HANDONG GLOBAL UNIVERSITY

Chapter 4
The Processor

Computer Architecture and Organization

School of CSEE

1. Review

- Logic design review
- MIPS instruction review

2. Building Datapath

3. A Simple Implementation Scheme

Introduction

- In this section, we will review several issues covered in Logic Design course.
- Make sure that you know the difference between combinational circuit and sequential circuit.
- Also be aware of the behavior of synchronous sequential circuit.

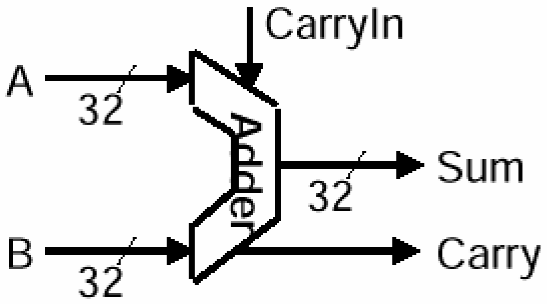
Two Types of Implementation Components

- Elements that operate on data values (combinational) : e.g. ALU
  - combinational : outputs depends only on the current inputs (same input → same output)

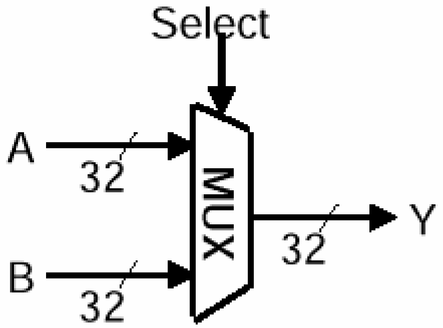
- Elements that contain state (sequential) : e.g. memory and registers
  - sequential : outputs depends on the current inputs and current state. (same input → possibly different output)

Review : Combinational Logic Elements

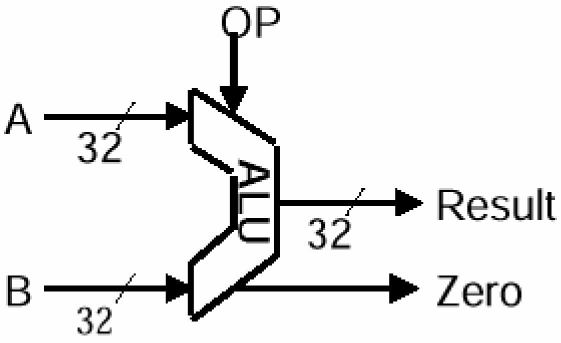
- **Adder**

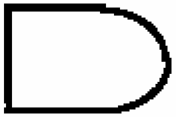


- **MUX**

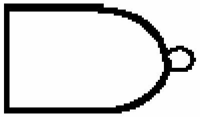


- **ALU**

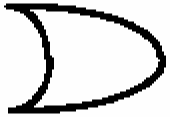




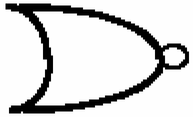
AND



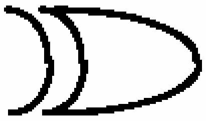
NAND



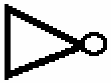
OR



NOR



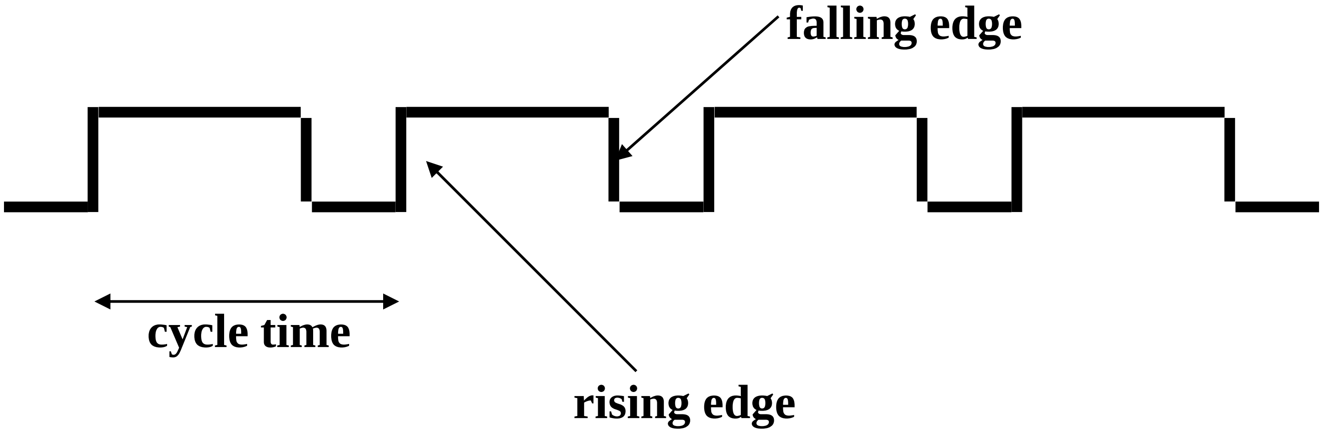
XOR



NOT

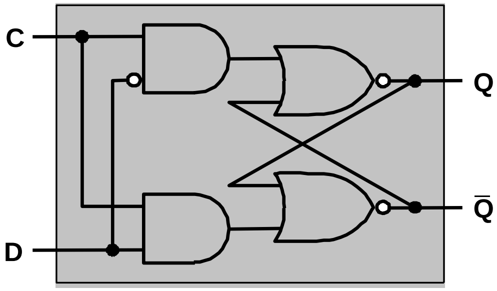
Review : State Elements

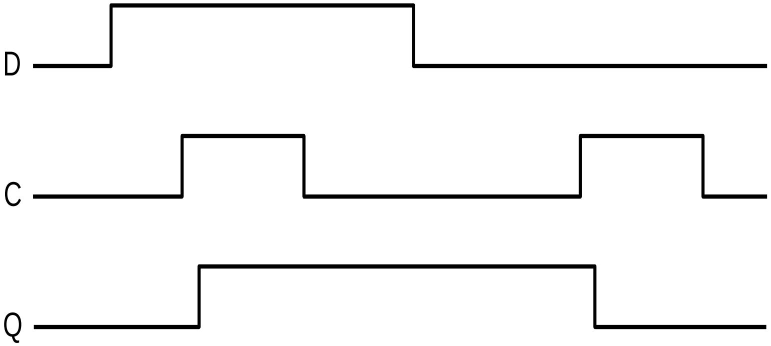
- Unclocked vs. Clocked (Asynchronous vs. Synchronous)
- Clocks used in synchronous logic – edge triggered (vs. master-slave)
  - when should an element that contains state be updated?



Review : D-latch

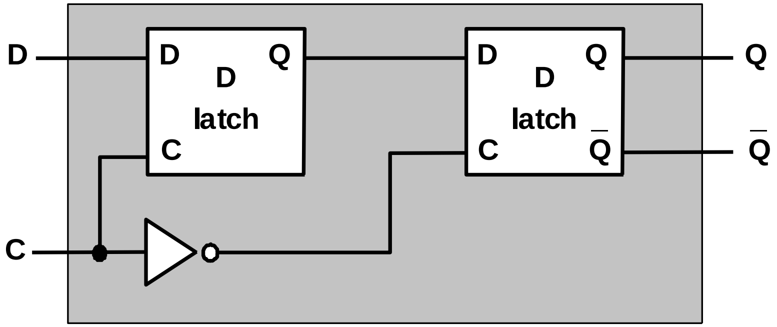
- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement
- The output is complemented while clock signal is active.

