always produces the same output.
- No storage available.
- Example: AND gate, Multiplexor, ALU, etc.

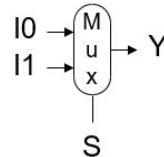
AND-gate

$$Y = A \& B$$



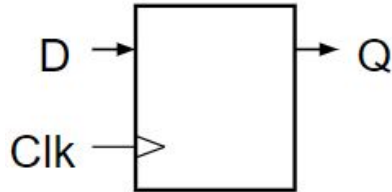
Multiplexer

$$Y = S ? I1 : I0$$



Sequential Elements

- Contains state.
- Outputs depend on the current inputs and previous inputs (state)
- Storage available.
- Example: Register or a memory, etc.



Clocking Methodology

The approach used to determine when data are valid and stable relative to the clock.

- Defines when signals can be read and when they can be written.
- It is designed to make hardware **predictable**. **How?**
 - A simultaneous read and write may result in reading the old value, new value or even a mix of the two. Such unpredictability is prevented by the clocking method.

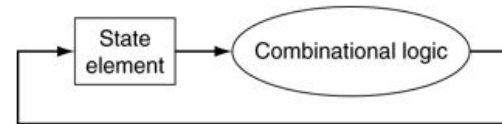
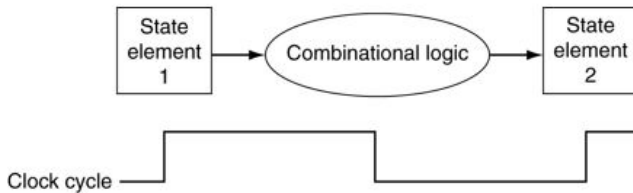
Edge-triggered clocking

A clocking scheme in which all state changes occur on a clock edge.

- Any values stored in a sequential logic element are updated only on a clock edge.
 - Clock edge: A quick transition from low to high or vice versa
- Combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements.

Edge-triggered clocking

- All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of **one clock cycle**.
- No feedback within a single clock cycle.



Control Signal

- A signal used for multiplexor selection or for directing the operation of a functional unit.
- Must be separated from the data signals.
- Only when a state element is not updated on every clock, an explicit write control signal is required.
 - The state element is changed only when the write control signal is asserted and a clock edge occurs.
- **asserted** and **deasserted**
 - **asserted**: the signal is logically high or true (1).
 - **deasserted**: the signal is logically low or false (0).

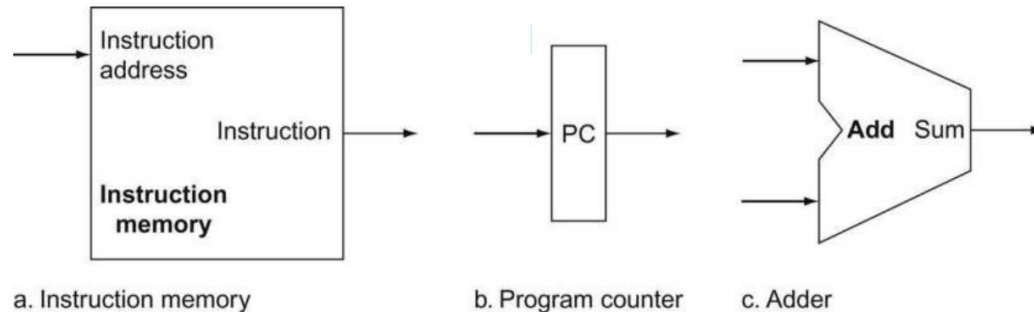
Building a Datapath

Building a datapath

- The datapath is responsible for executing instructions and performing data processing operations in a CPU.
- The datapath, along with the control unit, forms the core processing unit of a CPU.
- **Datapath element**
 - A unit used to operate on or hold data within a processor.
 - In RISC-V: The instruction and data memories, the register file, the ALU, and adders.
- **PC**
 - The register containing the address of the instruction in the program being executed.

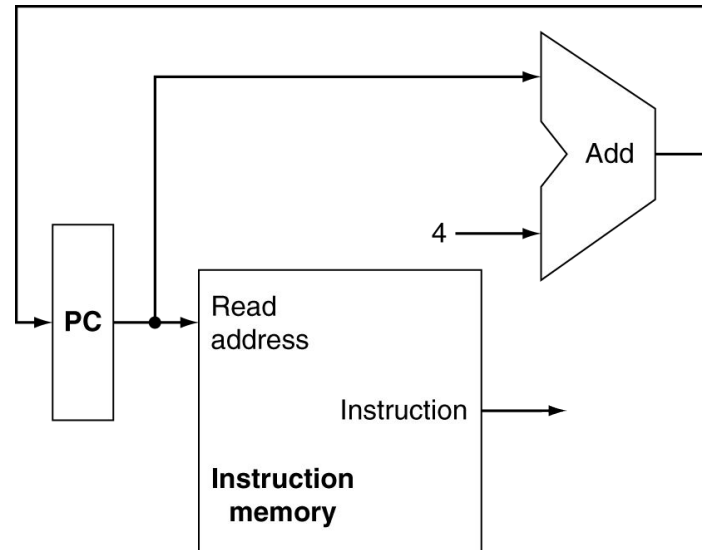
Datapath for Instruction Fetch

- **Instruction fetch includes:**
 - Fetching the instruction from **memory** given a specified address.
 - **Increment the program counter by 4 bytes for the next instruction.**
 - **The 4 bytes are added with the help of an adder.**
- **Datapath elements required for instruction fetch**
 - **Instruction memory, PC, Adder**



Datapath for Instruction Fetch

To form a datapath for instruction fetch, we need to combine the 3 elements.



A portion of the datapath used for fetching instructions and incrementing the program counter.

Datapath for Arithmetic-logical instructions

- add, sub, and, and or
- The R-format instructions
 - Reads registers
 - Performs ALU operation
 - Writes the result in a register.
- **Datapath elements**
 - Register file and ALU

Datapath for Arithmetic-logical instructions

Registers:

- From the register file, 2 data words are **read**
 - We need an input to the register file that specifies the **register number** to be read. No control signal is needed for read.
 - An output from the register file to carry the value that has been read.
- One word is **written** into the register file
 - We need 2 inputs for this operation
 - One to specify the register number to be written
 - Another to supply the data to be written into the register.
 - Writes are controlled by the write control signal.
 - It must be asserted for a write to occur at the clock edge.

Datapath for Arithmetic-logical instructions

Registers:

- Three 5-bit inputs for register numbers.
- One 64-bit input for data.
- Two 64-bit data output buses.

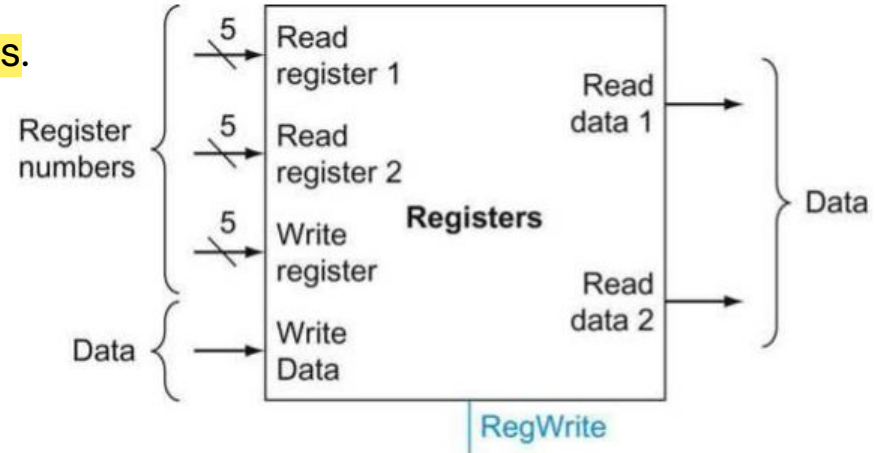


Fig: Register Files

Datapath for Arithmetic-logical instructions

ALU:

- Performs the operation specified in the instruction.
- Two 64-bit inputs.
- One 64-bit output
- 4-bit ALU operation signal
 - Specifies the operation type
- 1-bit Zero detection output
 - The bit is set to 1 if the result is zero.
 - Otherwise bit is set to 0.
 - Why do we need this detection?

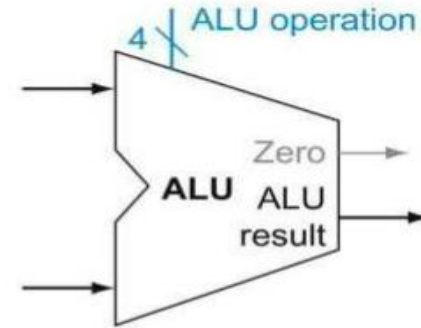


Fig: The ALU

Datapath for Load/Store instructions

The instructions:

- `ld x1, offset(x2)` – Load value from memory into reg x1.
- `sd x1, offset(x2)` – Store value from x1 into memory.
- So the datapath elements we need:
 - Data memory unit
 - Immediate generation unit
 - Also register files and ALU

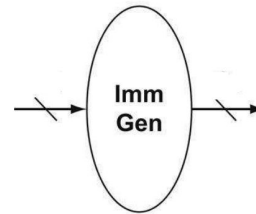


Fig: Immediate generation unit

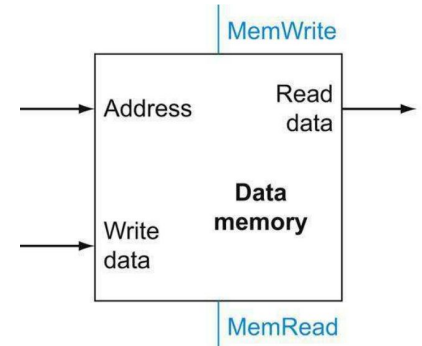


Fig: Data memory unit

Datapath for Load/Store instructions

Data Memory Unit:

- **Load**
 - One input for the address.
 - One output for the read result.
 - A read signal is needed because reading the value of an invalid address can cause problems.
- **Store**
 - Inputs for the address and data to be written.

