Lecture 9:

Implementing a Processor 1/5

Introduction to Computer Architecture
UC Davis EEC 170, Fall 2019

Introduction

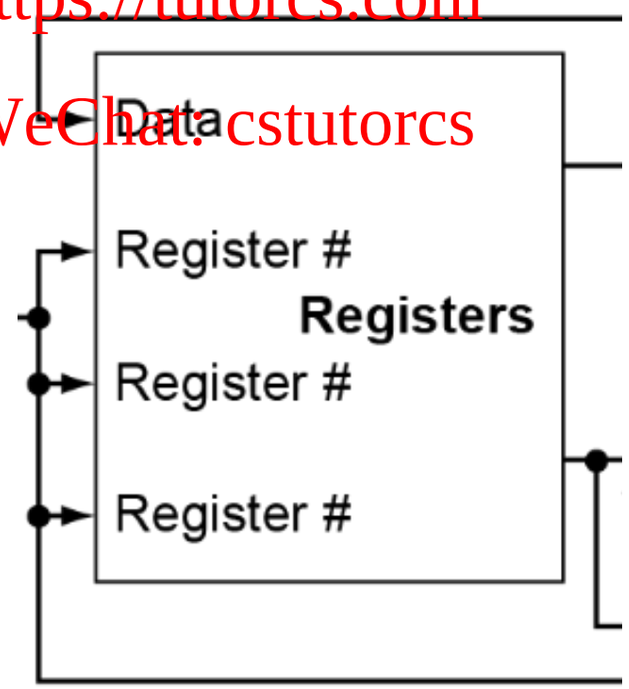
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two RISC-V implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: ld, sd
  - Arithmetic/logical: add, sub, and, or
  - Control transfer: beq

How to Design a Processor

- 1. Analyze instruction set => datapath requirements
  - the meaning of each instruction is given by the *register transfers* (what affects state?)
  - datapath must include storage element for ISA registers
    - possibly more
  - datapath must support each register transfer
- 2. Select set of datapath components and establish clocking methodology
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
- 5. Assemble the control logic

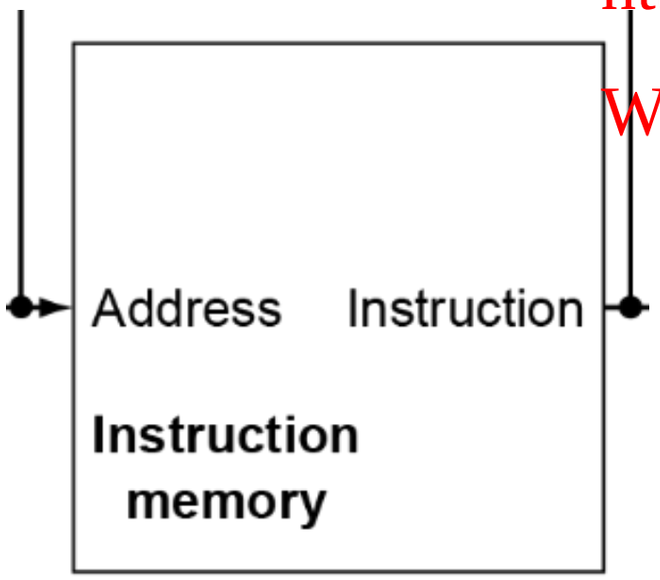
Ingredients: Register file

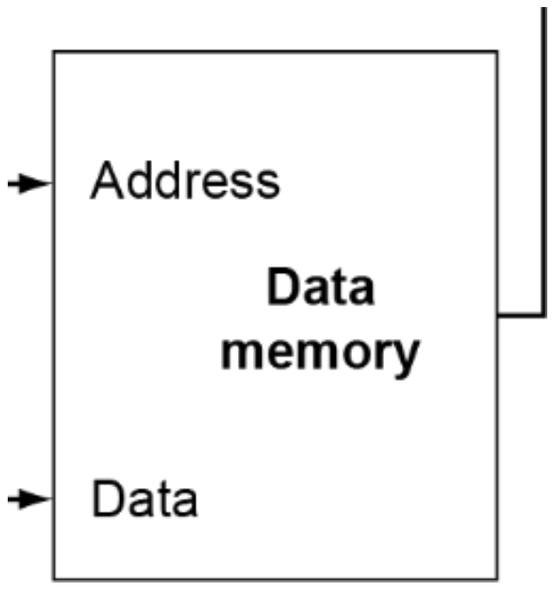
- How big is this in RISC-V?
- 1 "data port"
  - For what operations is this used?
- 3 "register port"s
  - For what operations are these used?
- Control (not shown)



Ingredients: Memories

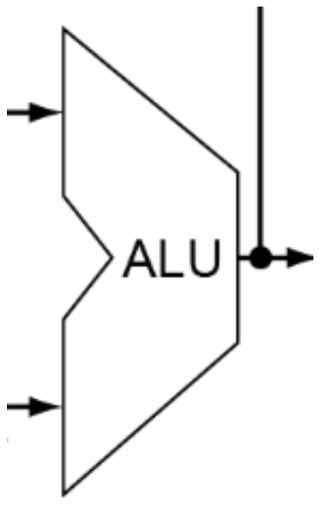
- What is the instruction memory used for?
- What is the data memory used for?
- Why doesn't the instruction memory have a data port?
- Why do we have two different memories?



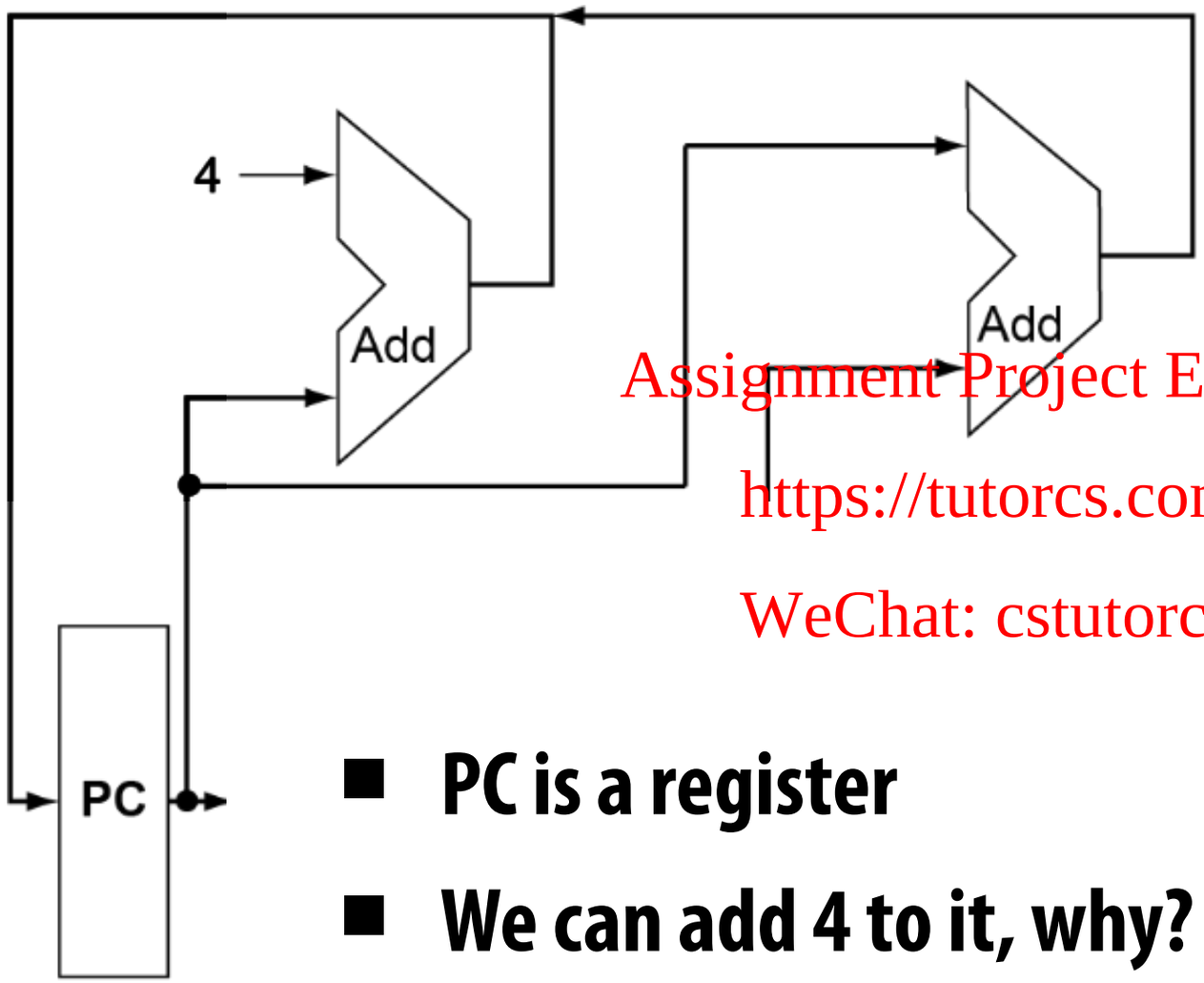


Ingredients: ALU

- This is the same ALU we designed last lecture
- 2 inputs, 1 output
  - Do all RISC-V instructions fit into 2 inputs, 1 output?

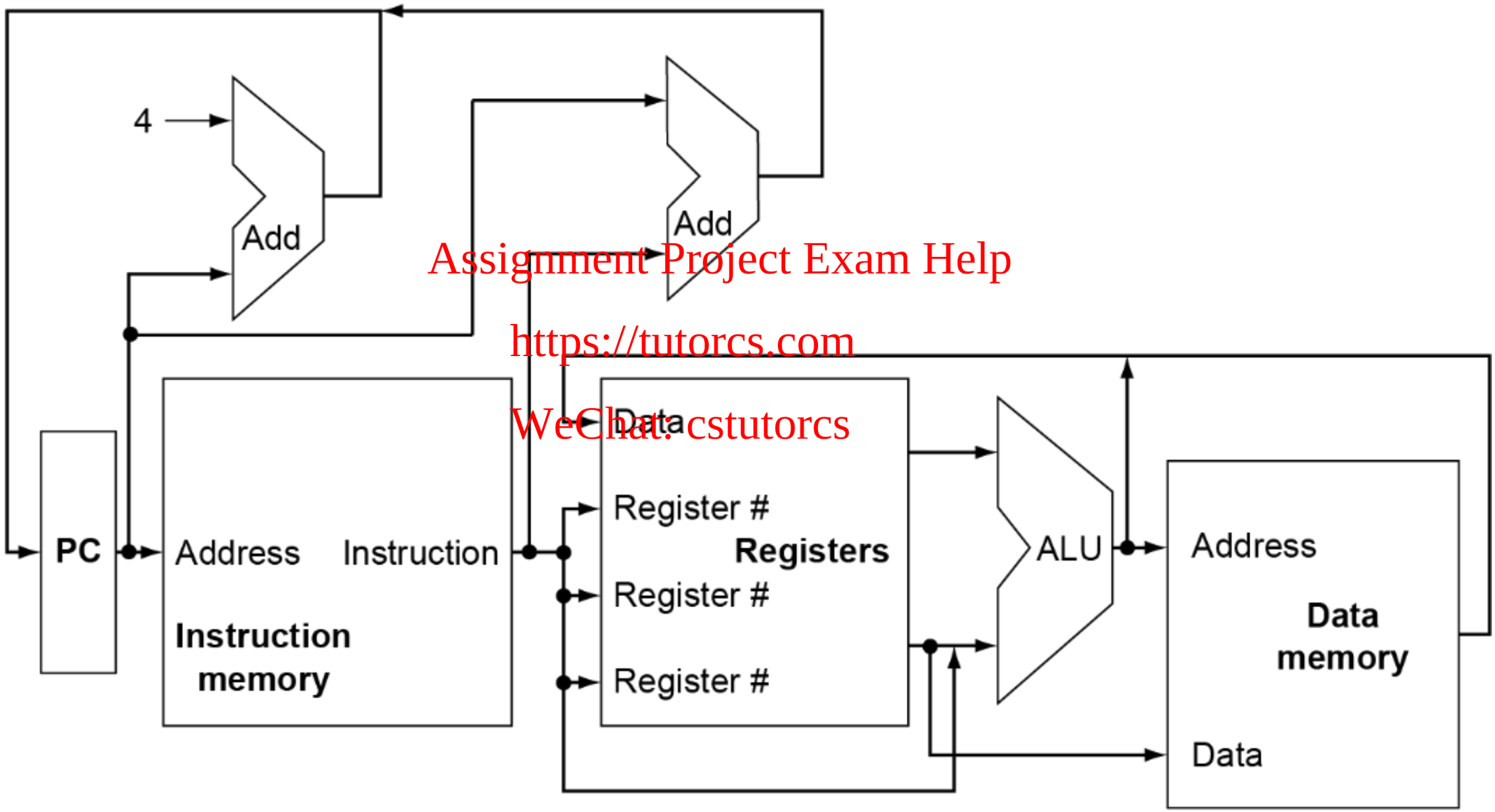


Ingredients: PC and adders



- PC is a register
- We can add 4 to it, why?
- We can add something that's not 4 to it, why?