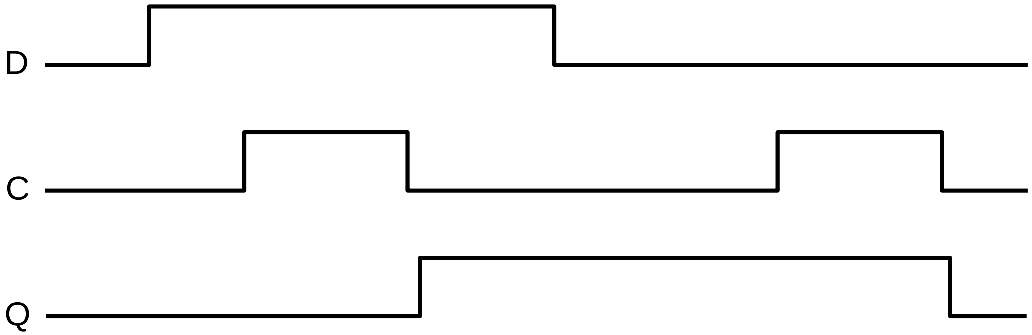




Review : D flip-flop

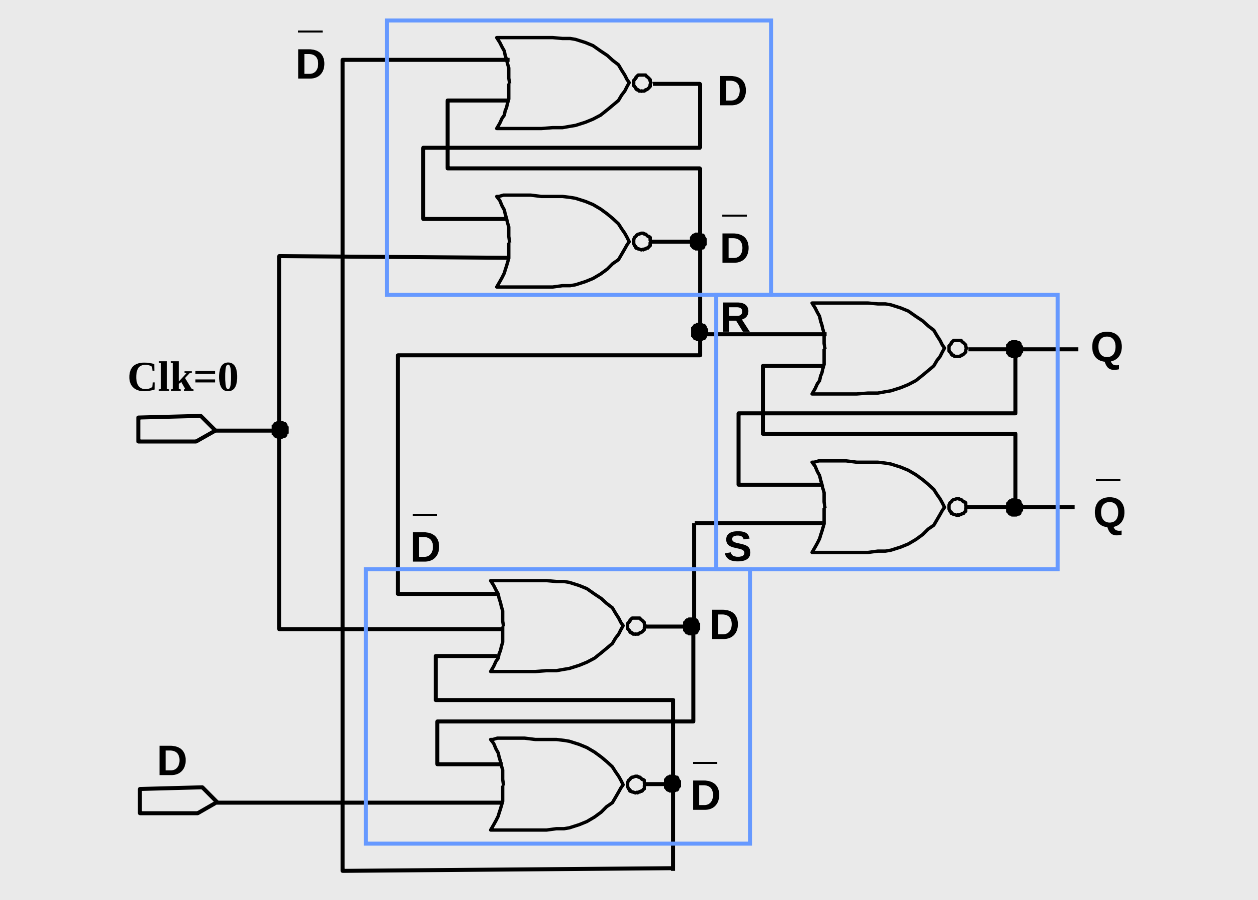
- **Output changes only on the clock edge**
- **Master-Slave flip-flop : have ones (or zeros) catching problem**





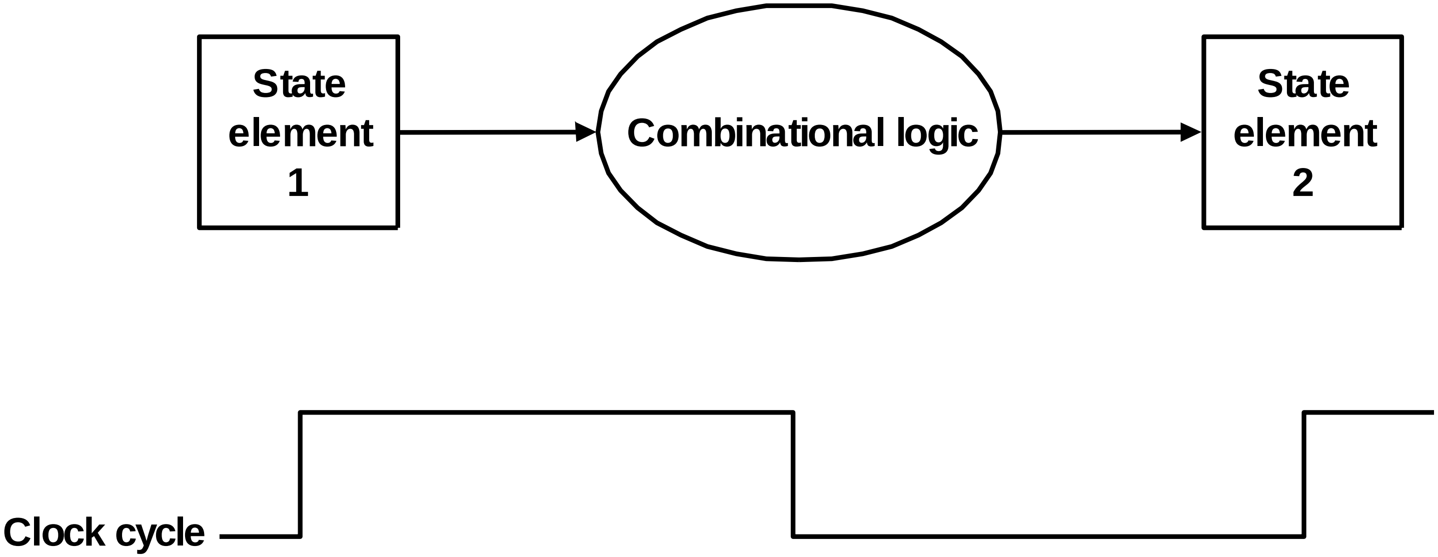
Review : D flip-flop

- **Edge-triggered flip-flop**



Review : Our Implementation

- **An edge triggered methodology**
- **Typical execution:**
  - **read contents of some state elements,**
  - **send values through some combinational logic**
  - **write results to one or more state elements**



Exercise

Explain the difference between synchronous sequential circuit and asynchronous sequential circuit.

Summary

- In sequential circuit the output depends on present state and input, while the output depends only on input in combinational circuit.
- In synchronous sequential circuit there is a special signal called ‘clock’ which synchronize the action of all components.