There are separate read and write signals

- Only one of these may be asserted on any given clock.

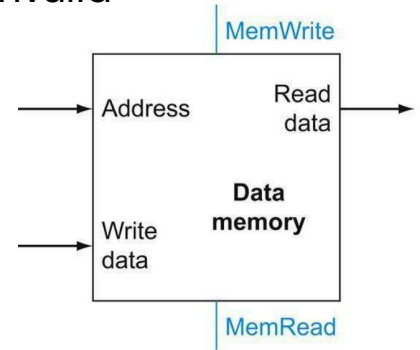


Fig: Data memory unit

Datapath for Load/Store instructions

Immediate Generation Unit (ImmGen):

- It gets an instruction as input.
- Selects a 12-bit field and converts it into a 32-bit or 64-bit value using sign-extend.
 - Sign-extend is the way increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger,

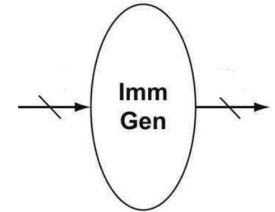


Fig: Immediate generation unit

Now let's look at the branch instructions!

```
if (i == j)
    f = g + h;
else
    f = g - h;
```

Compiled RISC-V code:

```
bne x22, x23, Else
add x19, x20, x21
beq x0, x0, Exit // unconditional
Else: sub x19, x20, x21
Exit: ...
```

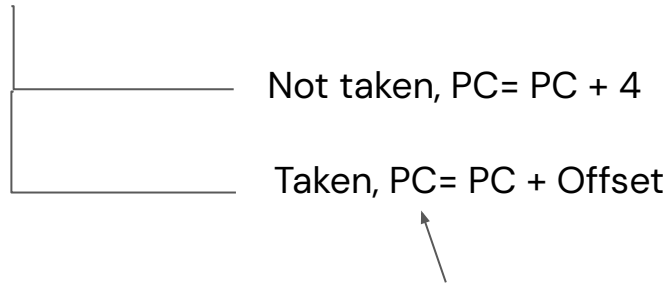
How do we get the address of
the Else and Exit?

In other words, how do we
determine how much to
jump?

Branch instructions

Branch instructions

- `bne x22, x23, Else`



Based on the comparison of x22 and x23 and the branch type, the branch can be taken or not taken. For this example:

- When ~~x22=x23~~, the branch will be **taken** and the Else labeled instruction will be executed next.
- If ~~x22=x23~~, the branch will be **not taken**, the next instruction will be executed.

Branch instructions

PC →	4	bne x22, x23, Else	# Branch Taken (PC = 4)
	8	add x19, x20, x21	.
	12	beq x0, x0, Exit	.
			.
		Else:	.
	16	sub x19, x20, x21	#Next line to be executed (PC=16)
	20	Exit: ...	

Branch instructions

bne x22, x23, Else

Offset

So, in branch instructions, we need:

- Two registers to compare
- An offset to determine which address to jump to.
- Branch target address
 - It is also called the **PC relative addressing**.
 - The address specified in a branch, which becomes the new program counter (PC) if the branch is taken.
 - It is the **sum** of the offset field of the instruction and the address of the branch.

Branch instructions

In PC relative addressing:

- Offset is calculated based on how many **instructions to skip** and the **instruction length**.

Byte address

4 bne x22, x23, Else

8 add x19, x20, x21

12 beq x0,x0,Exit

Else:

16 sub x19, x20, x21

20 Exit: ...

Branch Taken (PC = 4)

.

.

.

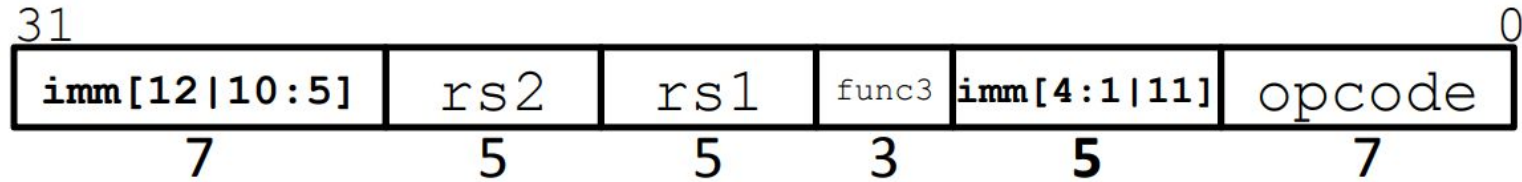
.

#Next line to be executed
(PC=16)

Offset

Branch instructions

The branch instructions follow the SB-format:



These 12 bits are for the offset

Branch instructions

As an extension, RISC-V allows variable length instructions

- As long as they are multiples of 16-bits in length (**must!**)

For 32 bit instructions, the offset will keep increasing by 4 always.

4 -	100
8 -	1000
12 -	1100



The offsets are even and have 0 in the least significant bit.

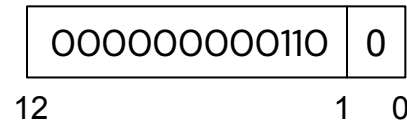
Branch instructions

As the LSB is always 0, we can use that space for an additional bit!

- Makes it possible to have **13 bits for offset**.
- The range increases.
- In the machine code, the insertion of the immediate bits (offset) is done from 1-12 instead of 0-11.
- And while sign-extending the immediate value, it is shifted left by 1 bit which automatically adds the 0 as least significant bit which we skipped earlier.

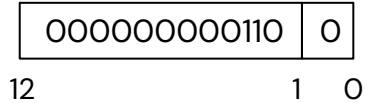
```
bne x22, x23, 0x12
```

Offset = 12 (000000001100)

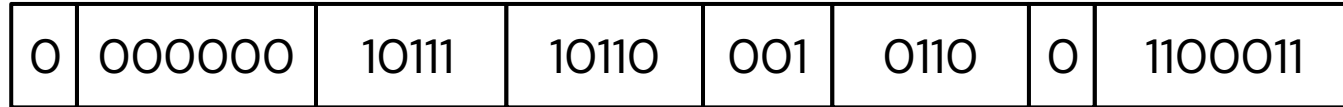
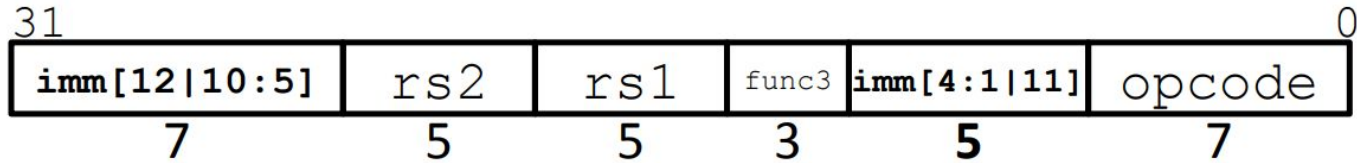
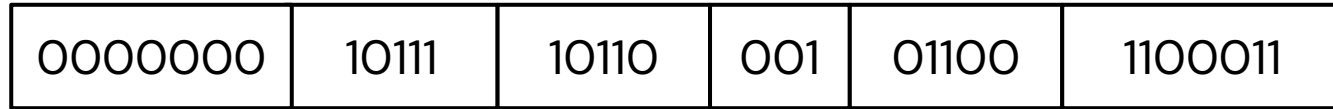


Branch instructions (SB-format)

Offset = 12 (0000000001100)



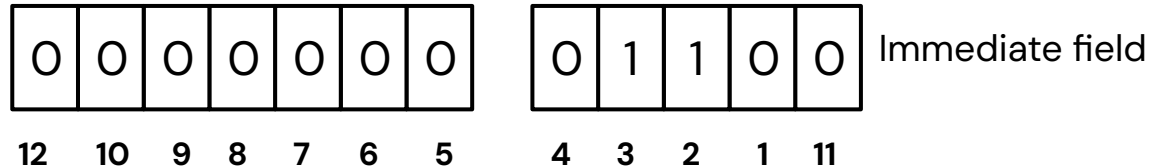
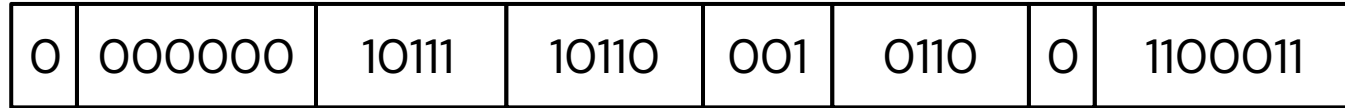
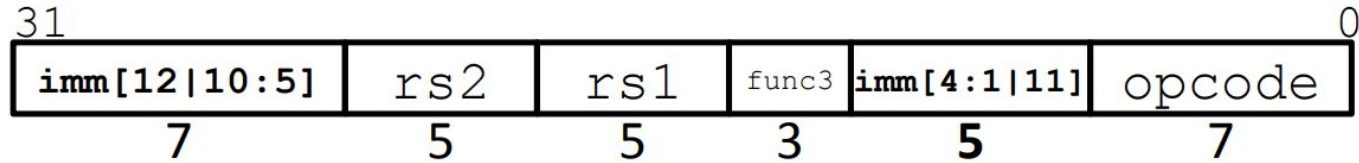
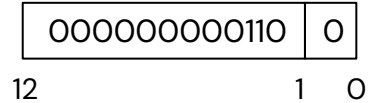
bne x22, x23, 0x12