CPU Overview



Register Transfers

- All start by fetching the instruction

  op | rs1 | rs2 | rd | funct{3,7} <= MEM[PC] # PC is Program Counter

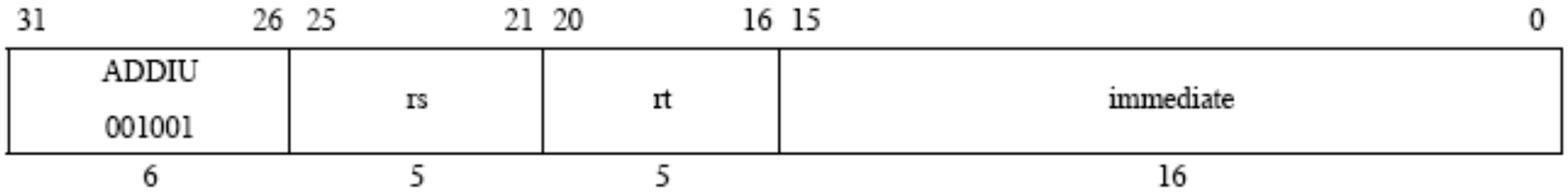
  op | rs1 | rd | imm <= MEM[PC]

- inst HDL description

inst	HDL description	
ADD	R[rd] <= R[rs1] + R[rs2];	PC <= PC + 4
SUB	R[rd] <= R[rs1] – R[rs2];	PC <= PC + 4
OR	R[rt] <= R[rs1] | R[rs2];	PC <= PC + 4
AND	R[rt] <= R[rs1] & R[rs2];	PC <= PC + 4
LOAD	R[rt] <= MEM[R[rs1] + imm];	PC <= PC + 4
STORE	MEM[R[rs1] + imm] <= R[rs2];	PC <= PC + 4
BEQ	if (R[rs1] == R[rs2])	PC <= PC + 4 + {imm << 1)
	else	PC <= PC + 4

ADDIU in MIPS manual

Add Immediate Unsigned Word **ADDIU**



Format: `ADDIU rt, rs, immediate` **MIPS32**

Purpose:

To add a constant to a 32-bit integer

Description: `GPR[rt] ← GPR[rs] + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

Exceptions:

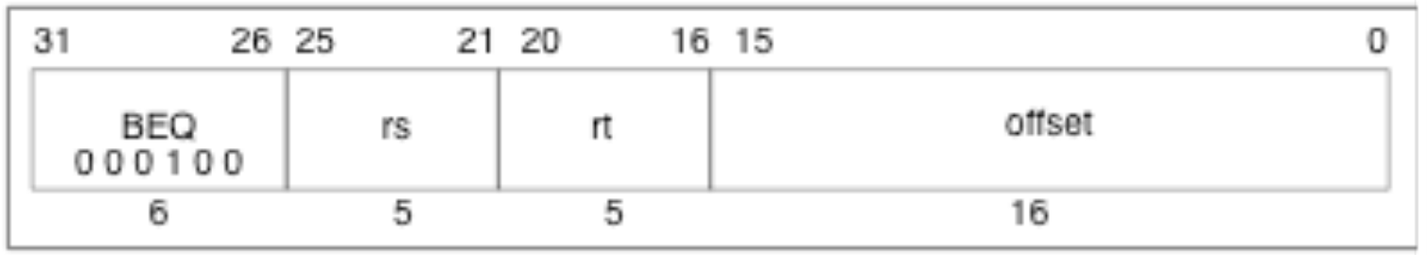
None

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

BEQ in MIPS manual

BEQ **Branch On Equal** **BEQ**



Format:

BEQ rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

Operation:

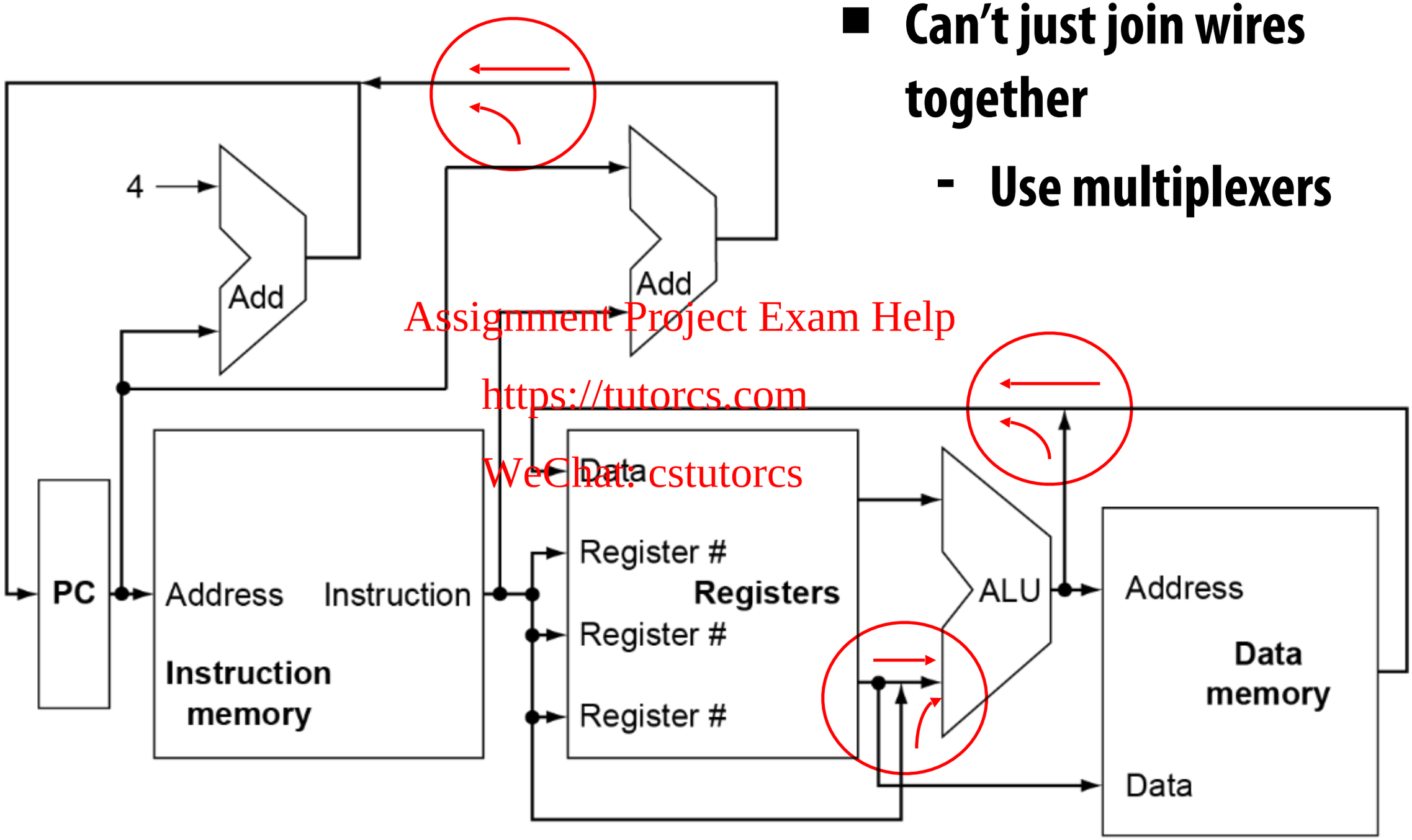
```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                        PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                        PC ← PC + target
            endif
```

Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch comparison
  - Access data memory for load/store
  - PC ← target address or PC + 4