Instruction format Summary

- **Three MIPS Instruction Formats:**

<table>
<tr><td>R:</td><td>op</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
<tr><td>I:</td><td>op</td><td>rs</td><td>rt</td><td colspan="3">constant or address</td></tr>
<tr><td>J:</td><td>op</td><td colspan="5">jump address (word address)</td></tr>
</table>

Instruction Summary

Category	Format	Instruction	Format	Meaning
Arithmetic	R	ADD	ADD rd,rs,rt	rd ←rs + rt
	R	SUB	SUB rd,rs,rt	rd ← rs – rt
	I	ADDI	ADDI rt,rs,imm16	rt ← rs + se(imm16)
	R	SLT	SLT rd,rs,rt	rd ← (rs<rt)?1:0
	I	SLTI	SLTI rt,rs, imm16	rt ← (rs>se(imm16))?1:0
Logical	R	AND	AND rd,rs,rt	
	R	OR	OR rd,rs,rt	
	I	ANDI	ANDI rt,rs,imm16	rt ← rs & ze(imm16)
	R	NOR	NOR rd, rs, rt	
	R	SLL	SLL rd, rs, shamt	
	R	SRL	SRL rd, rs, shamt	
Data Transfer	I	LW	LW rt, imm16(rs)	rt ←M[rs+se(imm16)]
	I	SW	SW rt, imm16(rs)	M[rs+se(imm16)]←rt
Control Transfer	I	BEQ	BEQ rs,rt, imm16	if (rs == rt) PC ← PC+se(imm16)*4
	I	BNE	BNE rs,rt, imm16	If (rs != rt) PC ← PC+se(imm16)*4
	R	JR	JR rs	PC ← rs
	J	JAL	JAL imm26	$ra ← PC+4; PC ← imm26*4
	J	J	J imm26	PC ← imm26*4

1. **Review**
    - Logic design review
    - MIPS instruction review
2. **Building Datapath**
3. **A Simple Implementation Scheme**

Introduction

- In this section we will learn basic datapath for single clock cycle datapath. The instruction is executed in one clock cycle in single clock cycle CPU.
- The datapath is arranged according to the MIPS instruction set.

The Processor: Datapath & Control

- **We're ready to look at an implementation of the MIPS**
- **Simplified to contain only:**
  - **memory-reference instructions:** `lw, sw`
  - **arithmetic-logical instructions:** `add, sub, and, or, slt`
  - **control flow instructions:** `beq, j`
- **Generic Implementation:**

fetch

1. **use the program counter (PC) to supply instruction address**
2. **get the instruction from memory**

execution

3. **read registers**
4. **use the instruction to decide exactly what to do**

→ Stored Program Concept

add Operation

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000				00000	100000

- **add rd, rs, rt**

```
IR ← mem[PC];
R[rd] ← R[rs] + R[rt];
PC ← PC + 4;
```

RTL Description

Fetch instruction from memory

ADD operation

Calculate next address

Register Transfer Language

RTL

sub Operation

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
000000				00000	100010

- **add rd, rs, rt**

```
IR ← mem[PC];
R[rd] ← R[rs] + ~R[rt] + 1;
PC ← PC + 4;
```

2's compl

RTL Description

Fetch instruction from memory

SUB operation

Calculate next address

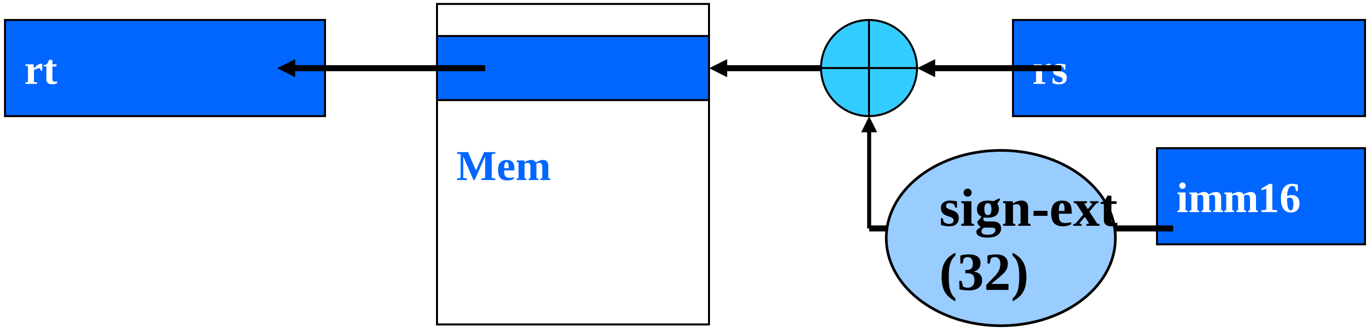
lw Operation

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits
100011			

- **lw rt, rs, imm16**

	RTL Description
`IR ← mem[PC];`	Fetch instruction from memory
`Addr ← R[rs] + SignExt(imm16);`	Compute memory address
`R[rt] ← Mem[Addr];`	Load data into register
`PC ← PC + 4;`	Calculate next address



sw Operation

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits
101011			

- **sw rt, rs, imm16**

```
IR ← mem[PC];
Addr ← R[rs] + SignExt(imm16);
Mem[Addr] ← R[rt];
PC ← PC + 4;
```

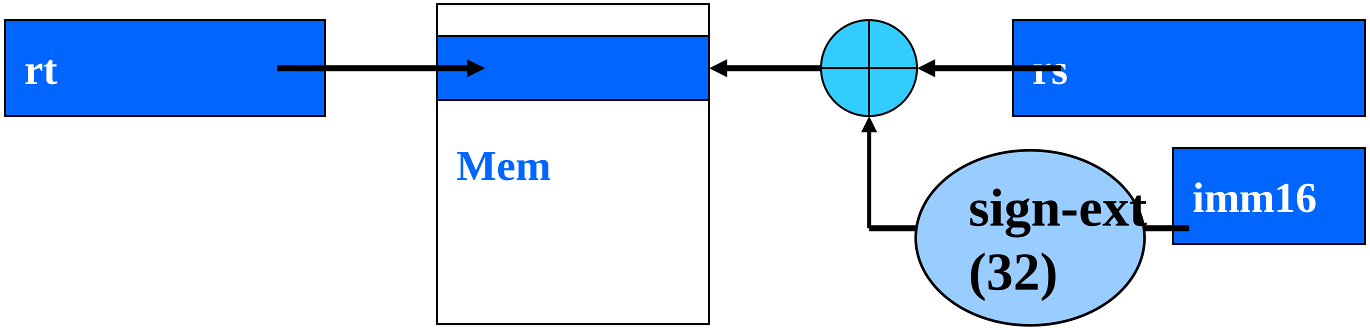
RTL Description

Fetch instruction from memory

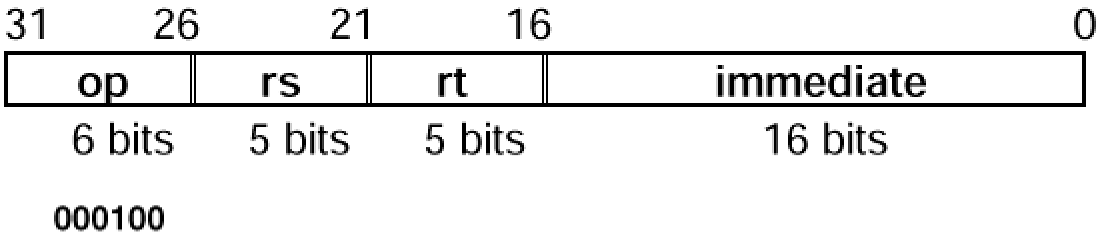
Compute memory address

Store data into memory

Calculate next address



beq Operation



- **beq rt, rs, imm16**

```
IR ← mem[PC];
Cond ← R[rs] + ~R[rt] + 1;
PC ← Cond ? PC + 4;
              : PC + 4 + (SignExt(imm16) << 2)
```