Branch instructions (SB-format)

bne x22, x23, 0x12

Offset = 12 (0000000001100)

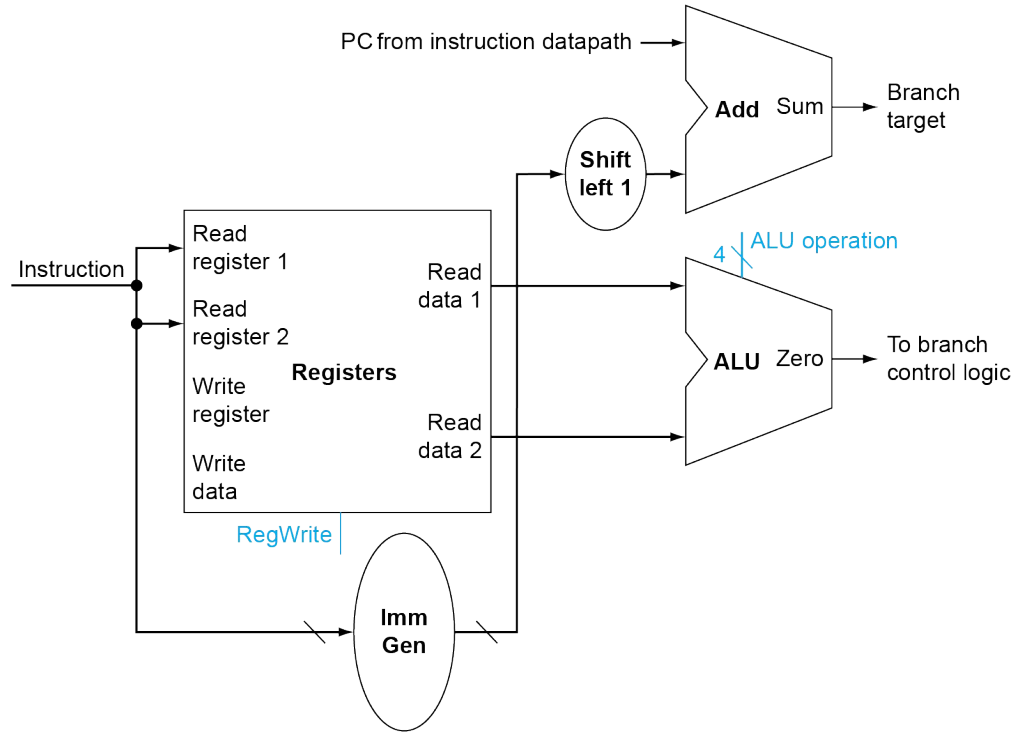


Building a datapath for branch instructions

The branch datapath must do two operations:

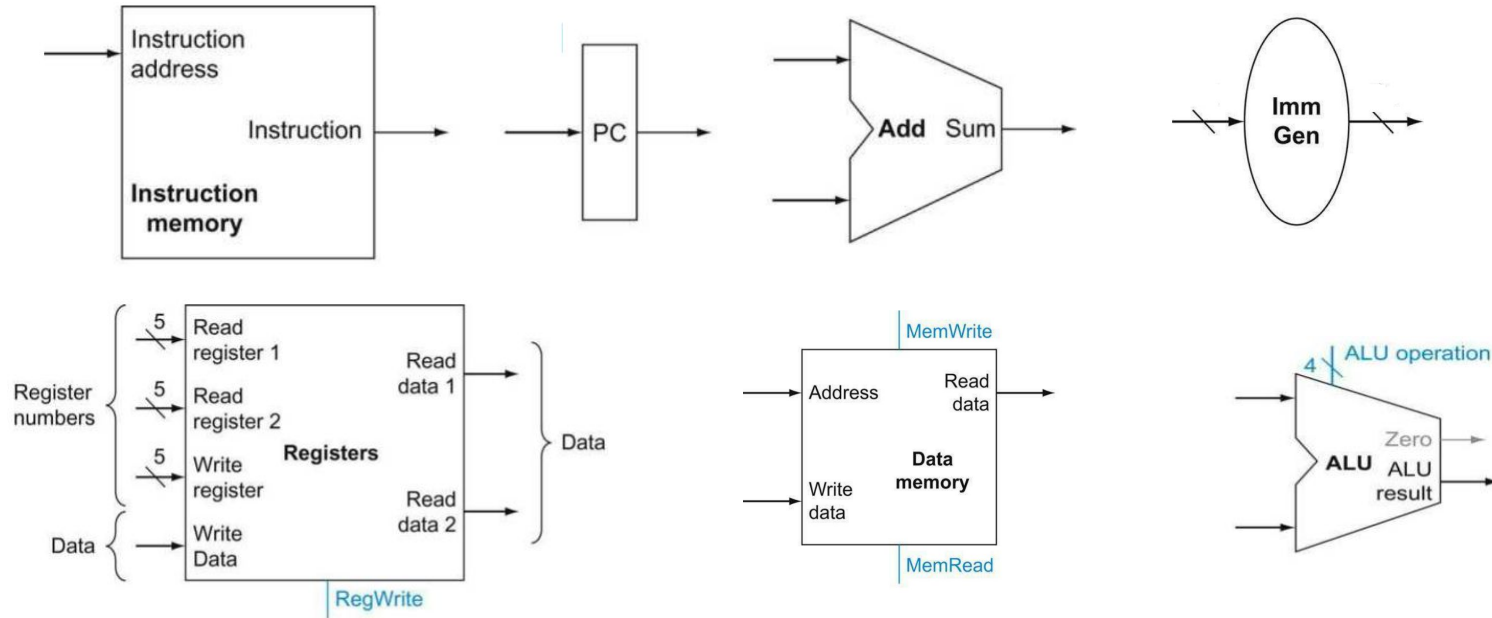
- Compute the branch target address.
 - an immediate generation unit
 - an adder
- Compare the register contents.
 - the register file
 - the ALU

Building a datapath for branch instructions



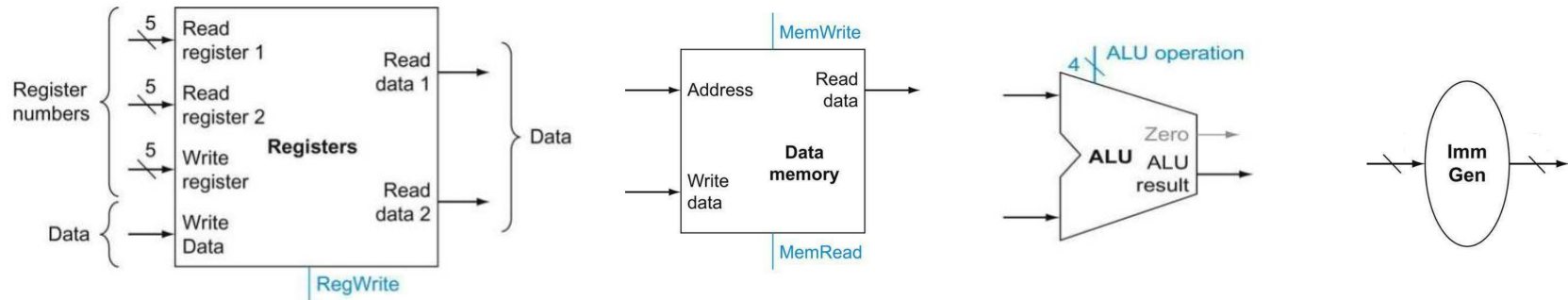
Building a single datapath

The different components required for all instruction classes:



Building a single datapath

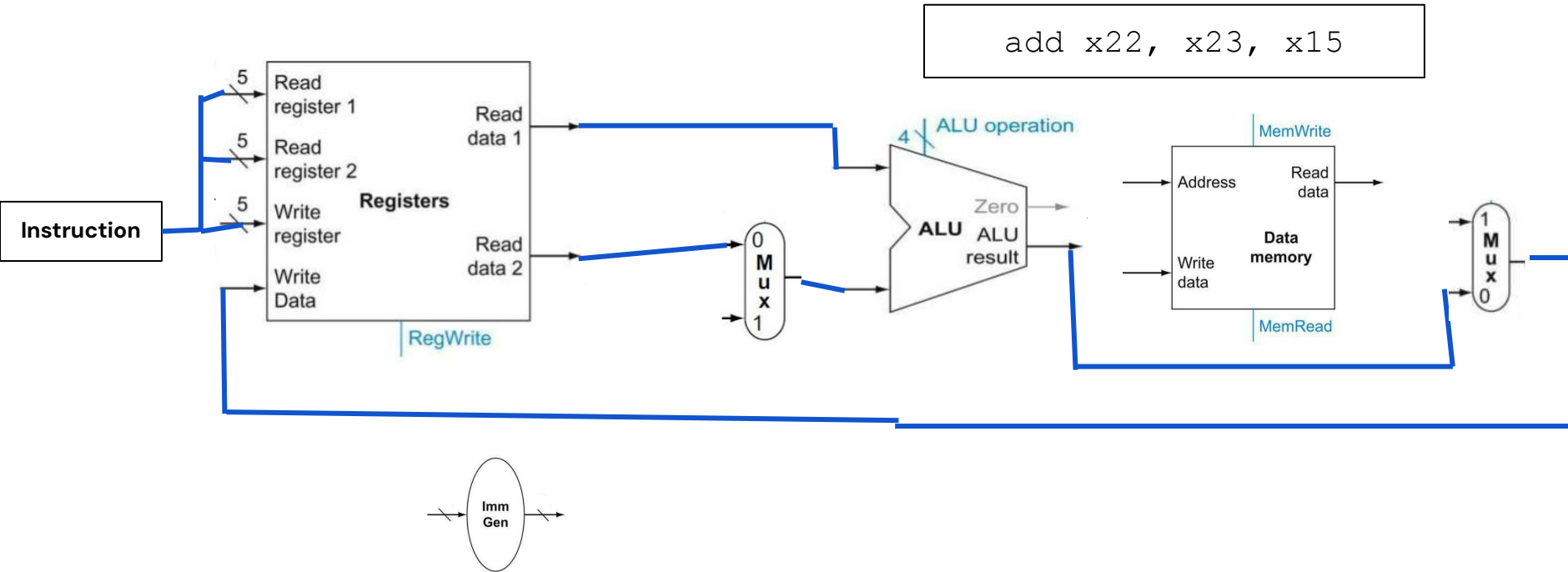
A datapath for the memory-reference and arithmetic-logical instructions
(a single register file and a single ALU and multiplexors)