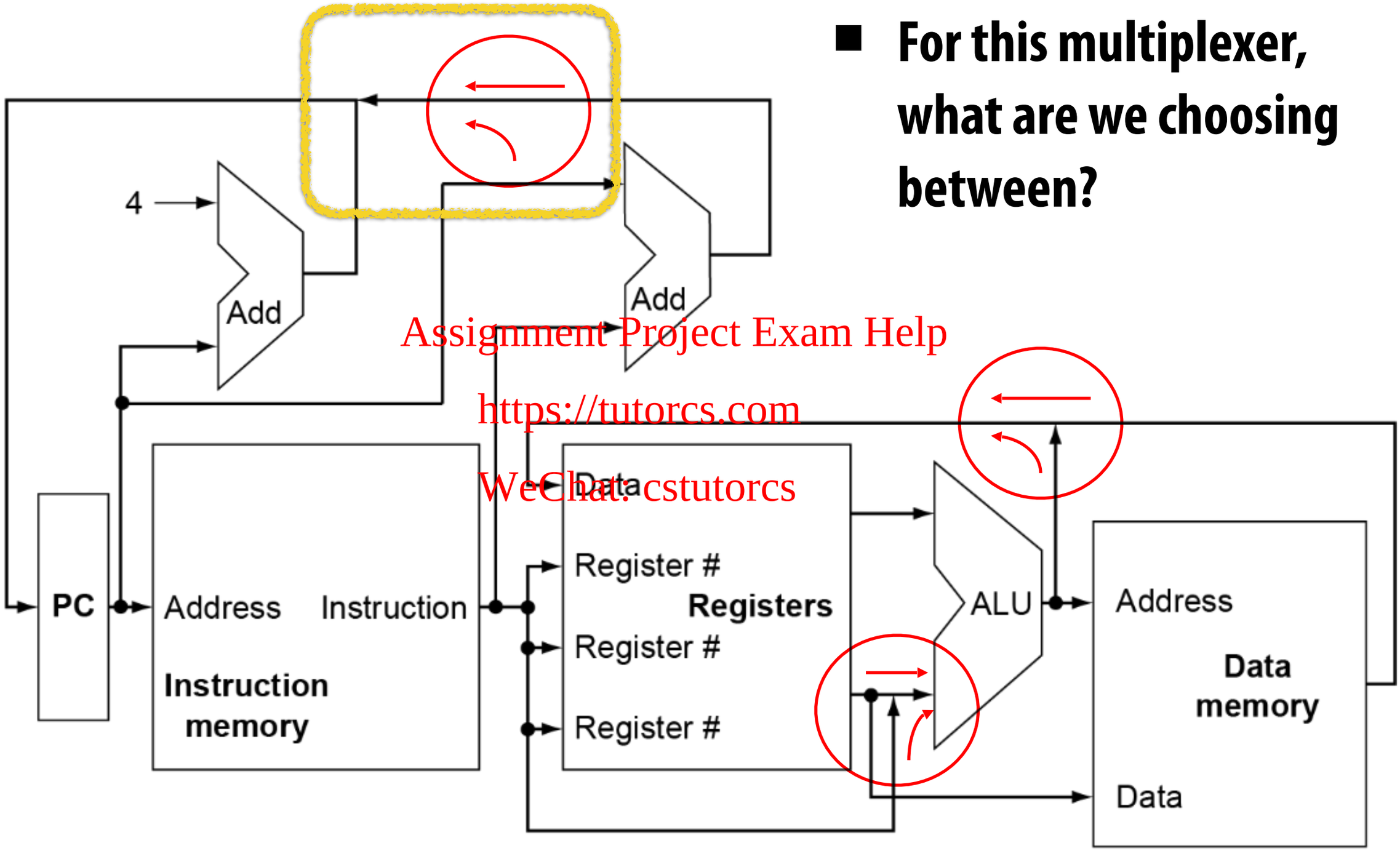
Multiplexers

- Can't just join wires together
  - Use multiplexers



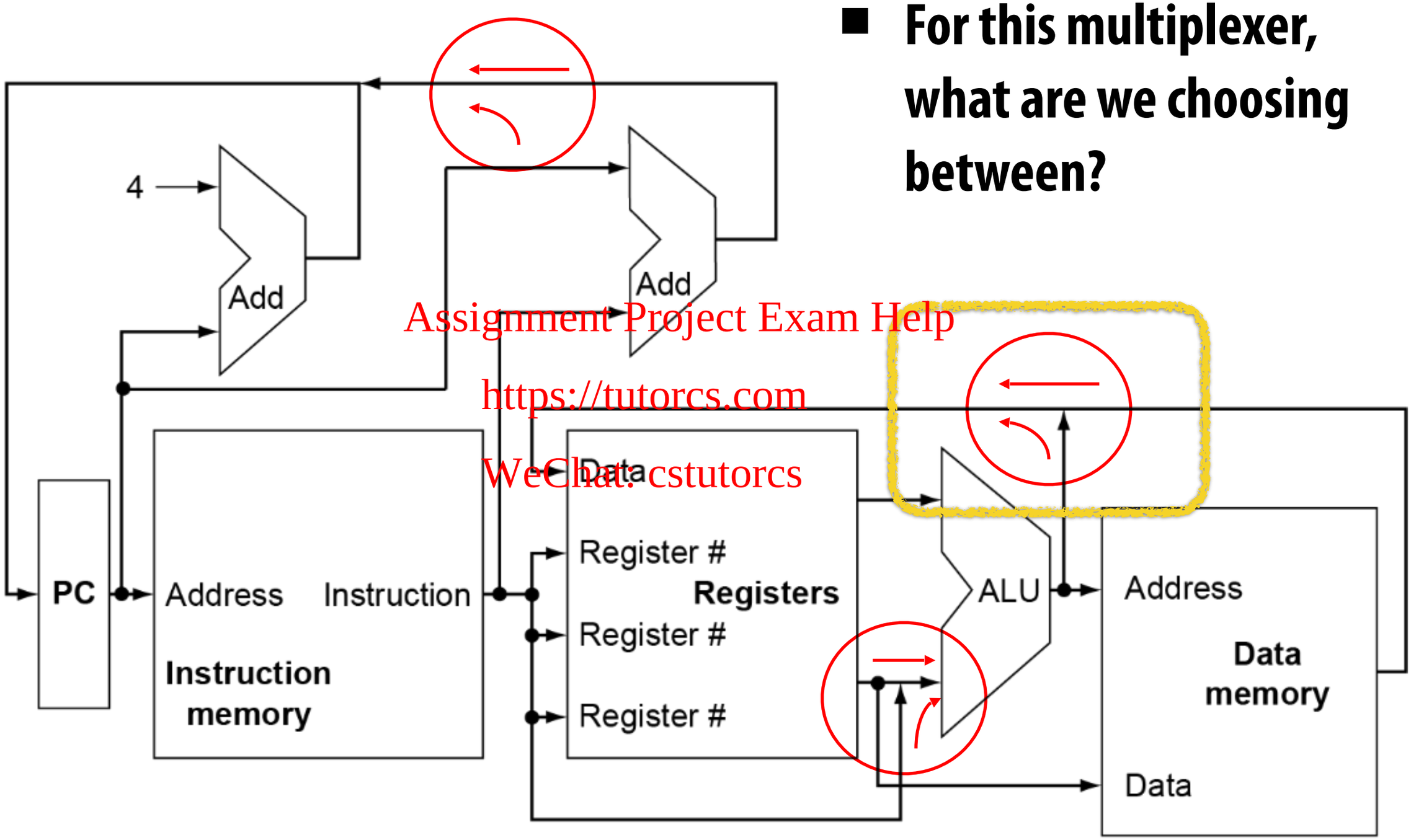
Multiplexers (1/3)

- For this multiplexer, what are we choosing between?



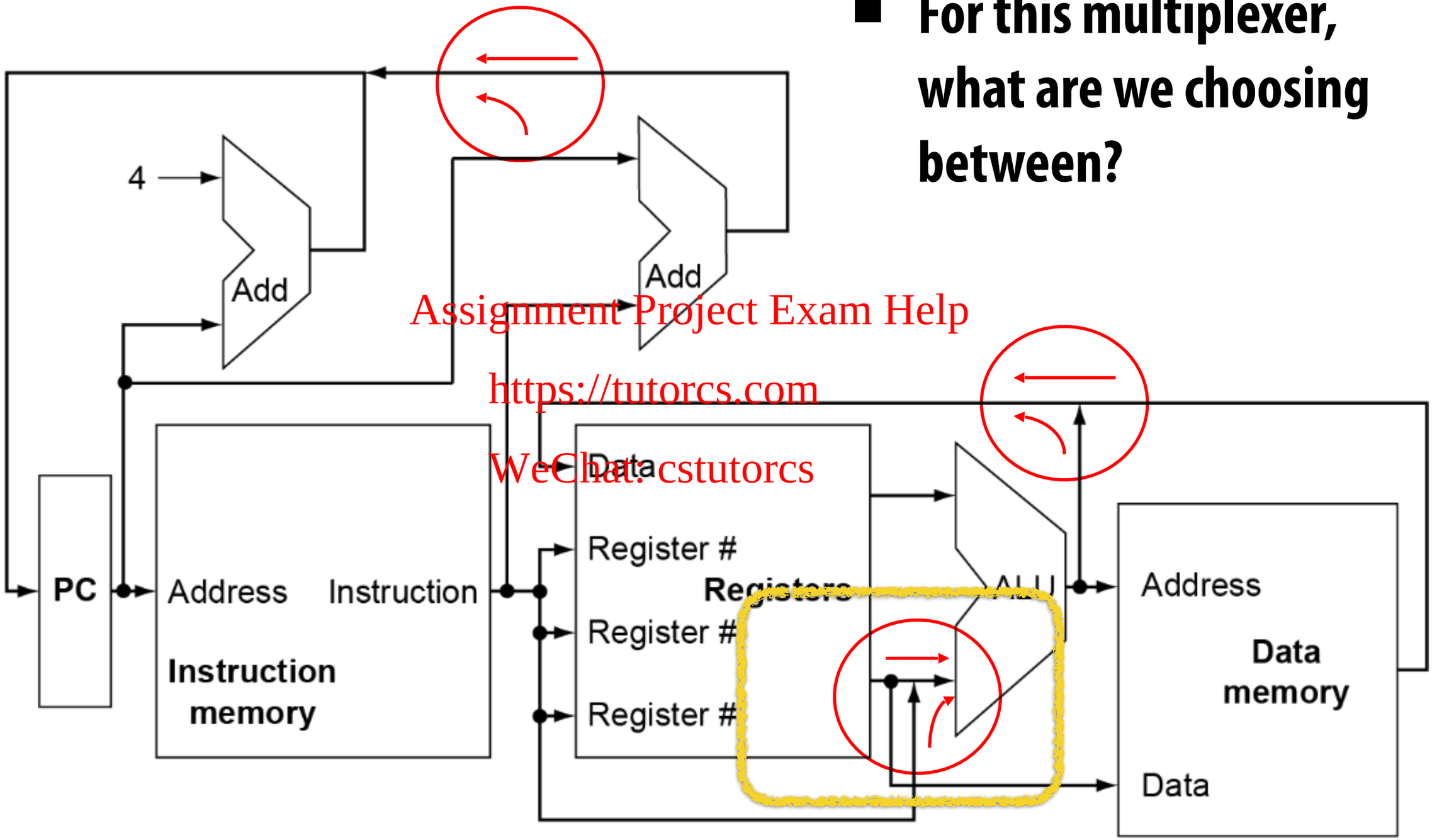
Multiplexers (2/3)

- For this multiplexer, what are we choosing between?

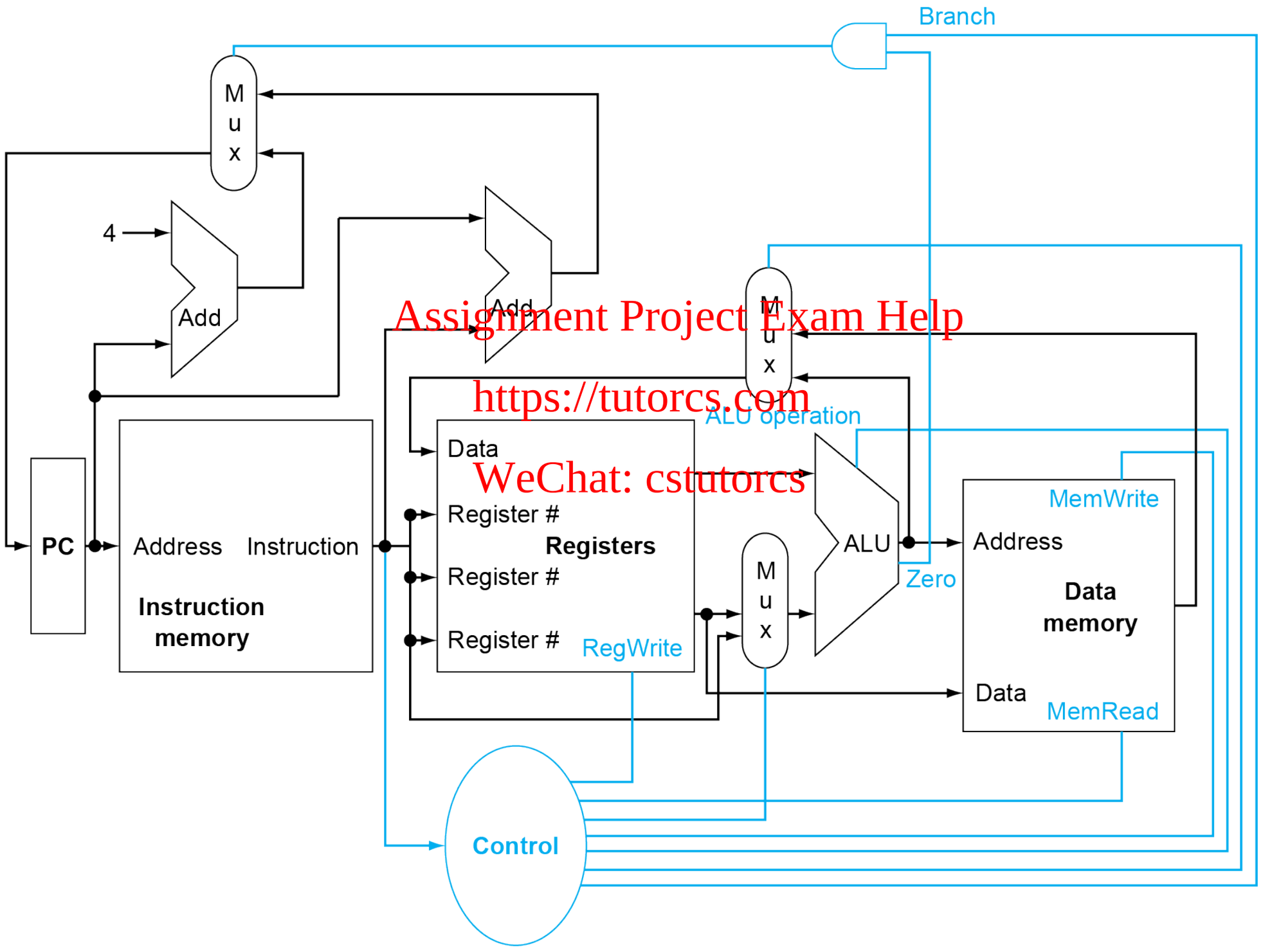


Multiplexers (3/3)

- For this multiplexer, what are we choosing between?