RTL Description

Fetch instruction from memory

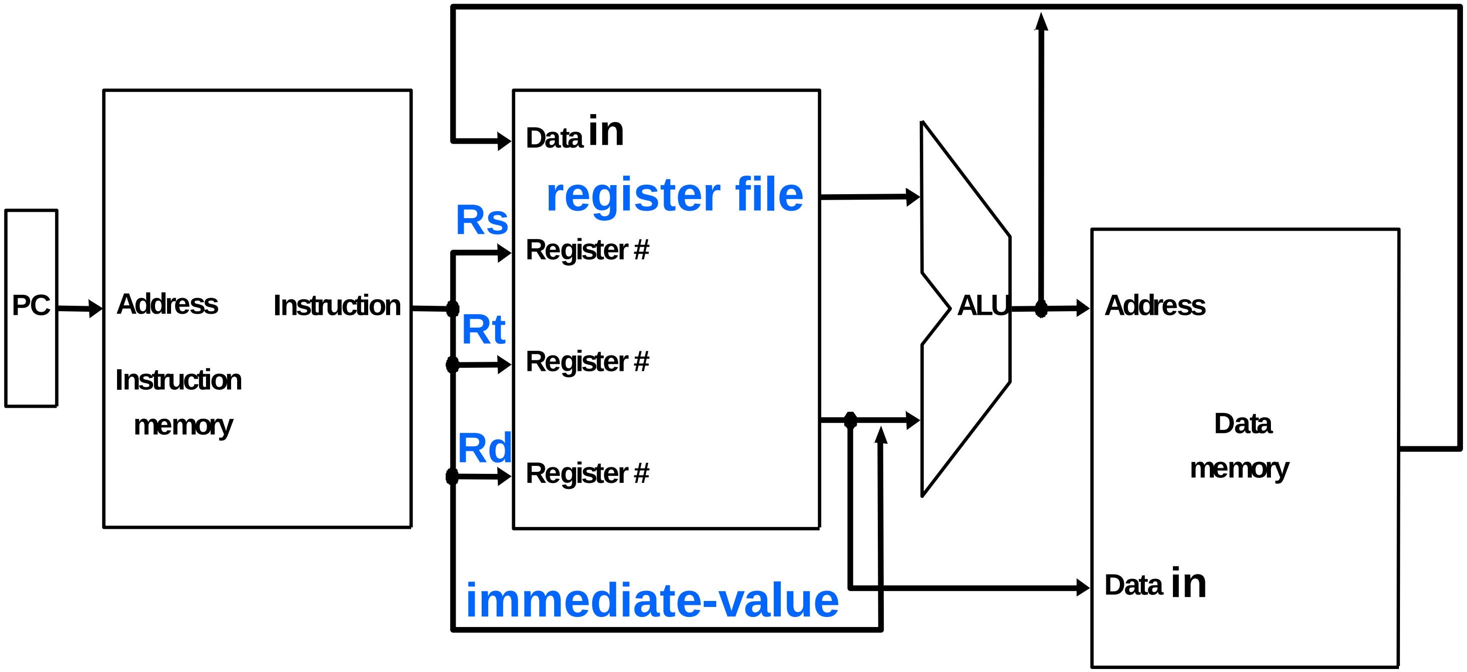
Compute conditional `Cond`

Fall through if non-zero `Cond`

Branch if zero `Cond`

However, the instruction won’t be stored in IR in single clock cycle implementation. Why?

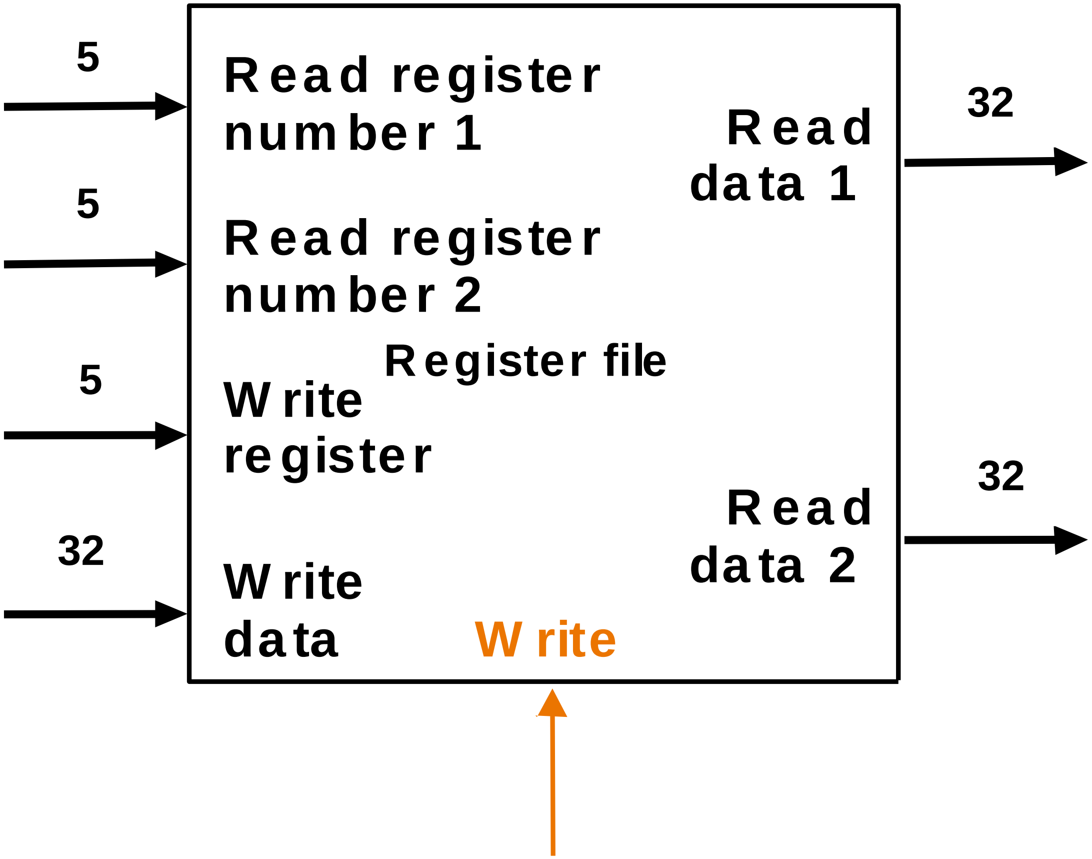
Abstract/Simplified View of Implementation



We need separate memory for instruction and data. Why?