Building a single datapath

A datapath for the memory-reference and arithmetic-logical instructions

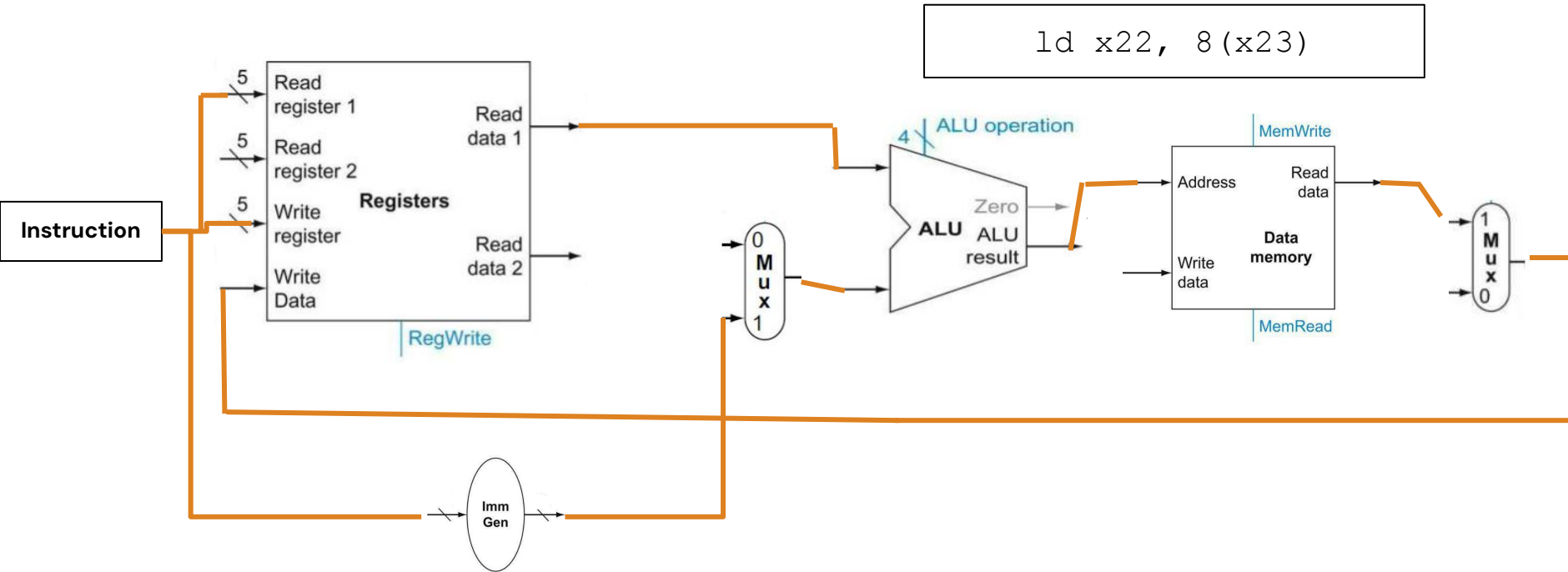
Arithmetic-logical instruction (R-format): Read from registers, operation, write into register.



Building a single datapath

A datapath for the memory-reference and arithmetic-logical instructions

Memory-reference (Load): Get data from memory and write into register.



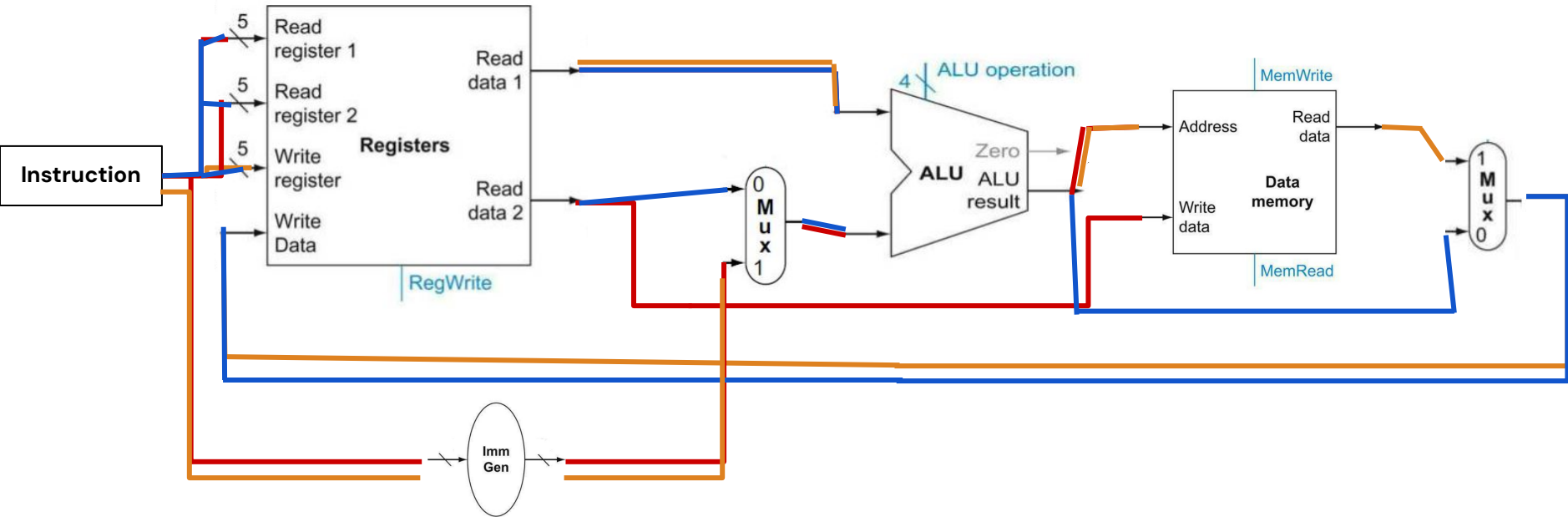
A datapath for the memory-reference and arithmetic-logical instructions

sd x22, 16(x23)



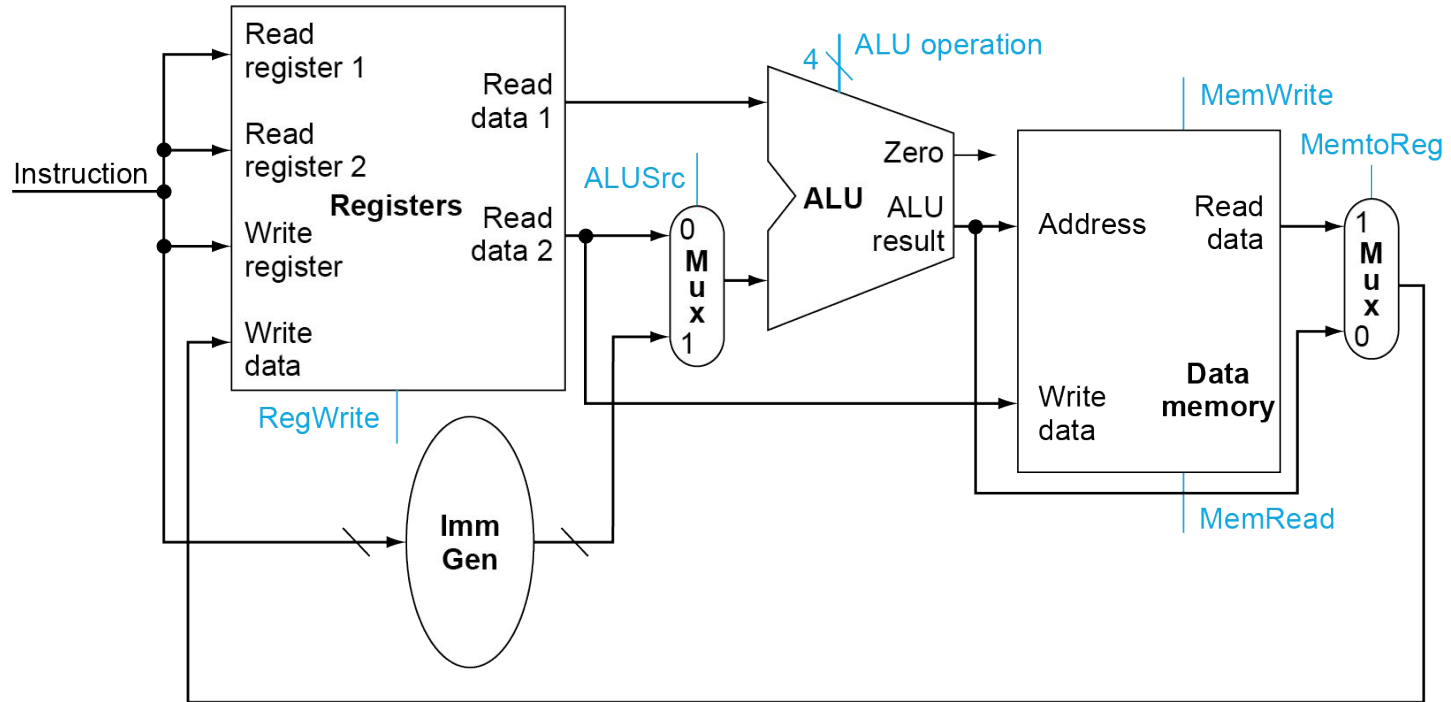
Building a single datapath

Combined path for memory-reference and arithmetic-logical instructions:



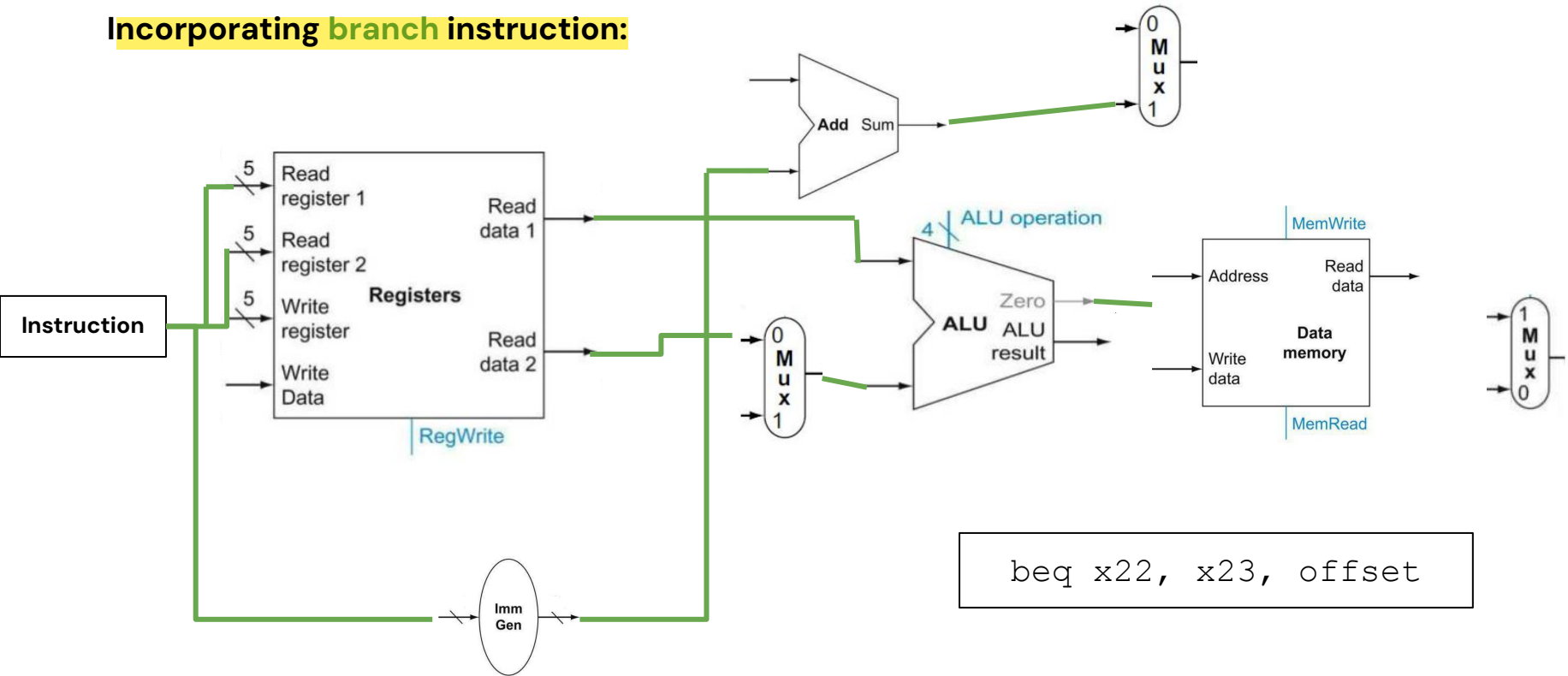
Building a single datapath

Combined path for memory-reference and arithmetic-logical instructions:



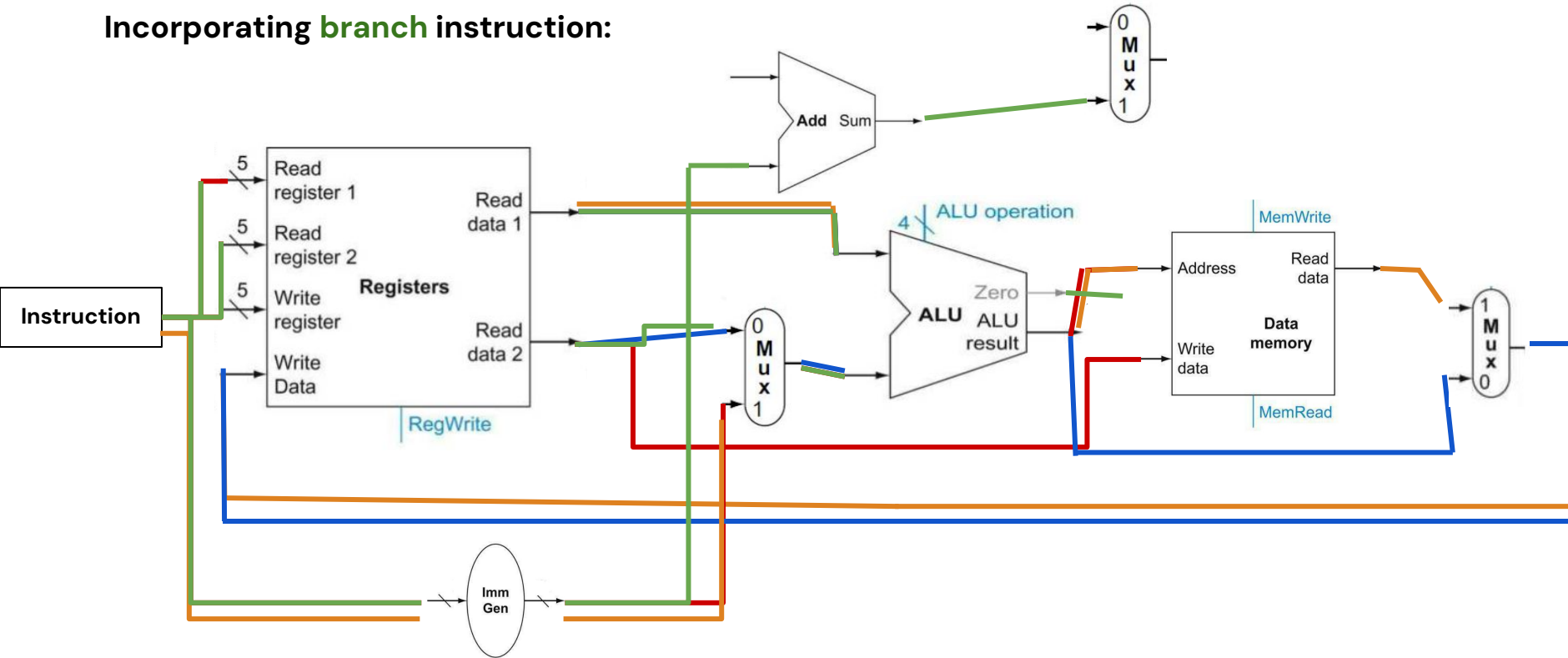
Building a single datapath

Incorporating **branch** instruction:



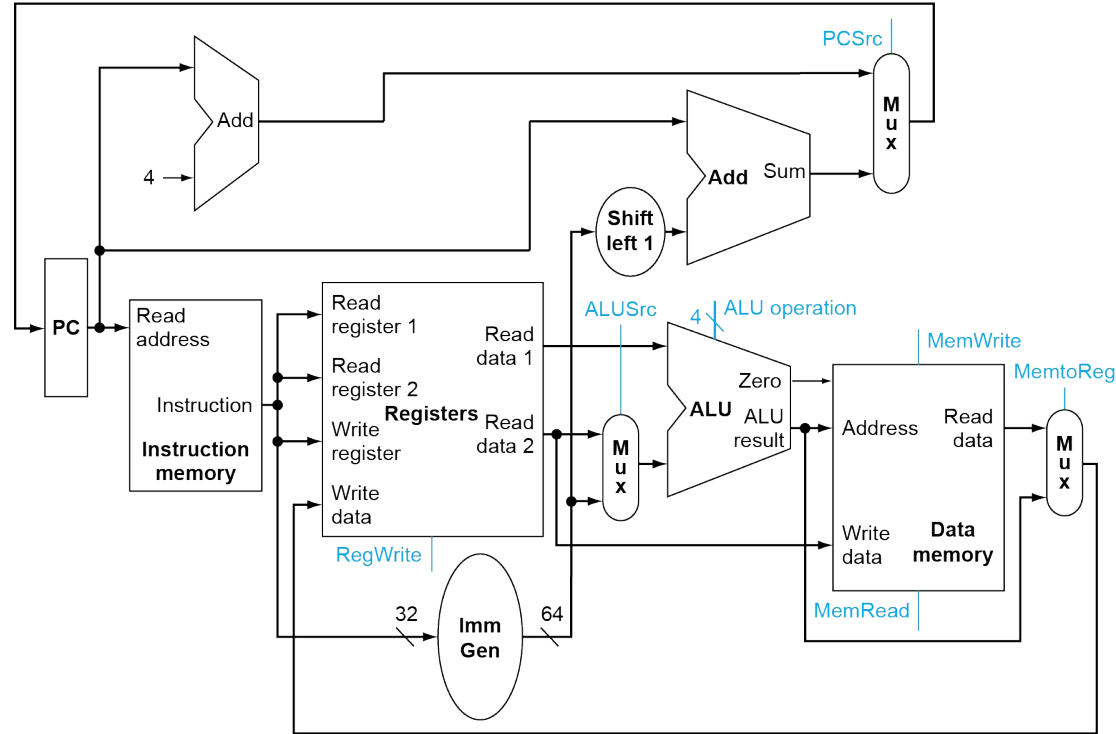
Building a single datapath

Incorporating **branch** instruction:

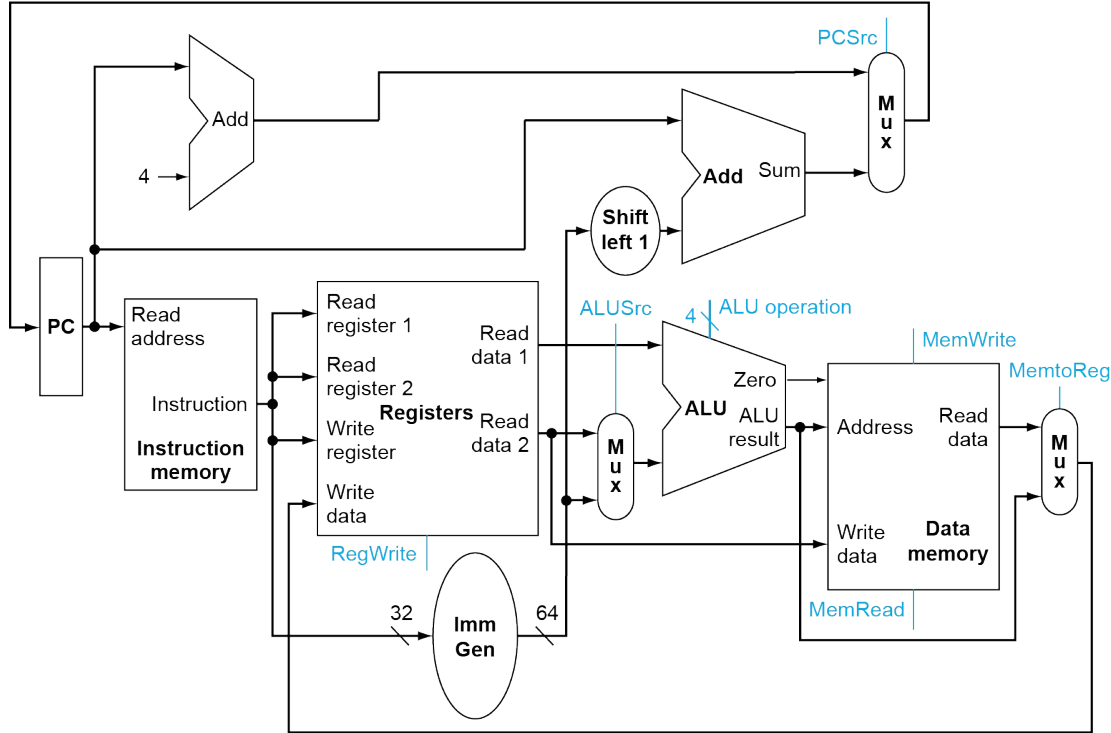


Building a single datapath

Combined path for **Fetch instruction, PC increment, Arithmetic-logic, Memory-ref and Branch:**



Building a single datapath



The control Unit is missing!

Designing the Control Unit

The ALU Control

Depending on the instruction class, the ALU will need to perform one of the following 4 functions:

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

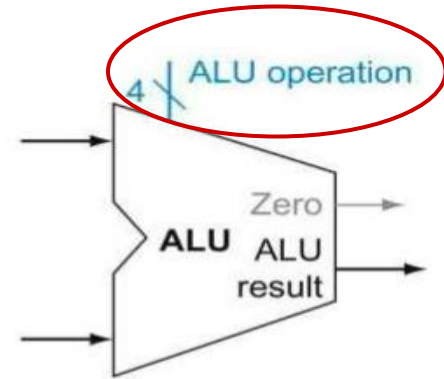
These 4 functions cover the operations required for all the instruction classes:

- **R-type:** Depends on opcode (add, sub, or, and)
- **Load/Store:** add
- **Branch (beq):** sub

The ALU Control

To generate the **4-bit ALU control input** we need:

- **2-bit control field (ALUOp)**
 - Used to determine the instruction type
 - 00 : Load/Store
 - 01 : Branch (beq)
 - 10 : R-type
- **funct7 and funct3**
 - Used to determine the R-type instruction



The ALU Control



Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

The ALU Control

The mapping from the 2-bit ALUOp field and the funct fields to the four ALU operation control bits.

- Recognizing the subset of possible values because
 - The *funct* fields only matter when ALUOp is 10.
 - Only a selected number of *funct* fields are important.
- Design a truth table.

ALUOp		Funct7 field							Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
Id/sd branch	0	0	X	X	X	X	X	X	X	X	X	0010
	X	1	X	X	X	X	X	X	X	X	X	0110
R-type	1	X	0	0	0	0	0	0	0	0	0	0010
	1	X	0	1	0	0	0	0	0	0	0	0110
	1	X	0	0	0	0	0	0	1	1	1	0000
	1	X	0	0	0	0	0	0	1	1	0	0001

The ALU Control

- The ALUOp does not use the encoding 11. So the truth table can contain **entries 1X and X1**.
- In the *funct* field, only bits 30, 14, 13, and 12 are interesting.

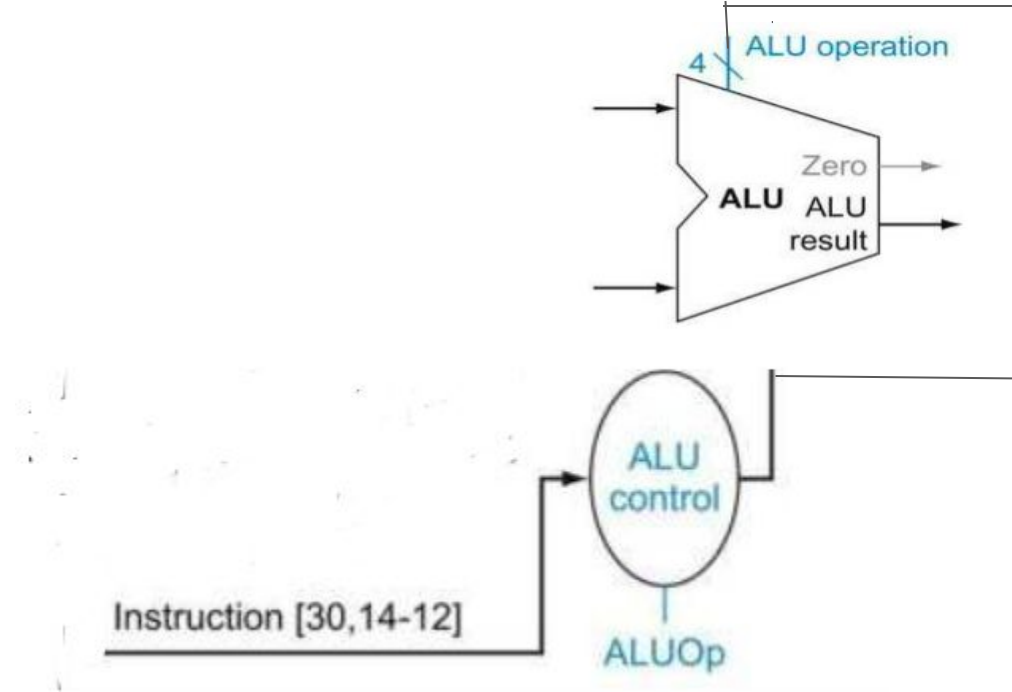
ALUOp		Funct7 field							Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

The ALU Control

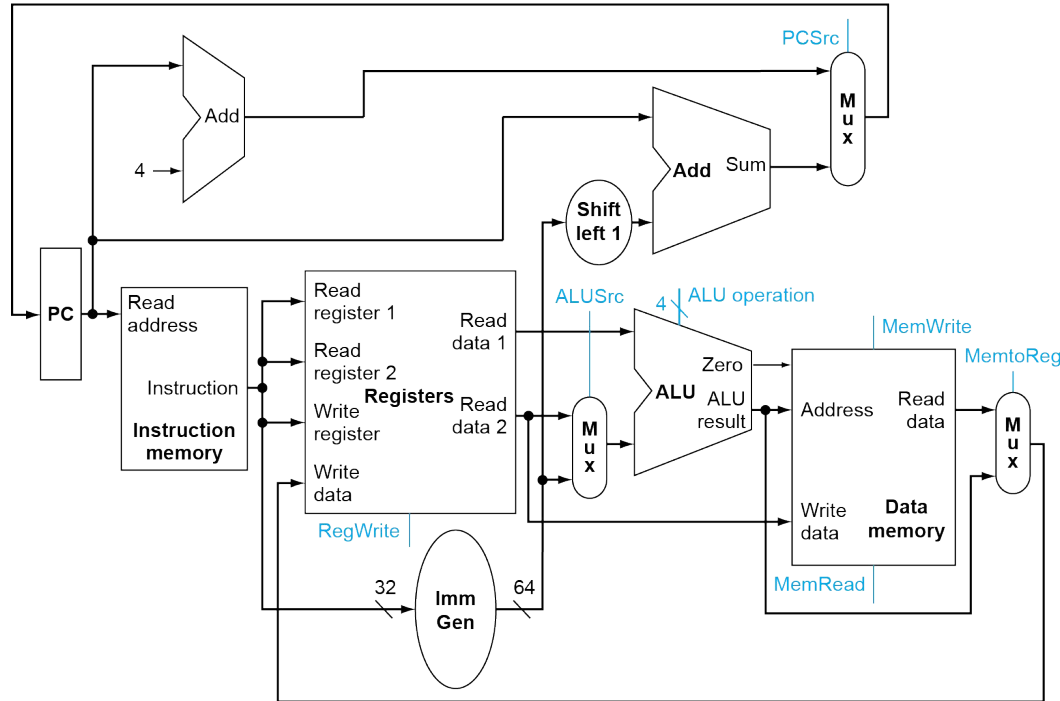
- don't-care terms
 - A don't-care term in the truth table indicates that the output does not depend on the value of that term.
 - In the following table, it is represented by an X in an input column)