Control

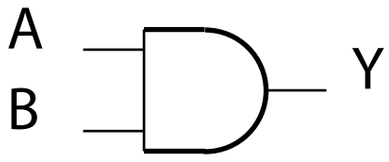


Logic Design Basics

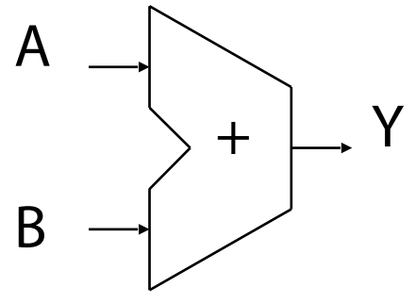
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

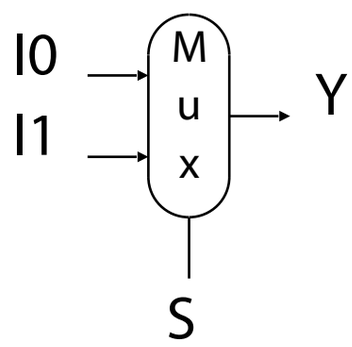
- AND-gate
  - Y = A & B



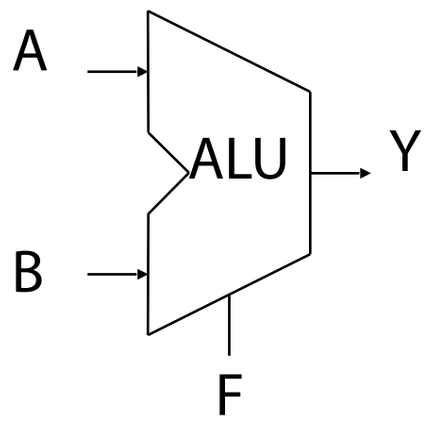
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

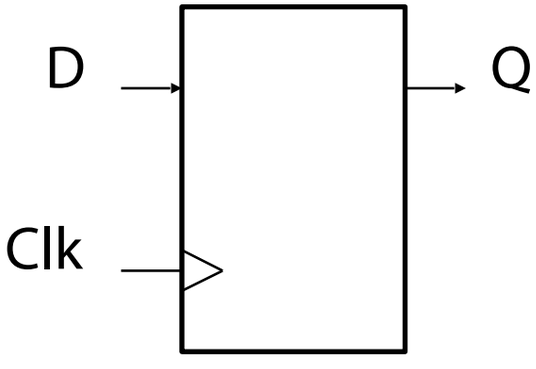


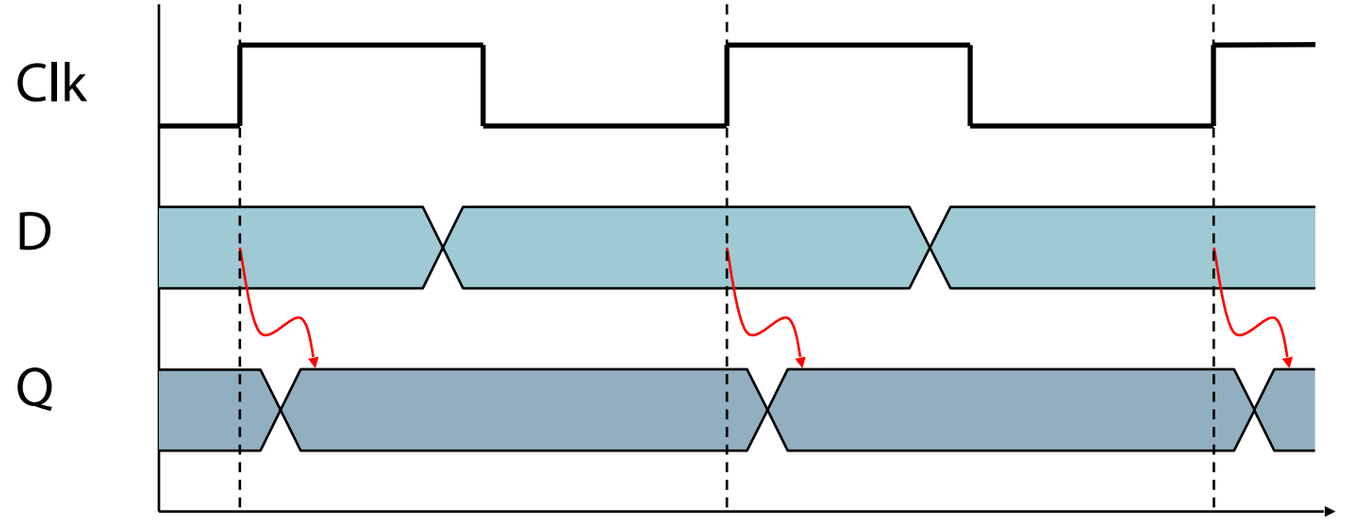
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

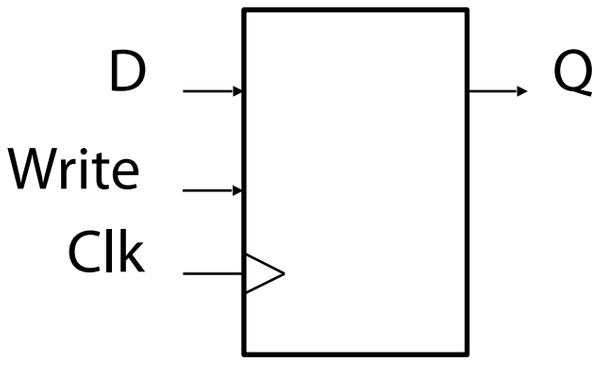
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1 (or 1 to 0)

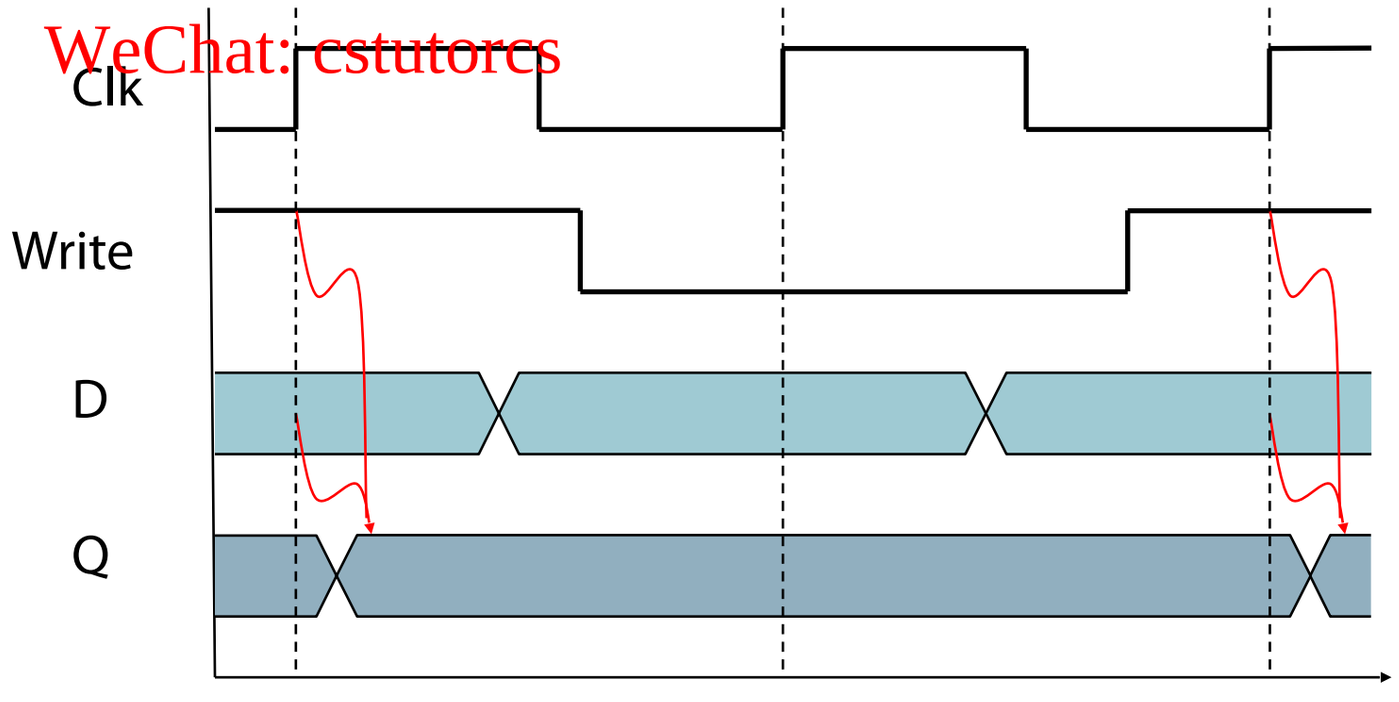




Sequential Elements

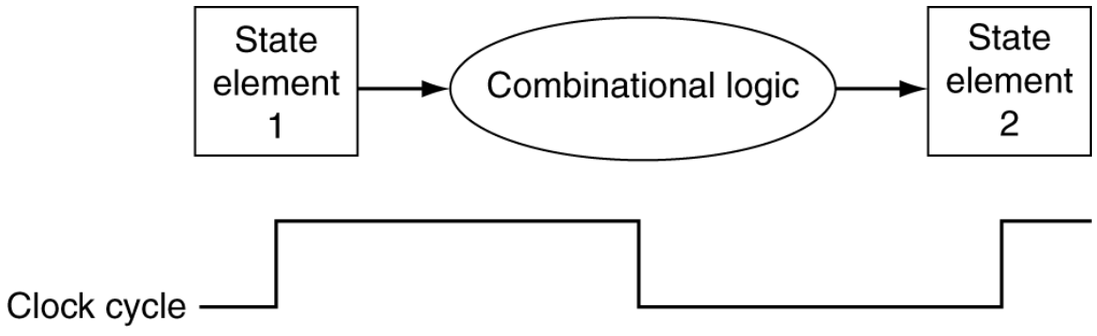
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

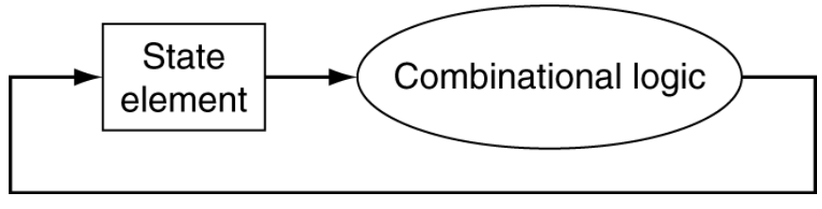




Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
    - If you care about a signal, it better be stored in a state element at the end of a clock cycle
  - Longest delay determines clock period