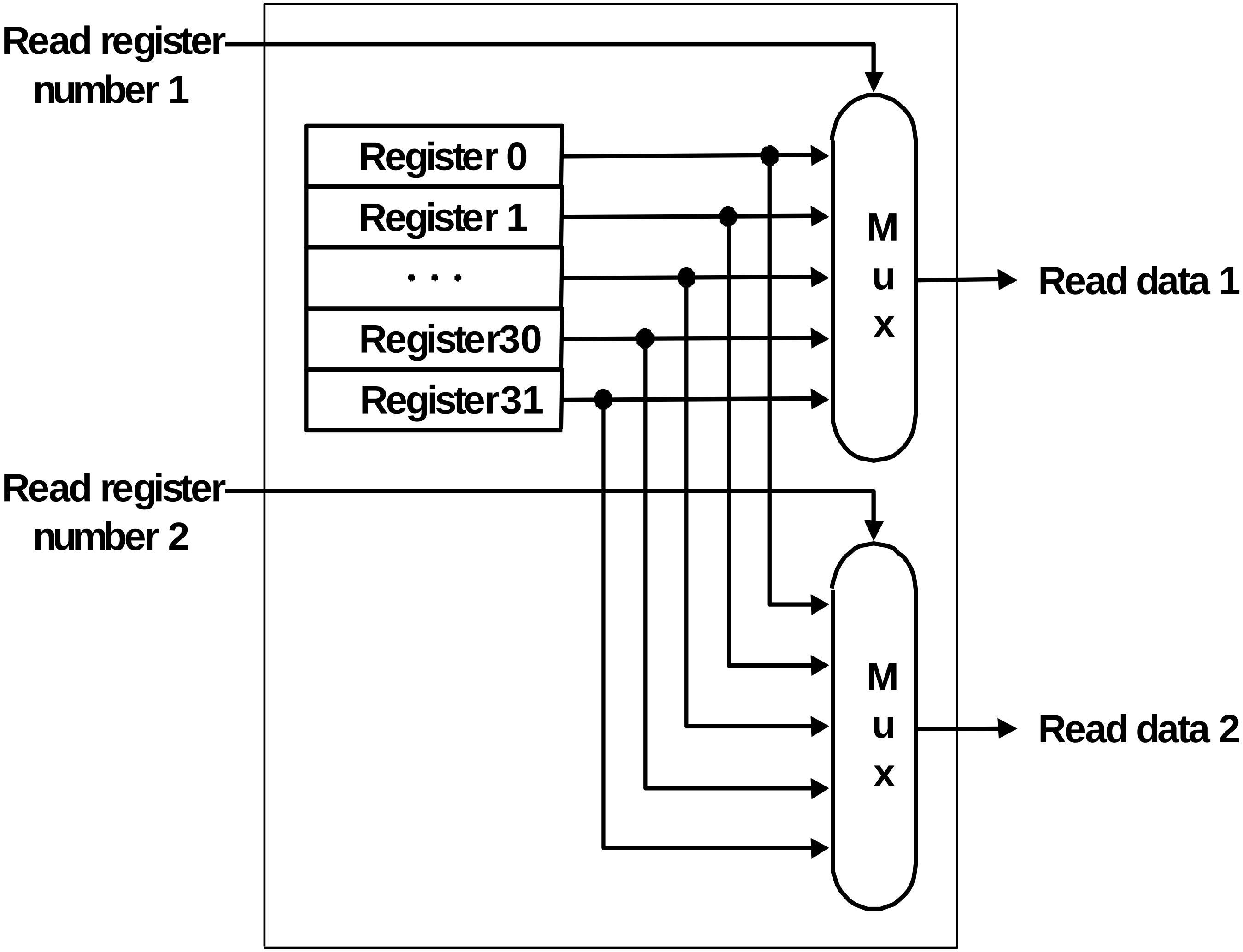
Register File

- **Has been built using D flip-flops**



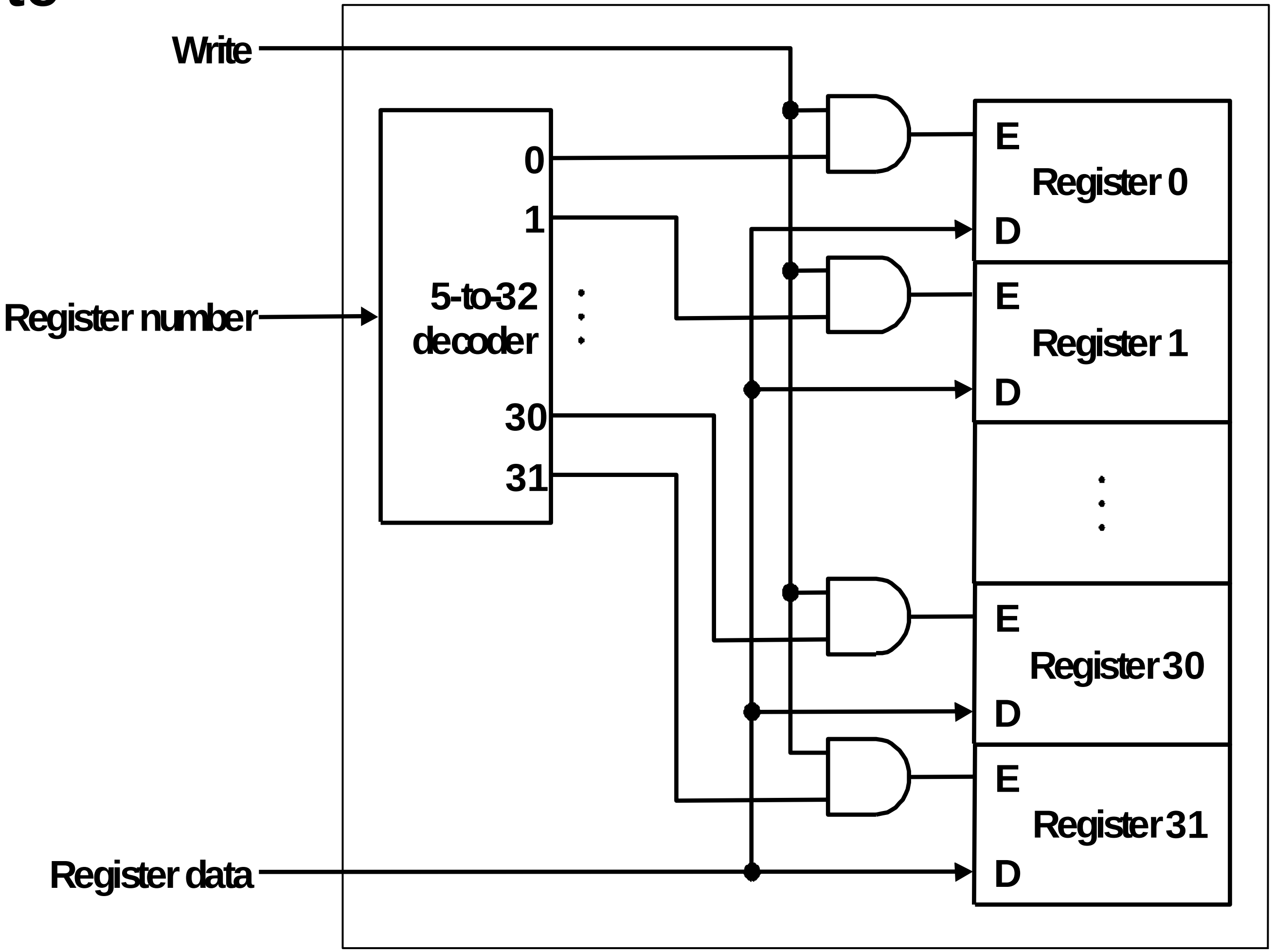
Register File

- **Has been built using D flip-flops**



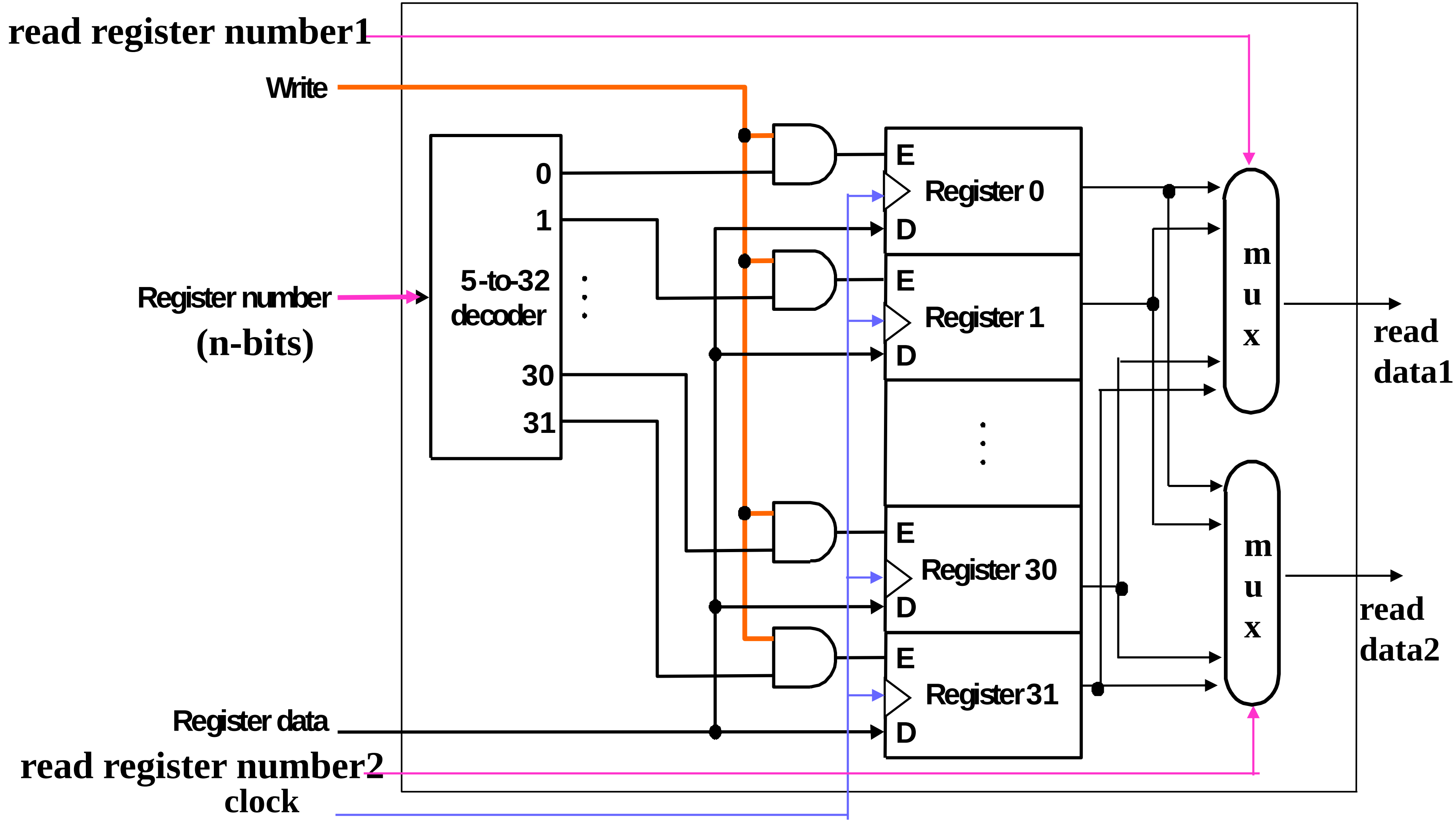
Register File

- **Note: we still use the real clock to determine when to write**



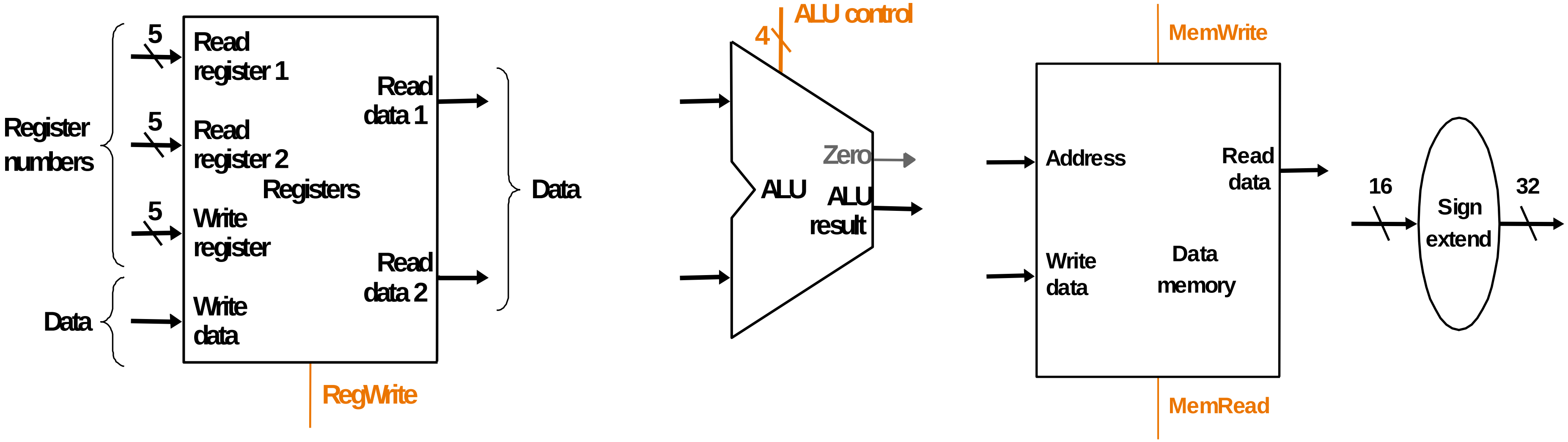
Register File

- Note: We can read and write at the same time!