ALUOp		Funct7 field							Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

The ALU Control



Designing the Main Control Unit



We need to identify:

The fields of an instruction and the control lines needed to complete this datapath

Designing the Main Control Unit

Reviewing the formats of the 4 instruction classes: **arithmetic, load, store, and conditional branch instructions.**

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

Designing the Main Control Unit

simplicity favors regularity

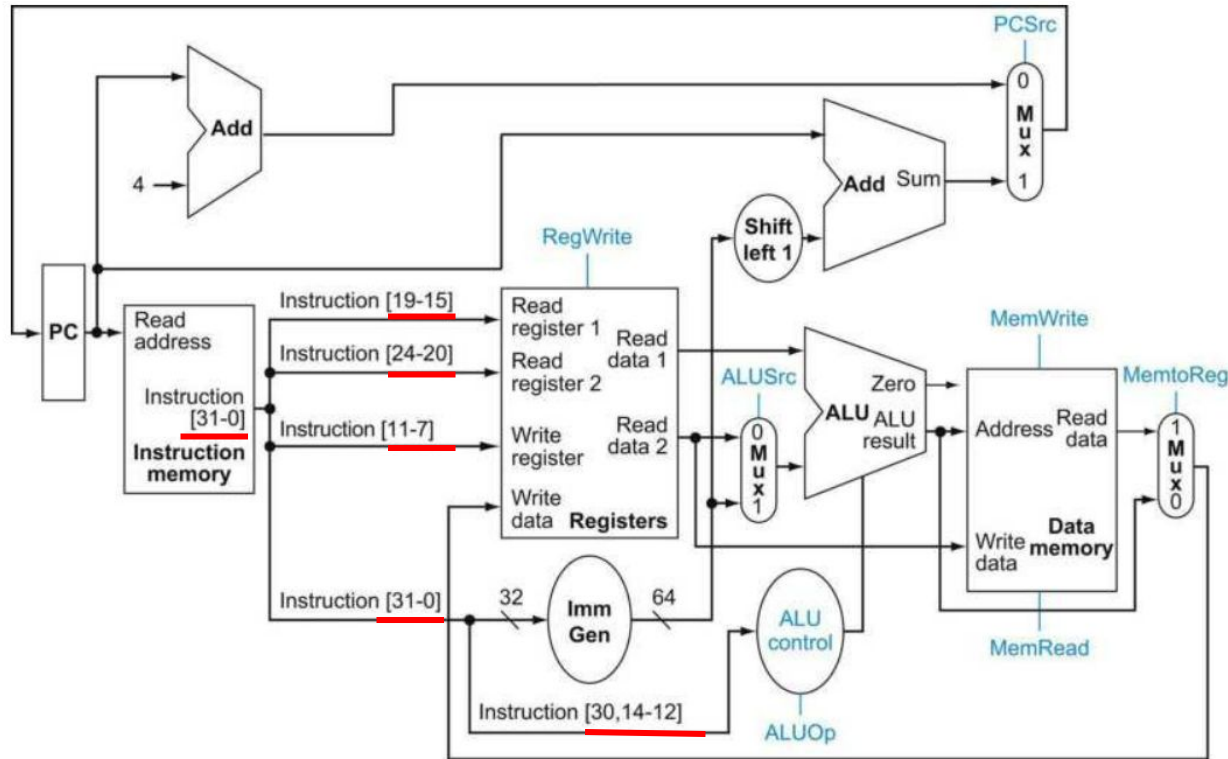
Identifying the fields:

- **Opcode:** bits 6:0. Also depending on this code
 - funct3: bits 14:12
 - funct7: bits 31:25
- **rd:** bits 11:7 in R-type and I-type
- **rs1:** bits 19:15
- **rs2:** bits 24:20 in R, S and SB type
- **Imm:** The immediate fields also follows convention.
 - Observe S and SB format imm field!

	Fields					
Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

Designing the Main Control Unit

simplicity favors regularity



- **Opcode:** bits 6:0.
- **rd:** bits **11:7**
- **rs1:** bits **19:15**
- **rs2:** bits **24:20**

Other Control Signals

There are 6 other control signals:

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Asserted

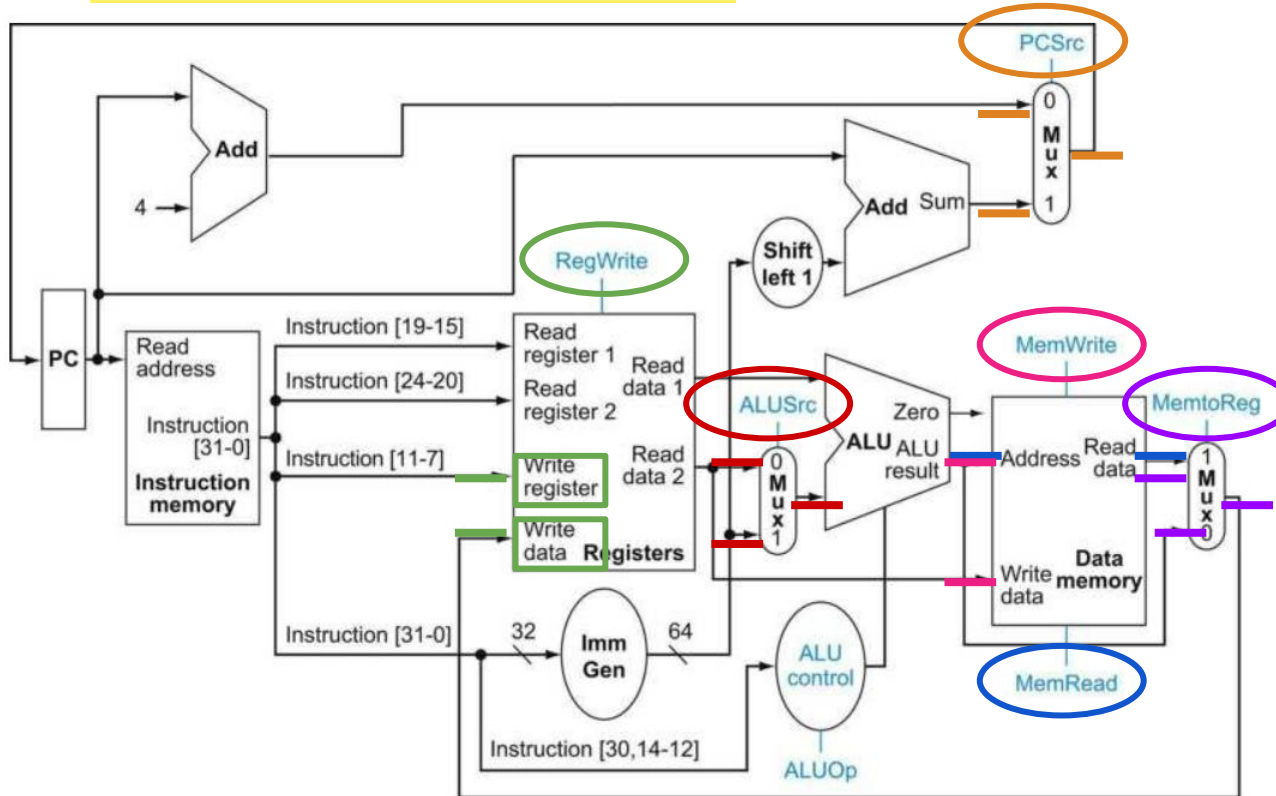
- signal value 1

Deasserted

- Signal value 0

Other Control Signals

There are 6 other control signals:



1. RegWrite

2. ALUSrc

3. PCSrc

4. MemRead

5. MemWrite

6. MemtoReg

Setting the control signal

- 5 out of the 6 signals can be set based on the opcode and funct fields.
 - PCSrc is the exception.
- **PCSrc**
 - Requires both a signal from control unit (determining it is a branch instruction) and Zero signal from ALU.
 - An **AND gate** is used to combine the **branch control signal** and the **Zero output** from the ALU.
 - So this is a derived signal!
- Control unit takes 7 bit opcode as input and generates the 8 control signals.
 - 6 signals – RegWrite, MemRead, MemWrite, MemtoReg, ALUSrc, PCSrc.
 - 2 signals – ALUOp.