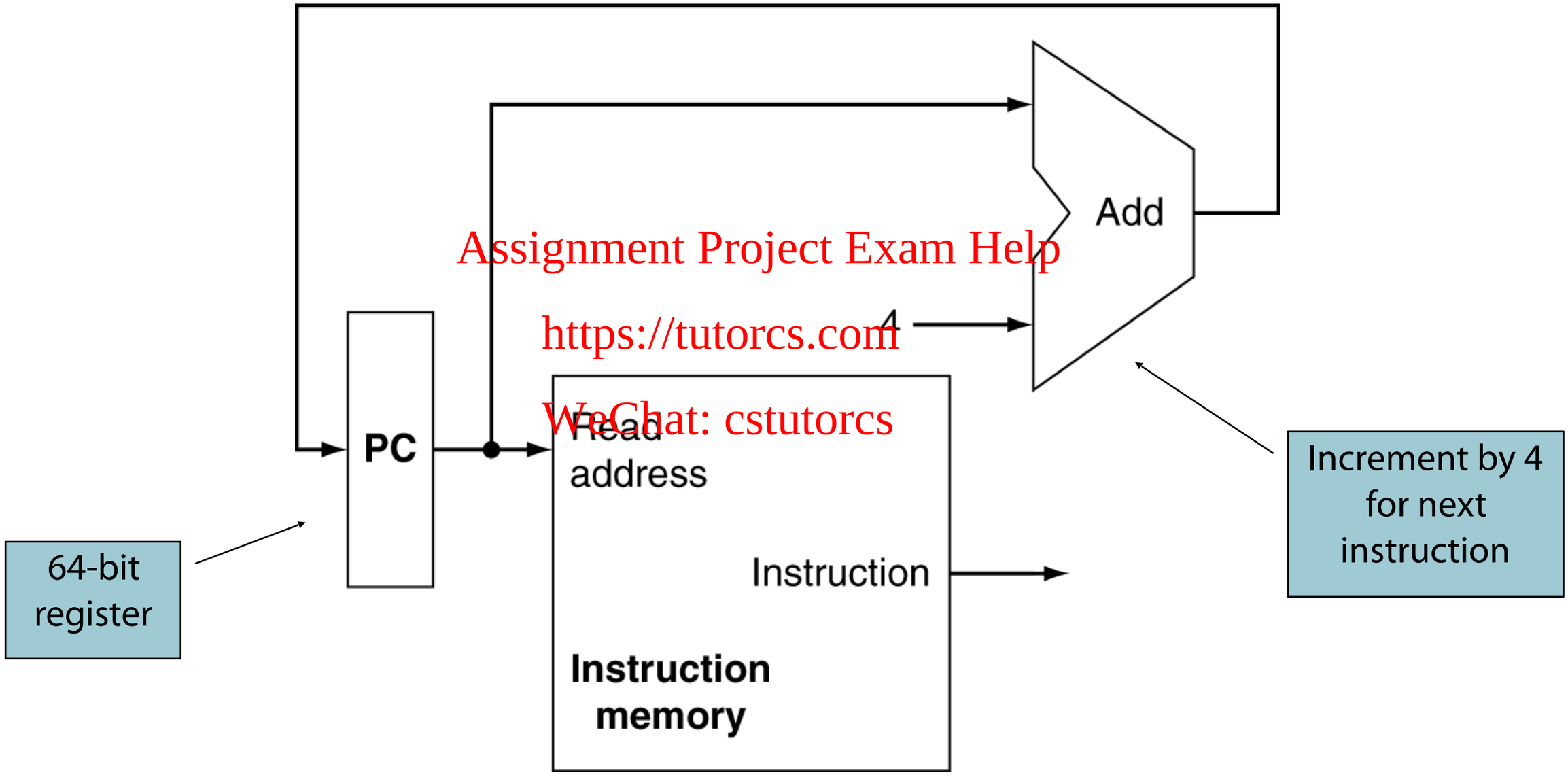




Building a Datapath

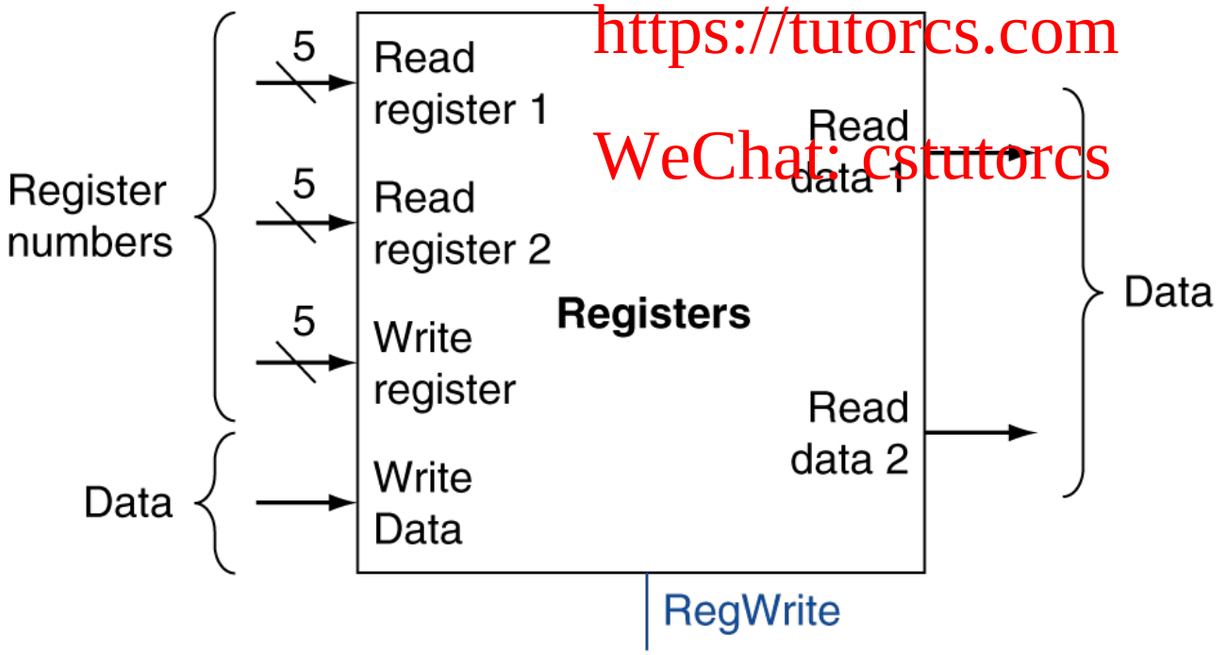
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, muxes, memories, …
- We will build a RISC-V datapath incrementally
  - Refining the overview design

Instruction Fetch

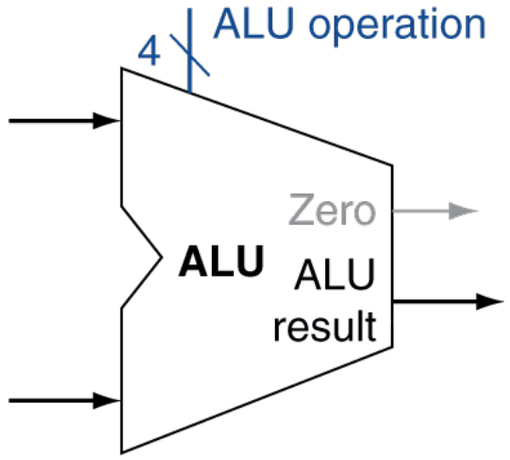


R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers

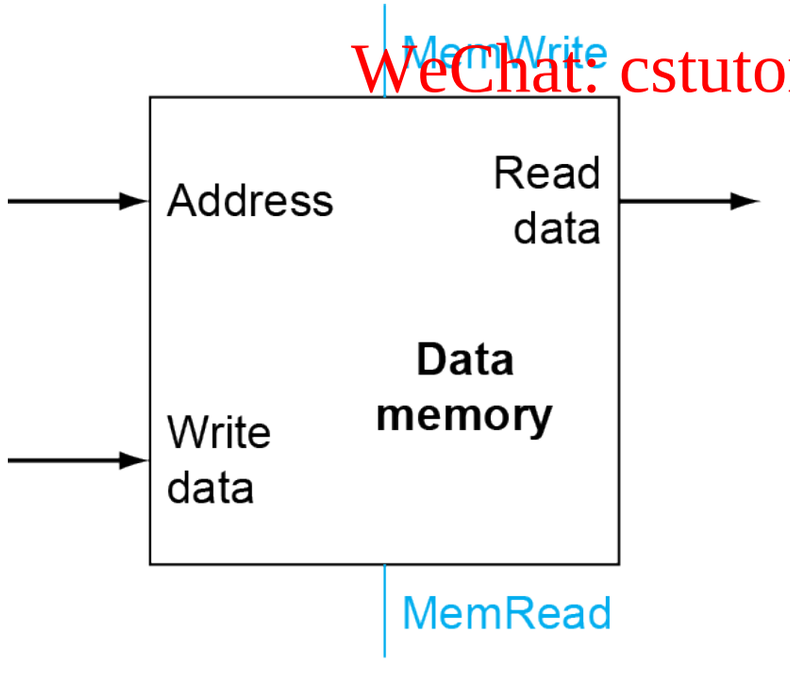


b. ALU

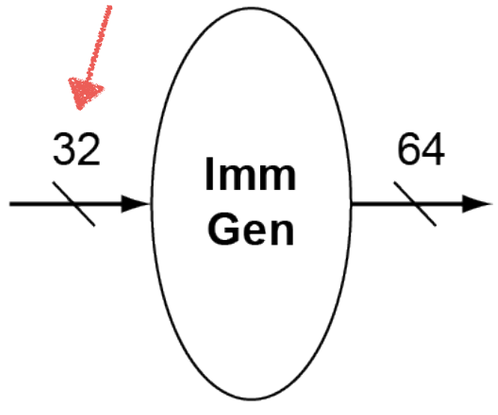
Load/Store Instructions

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

Why 32? Input here is an entire instruction.