Components for Simple Implementation

- Include the functional units we need for each instruction

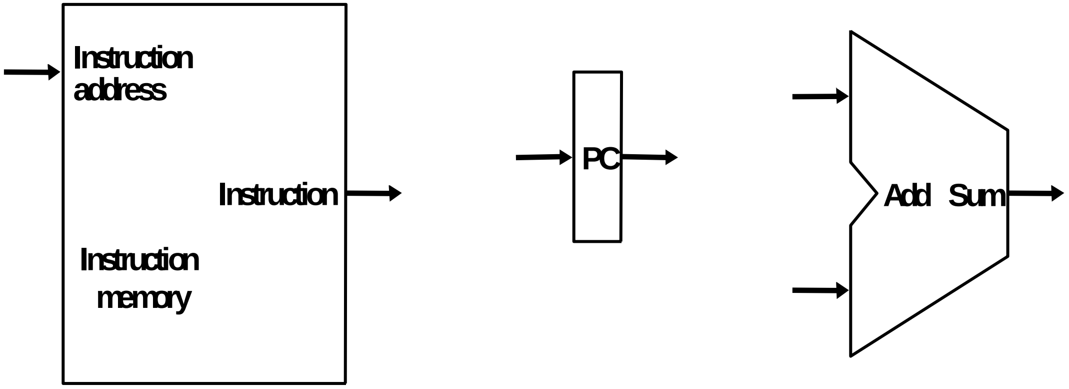


a. Registers

b. ALU

c. Data memory unit

d. Sign-extension unit



e. Instruction memory

f . Program counter

g. Adder

Why do we need this stuff?

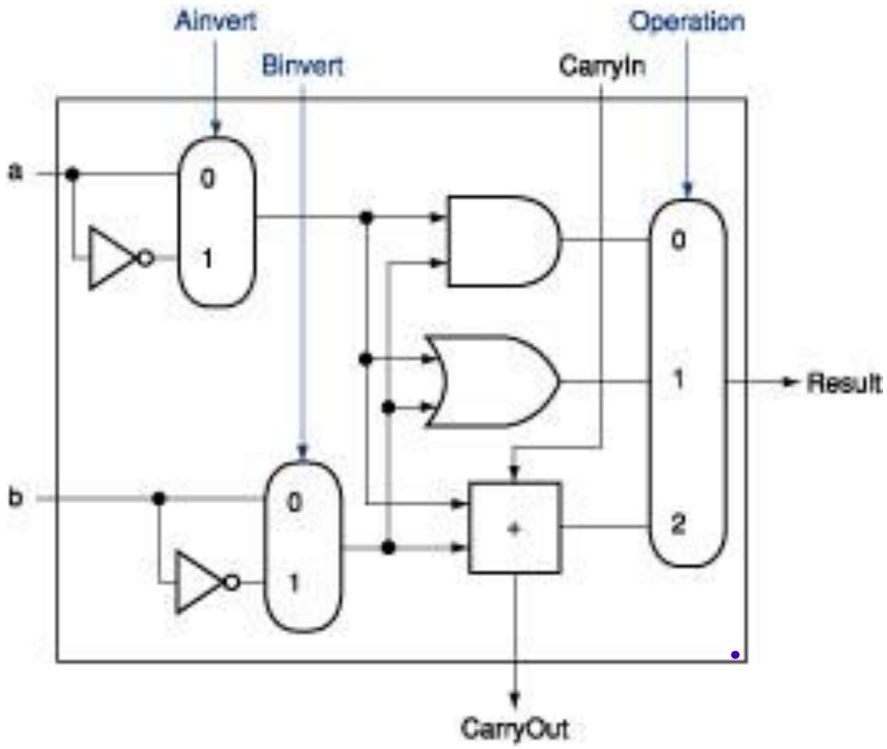
Instruction fetch(e~g)

R-format ALU operations(a,b)

sw,lw instr. (a,b,c,d)