CSE 4305: Lecture 4



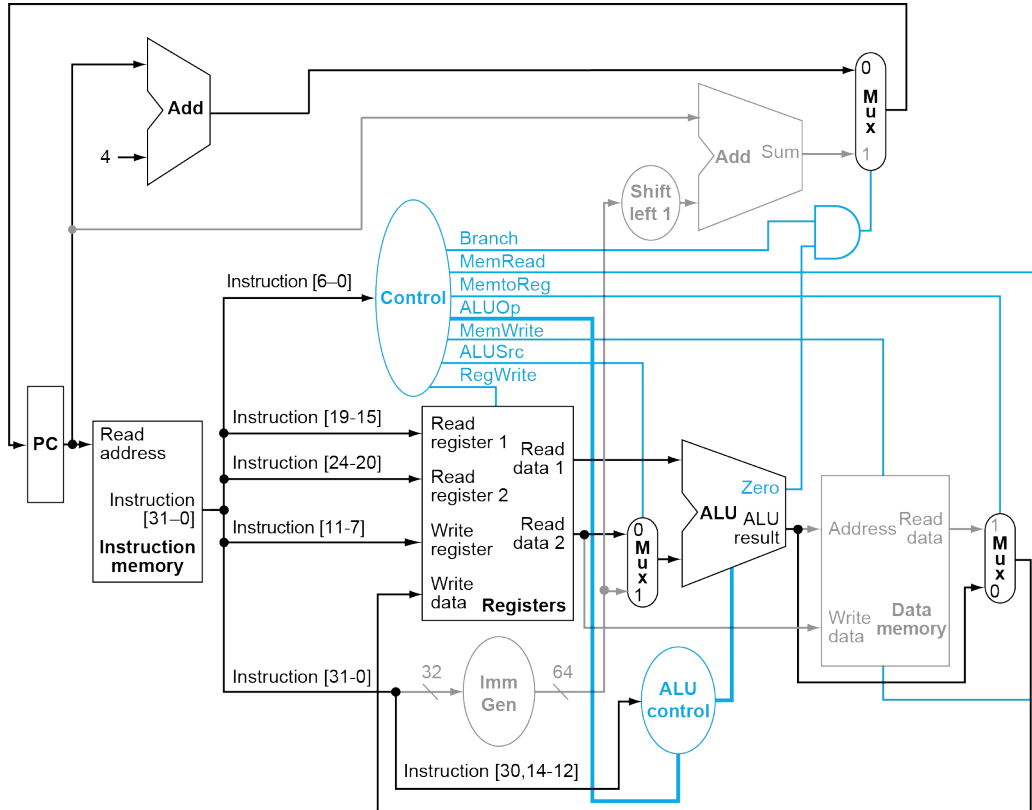
Setting the control signal

Based on the opcode:

- we define whether each control signal should be 0, 1, or don't care (X)

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

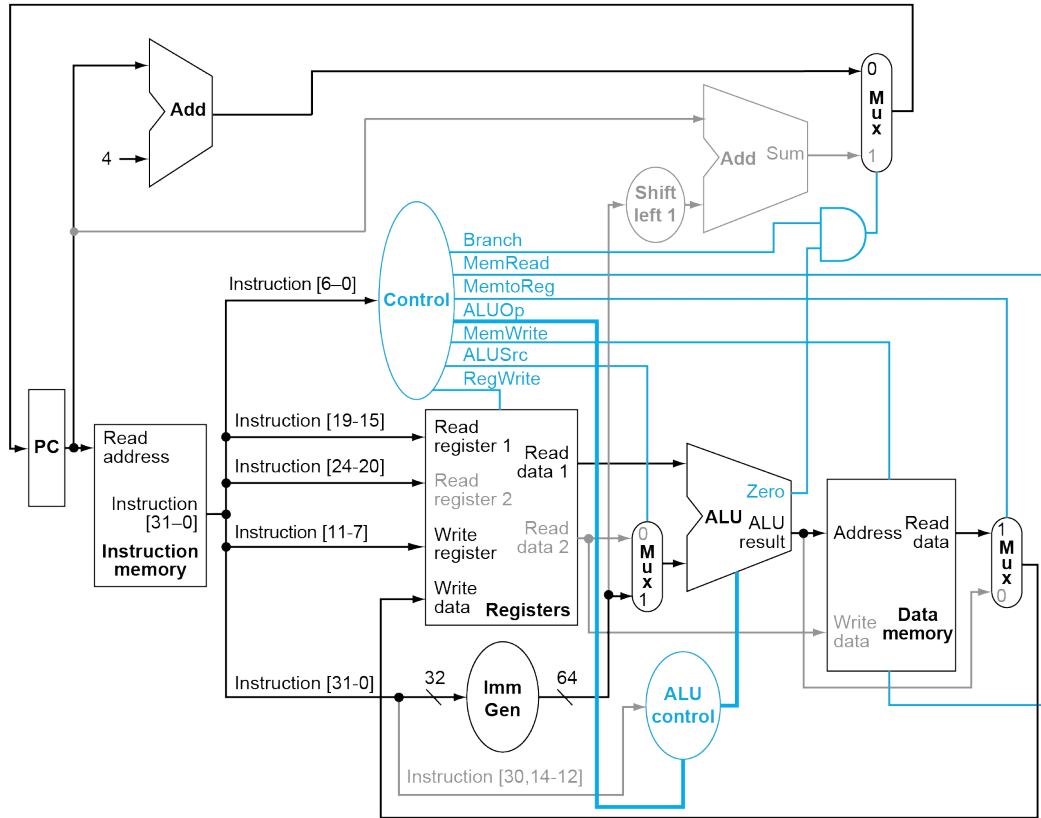
Active Datapath for R-type instructions



```
add x22, x23, x15
```

1. The instruction is fetched, and the PC is incremented.
2. Read 2 registers from the register file and set the control lines.
3. The ALU operates on the data.
4. The result from the ALU is written into the destination register.

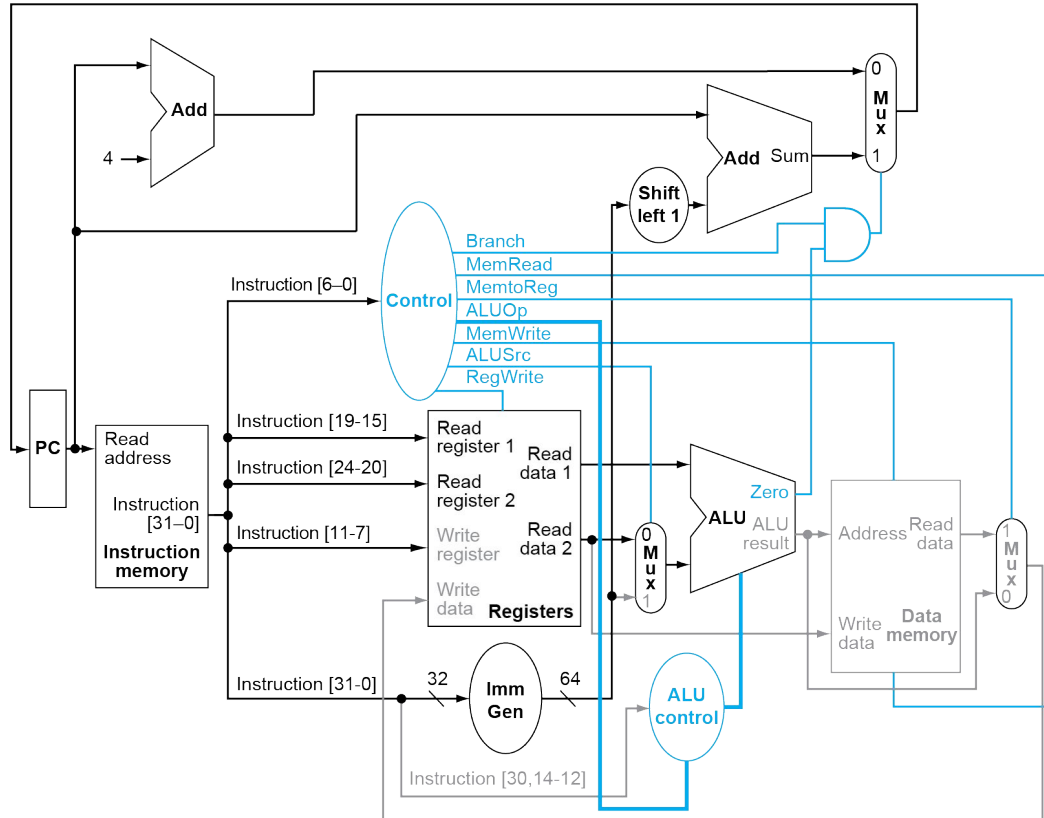
Active Datapath for Load instructions



1d x22, 8 (x23)

1. The instruction is fetched, and the PC is incremented.
2. Read 1 register value.
3. The ALU sums the register value and the sign-extended 12 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. data from the memory unit is written into the register file.

Active Datapath for Branch instructions



beq x22, x23, offset



Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Let's design the control logic!

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Finalizing the control

Input or output	Signal name	R-format	ld	sd	beq	
Inputs	I[6]	0	0	0	1	1
	I[5]	1	0	1	1	11
	I[4]	1	0	0	0	8
	I[3]	0	0	0	0	0
	I[2]	0	0	0	0	0
	I[1]	1	1	1	1	15
	I[0]	1	1	1	1	15
Outputs	ALUSrc	0	1	1	0	6
	MemtoReg	0	1	X	X	4xx
	RegWrite	1	1	0	0	12
	MemRead	0	1	0	0	4
	MemWrite	0	0	1	0	2
	Branch	0	0	0	1	1
	ALUOp1	1	0	0	0	8
	ALUOp0	0	0	0	1	1

The control function for the simple single-cycle implementation is completely specified by this truth table

Disadvantages of single cycle implementation

- Clock cycle must have the same length for every instruction.
- Clock cycle is determined by the longest possible path in the processor
 - Load instruction has the longest path.
 - Load: Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Acceptable for this small instruction set but **not efficient** for a complex instruction set.
- Performance is improved by **Pipelining!**

Reference:

1. Computer Architecture and Design RISC-V Edition, by David A Patterson and John L. Hennessy.
2. Documents from CS61C Su18, UC Berkeley.

Thank You!