a. Data memory unit

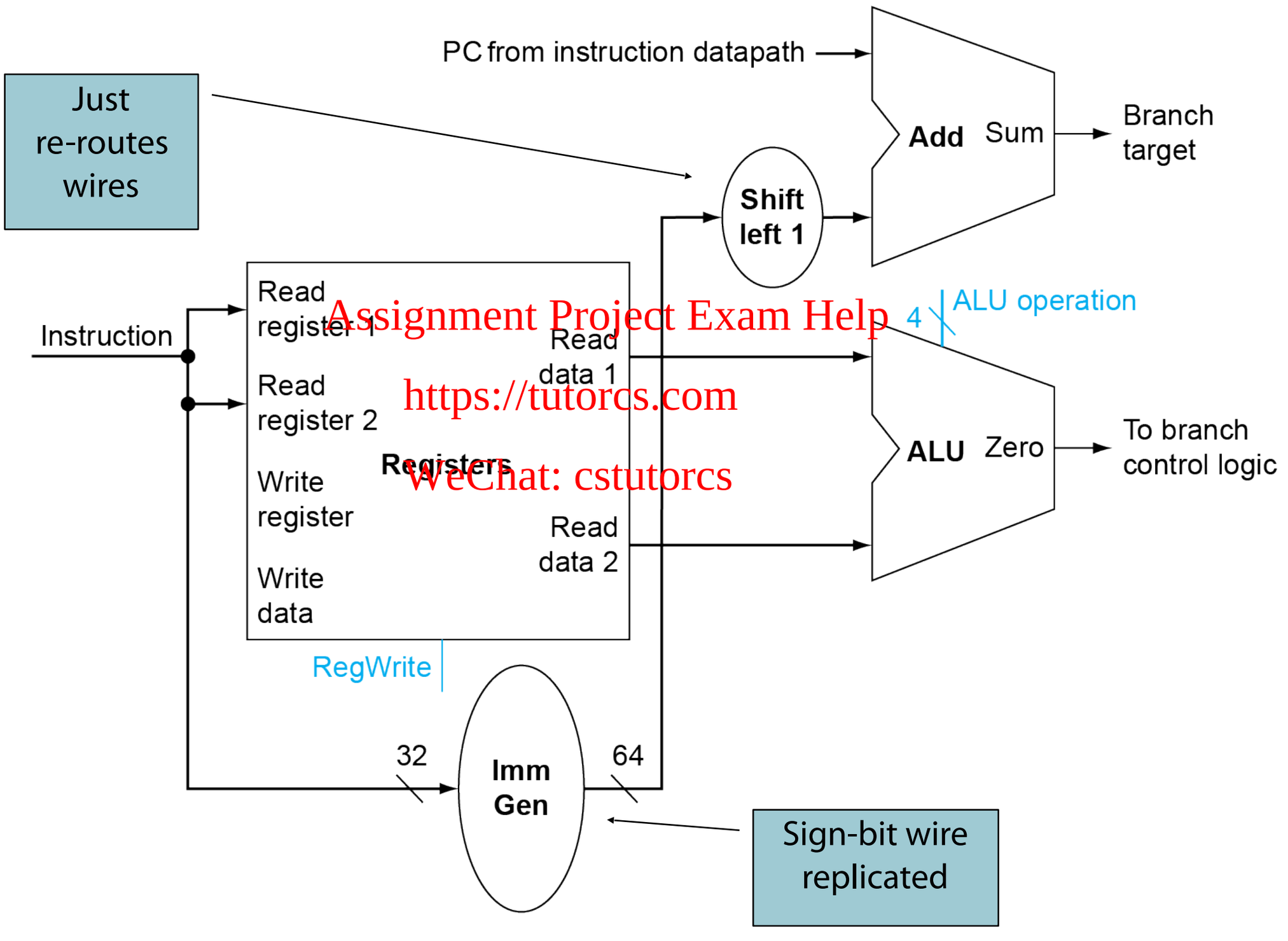


b. Immediate generation unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

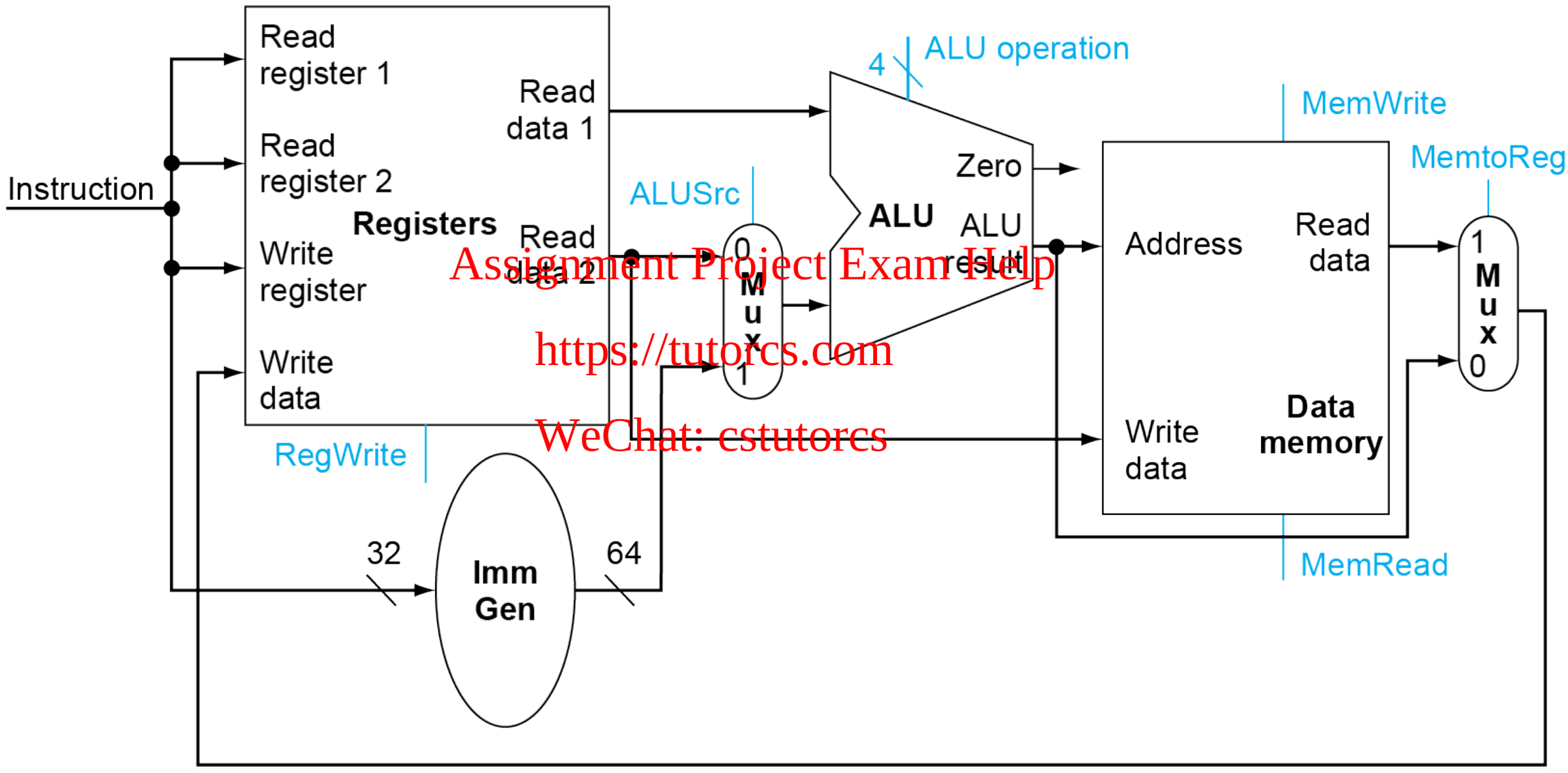
Branch Instructions



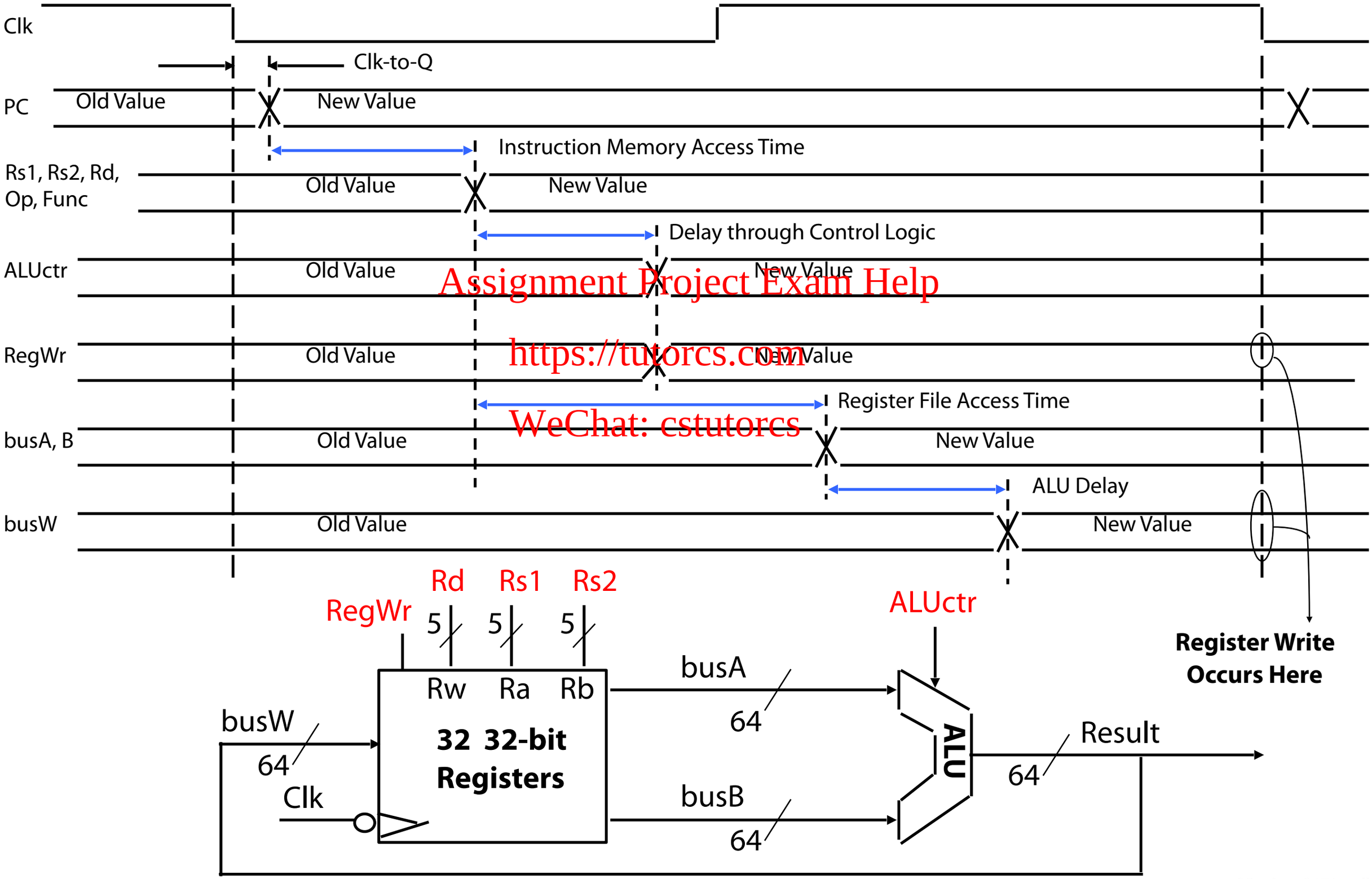
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



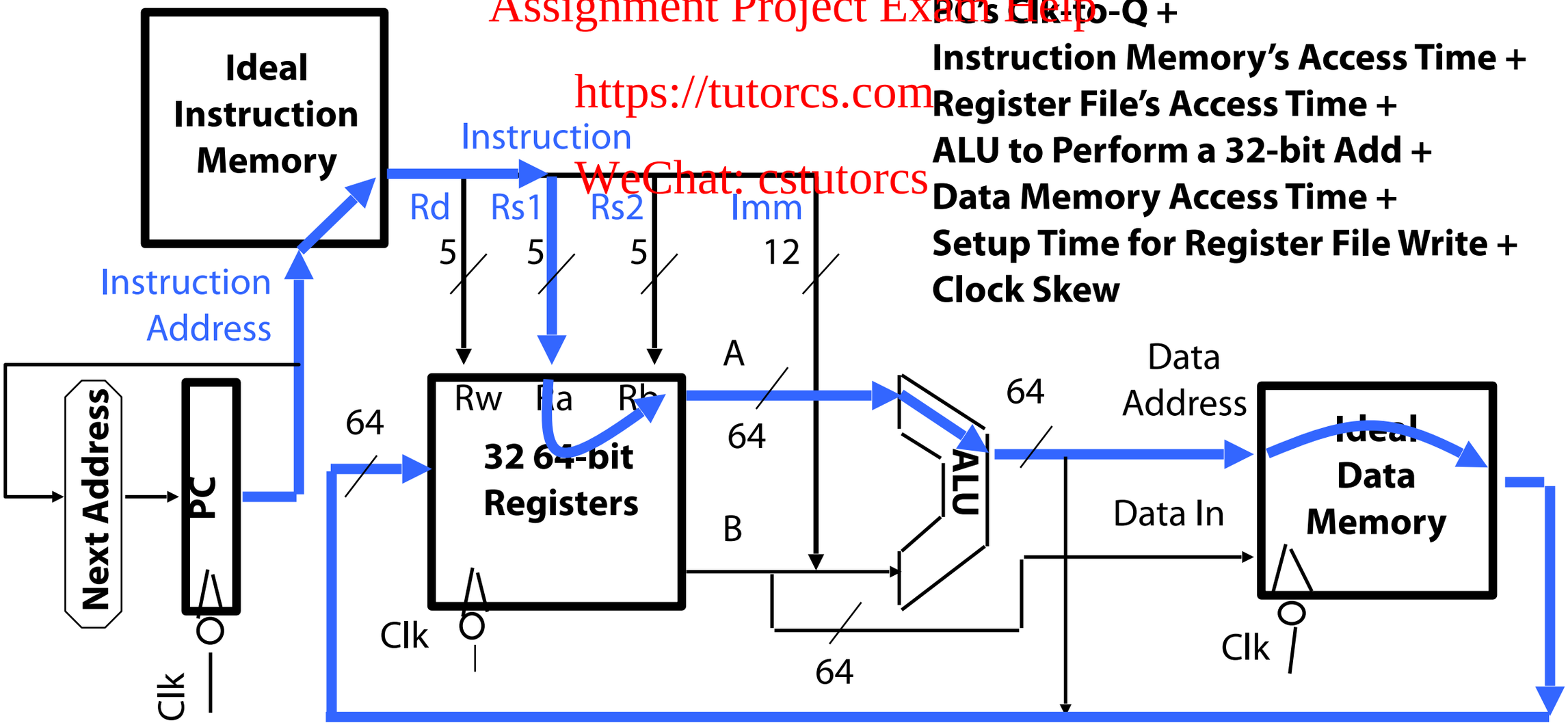
Register-Register Timing: One complete cycle



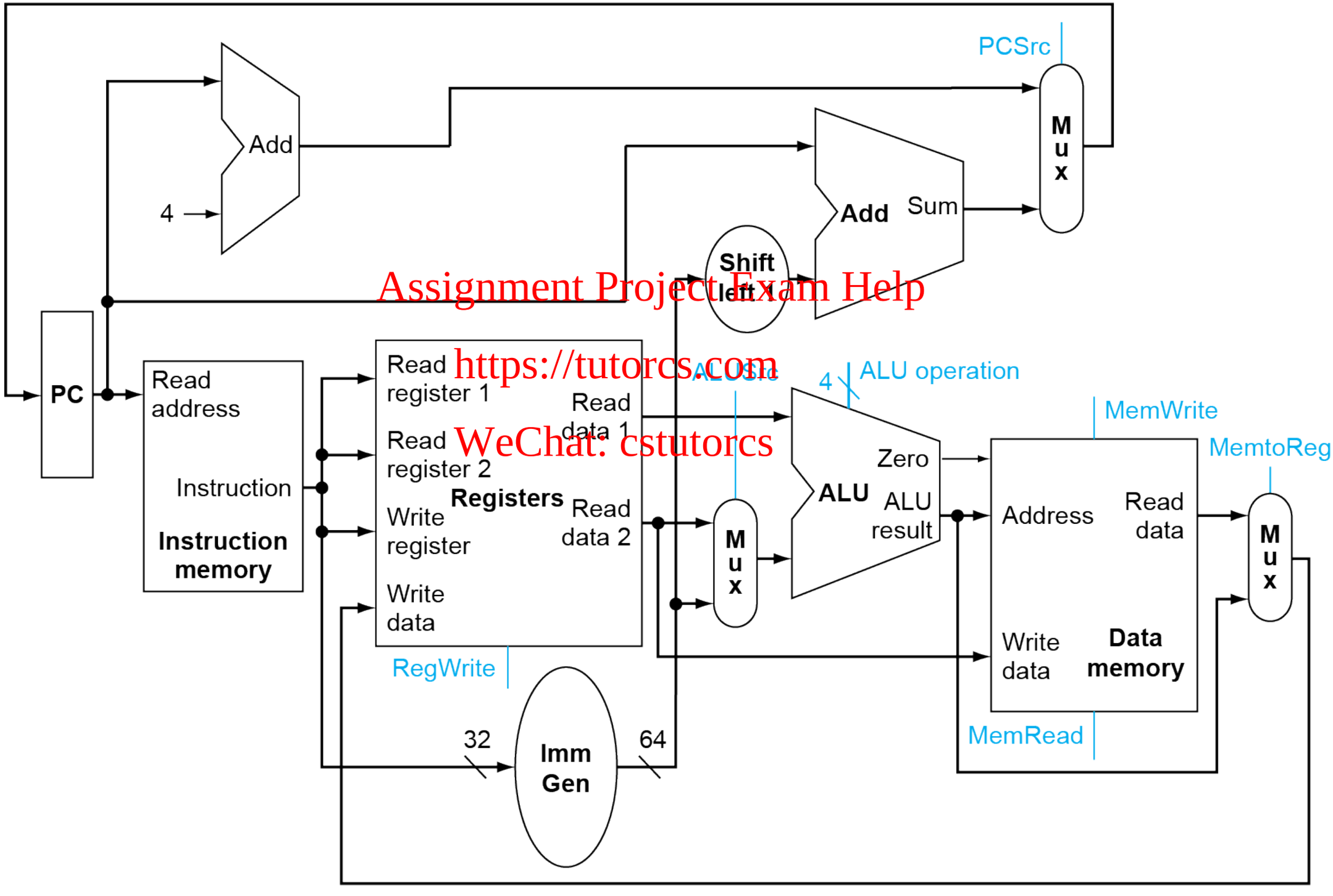
An Abstract View of the Critical Path

- Ideal memory AND register file:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after "access time."