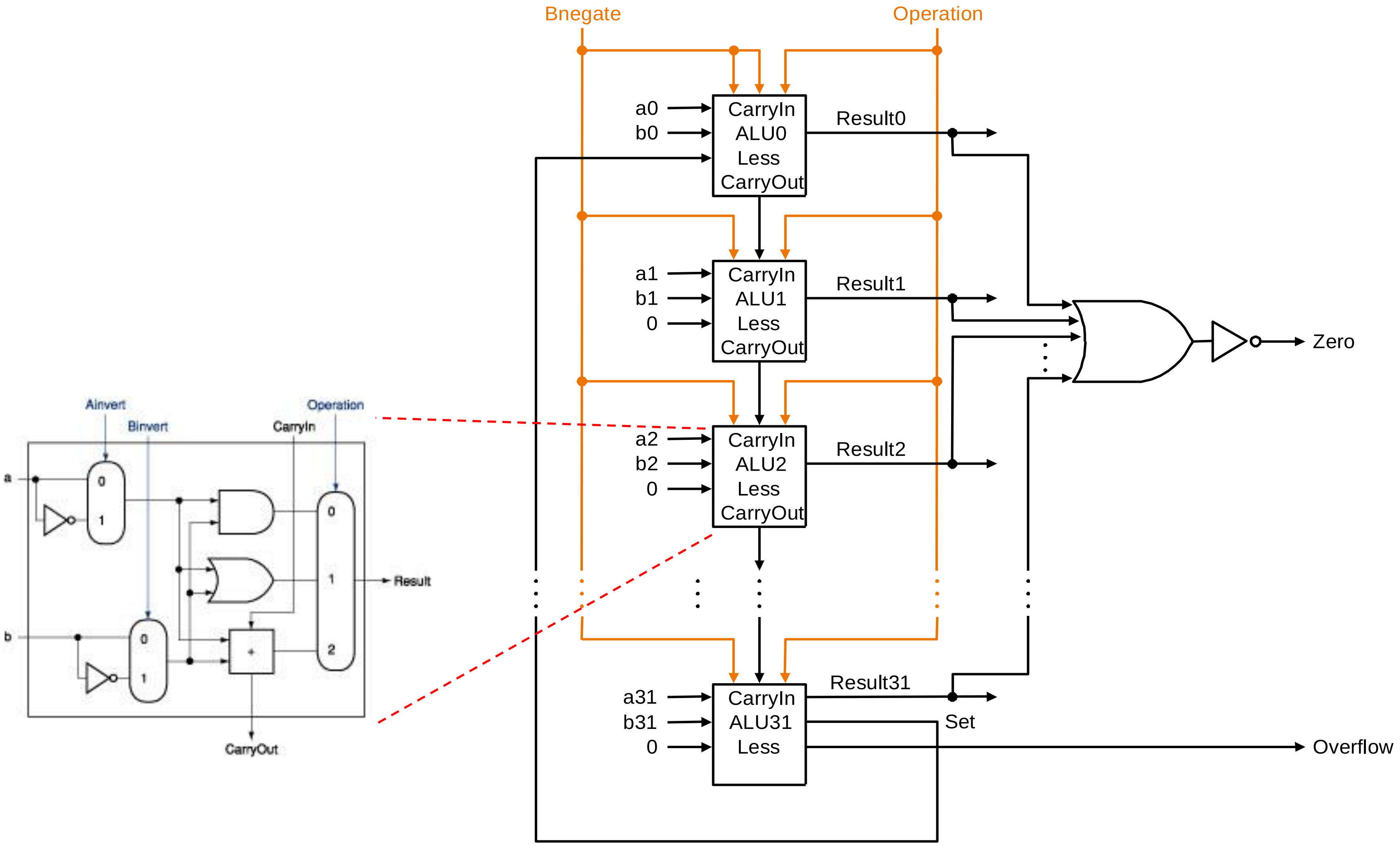
beq instruction (a,b,d,f,g)

ALU with 4 control signals



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

MIPS ALU



•*Note: zero is 1 when the result is zero!*

Status Bit

Status bits (or condition code)

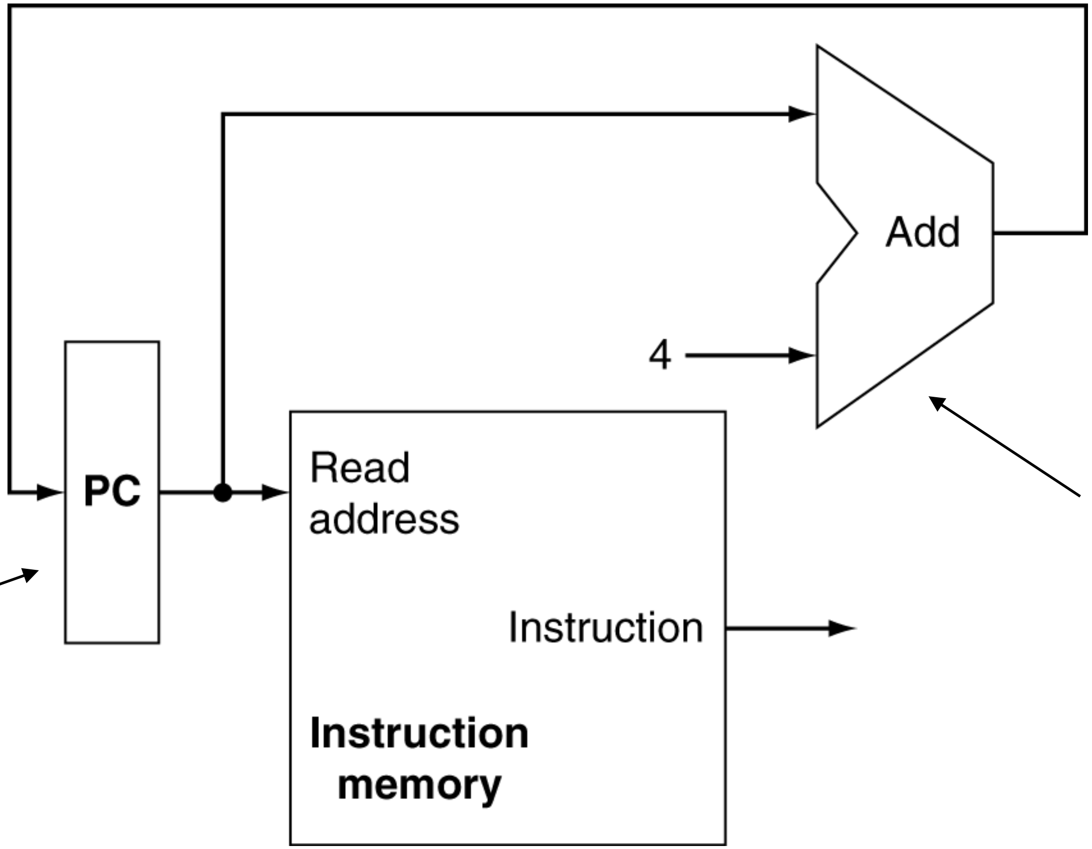
1. Z : zero – 1 if result is 0
   0 if result is non-zero