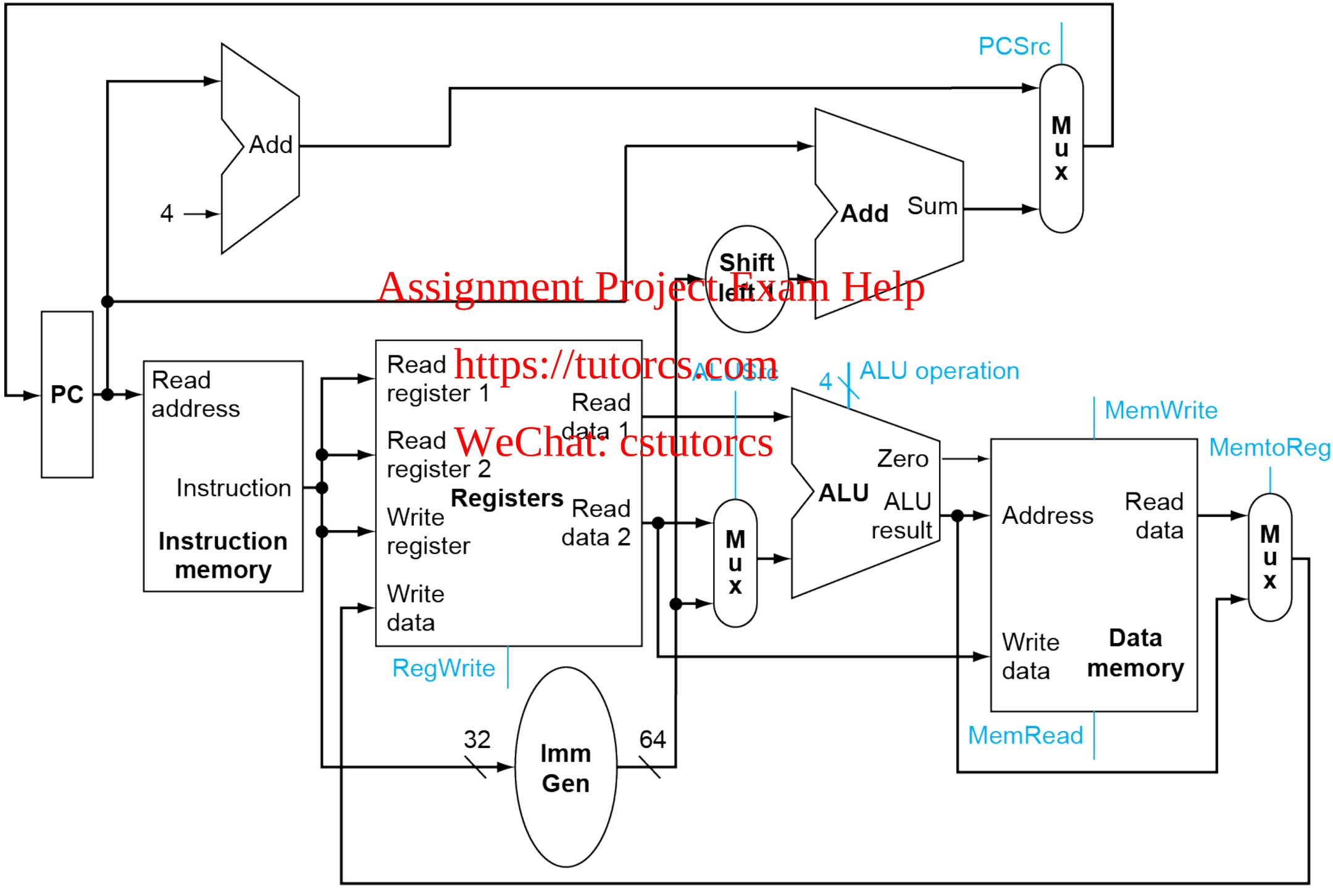
Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction Memory's Access Time +
Register File's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew



Full Datapath



Full Datapath



ALU Control

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

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

ALU Control

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

opcode	ALUOp	Operation	funct7/3 fields	ALU function	ALU control
ld	00	load word	XXXXXXXXXXXX	add	0010
sd	00	store dword	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	0000000/000	add	0010
		subtract	0100000/000	subtract	0110
		AND	0000000/111	AND	0000
		OR	0000000/110	OR	0001

The Main Control Unit

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

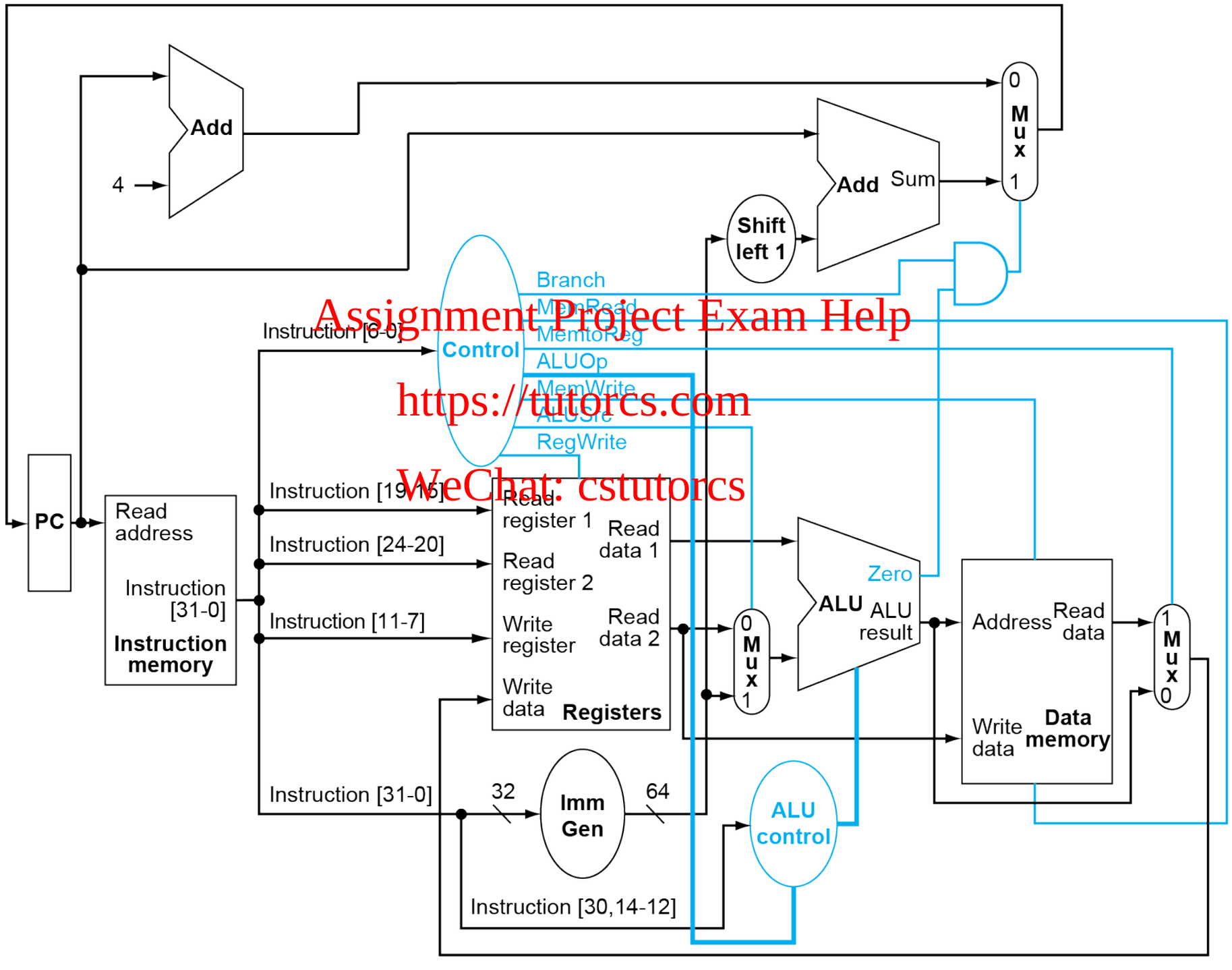
- Control signals derived from instruction

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

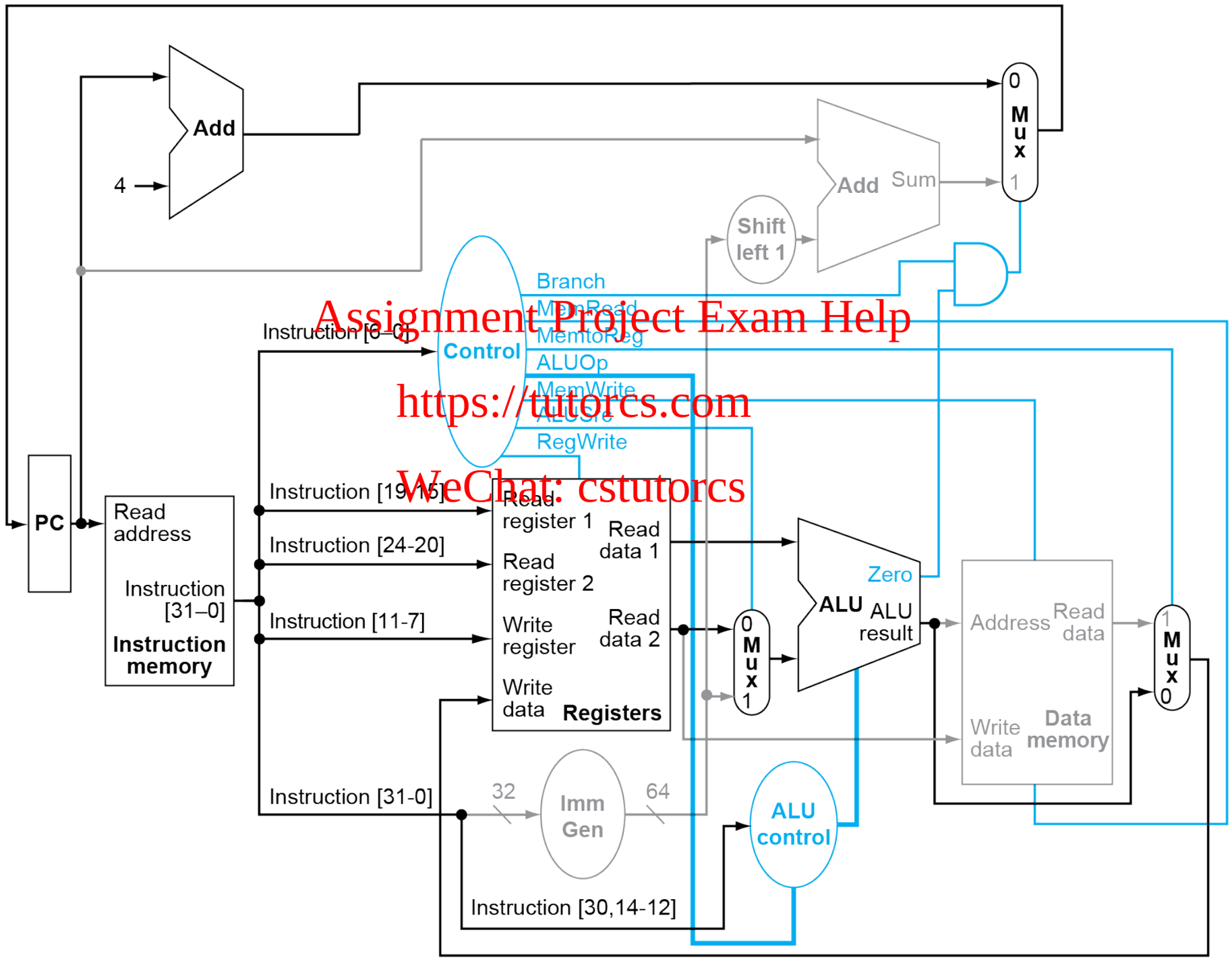
ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

these four bits are the only ones that matter for generating "operation"

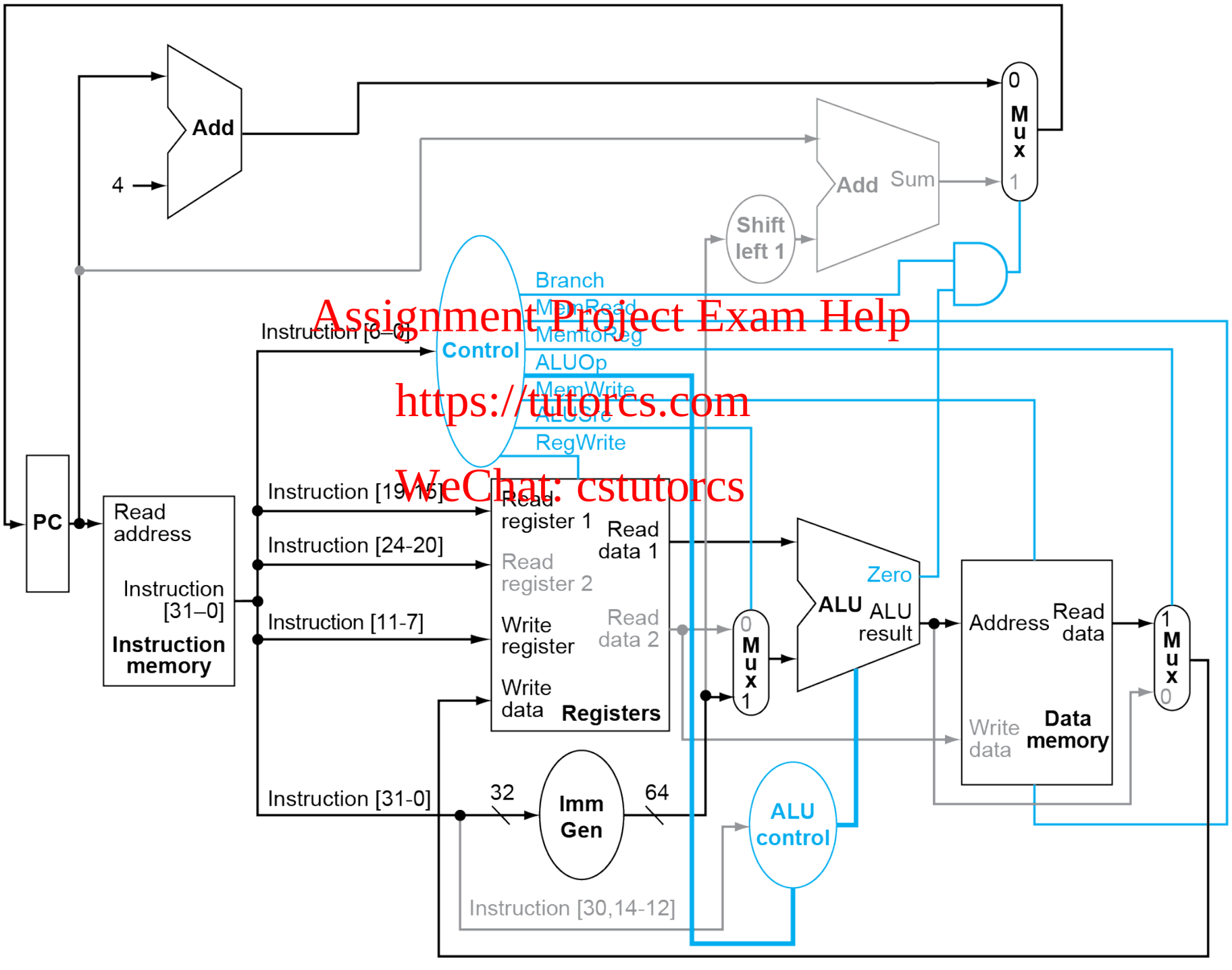
Datapath With Control



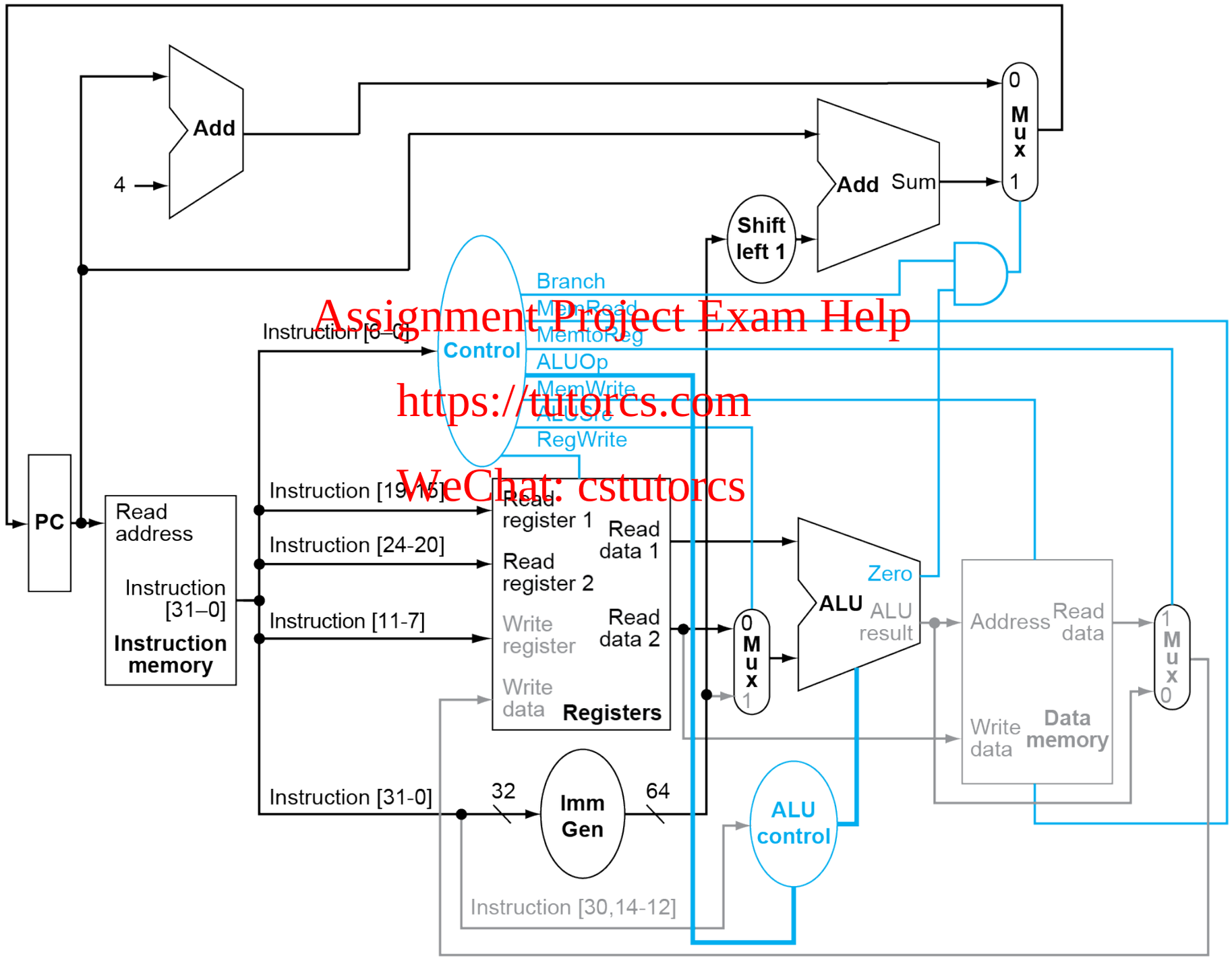
R-Type Instruction



Load Instruction



BEQ Instruction

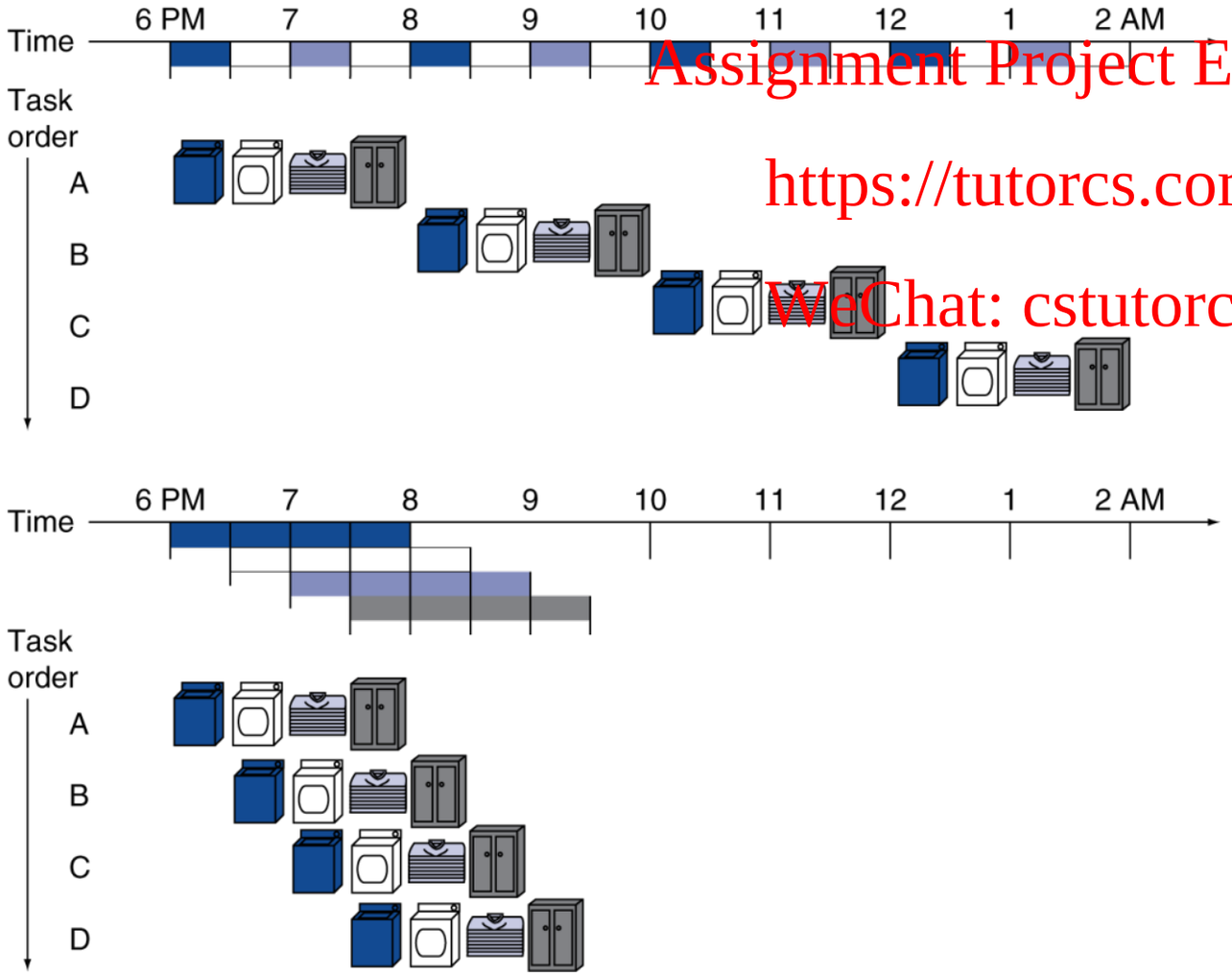


Performance Issues

- 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
- We will improve performance by pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance
- How many "stages" is this laundry process?



- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

Good Exam Questions

- What are the datapath changes necessary to support new instruction X?
- Given a datapath, what are the control signals necessary to perform instruction Y?
- Given a set of control signals and opcodes, what is the logic for generating control signal Z?