2. S : sign – 0 if result is positive number
   1 if result is negative number

3. V : overflow – 0 if overflow does not occur
   1 if overflow occurs

4. C : carryout – 0 if there is a carryout
   1 if there is no carryout

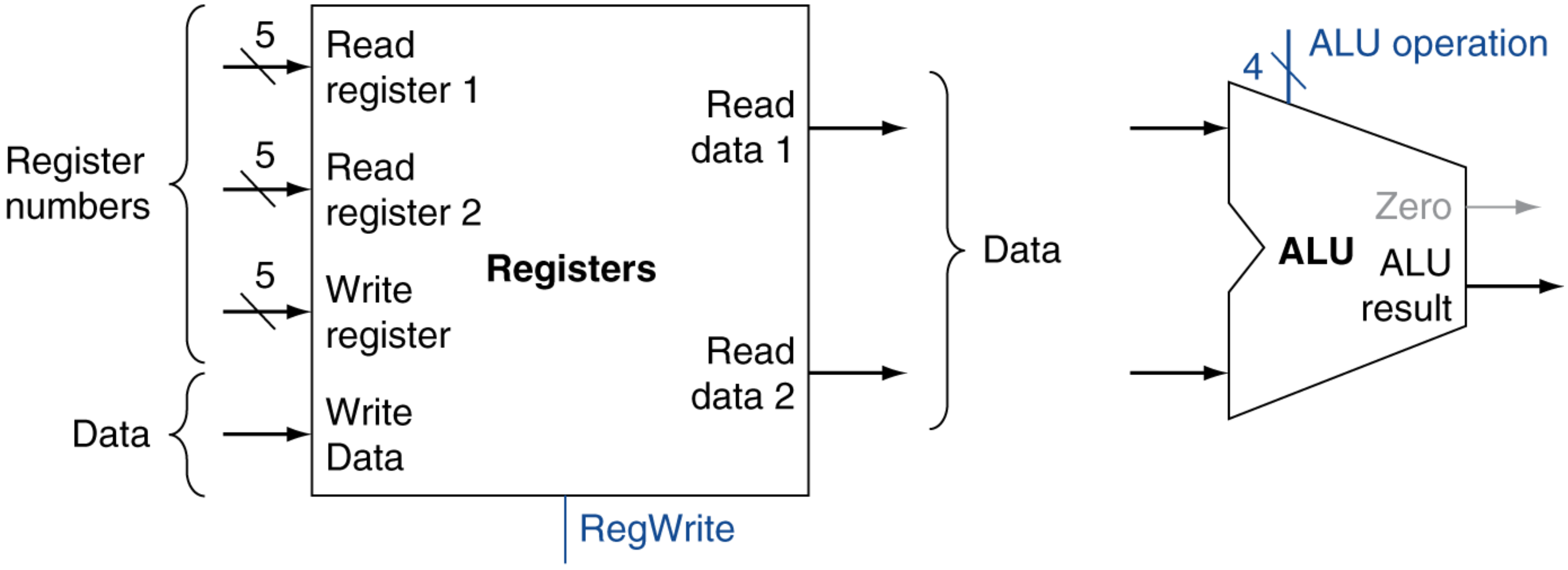
Instruction Fetch Datapath



32-bit register

Increment by 4 for next instruction

Datapath for R-type Instructions

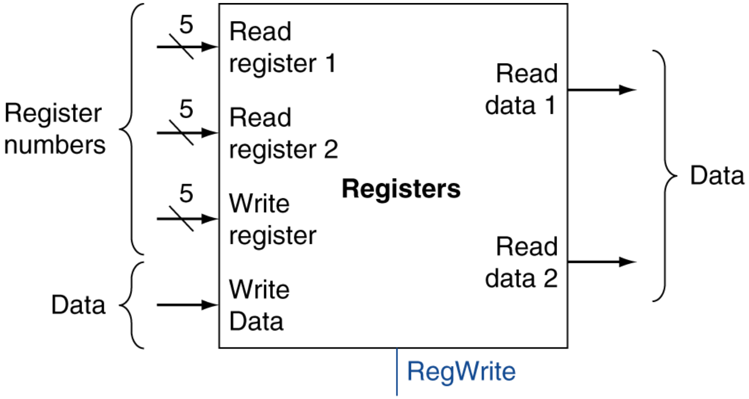


a. Registers

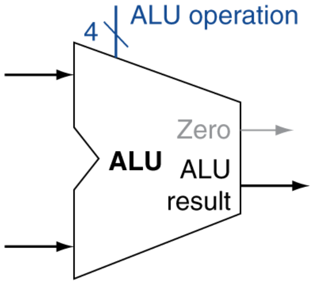
b. ALU

add, sub, and, or, slt

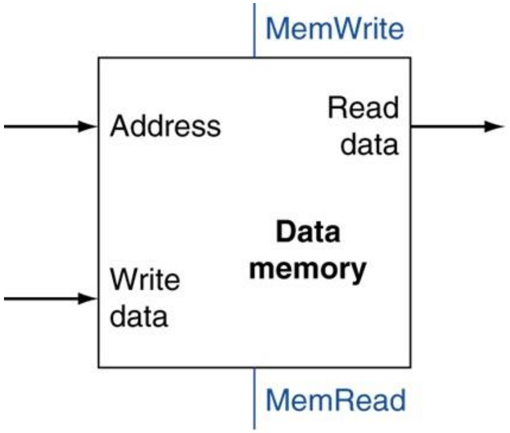
Datapath for a Load or Store

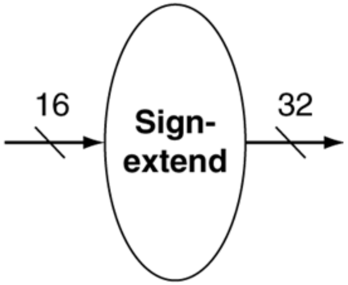


Registers



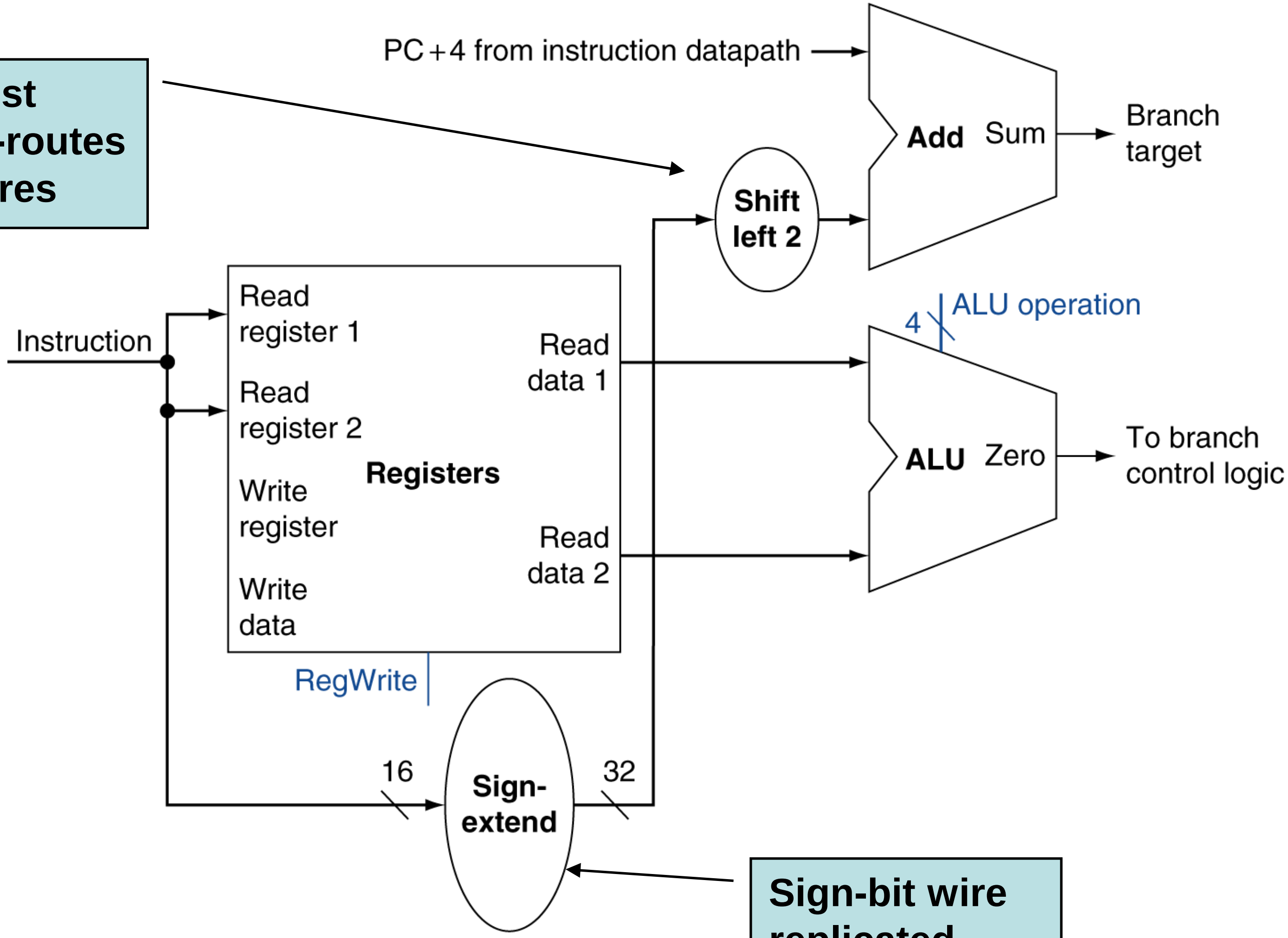
ALU





lw, sw

Datapath for a Branch

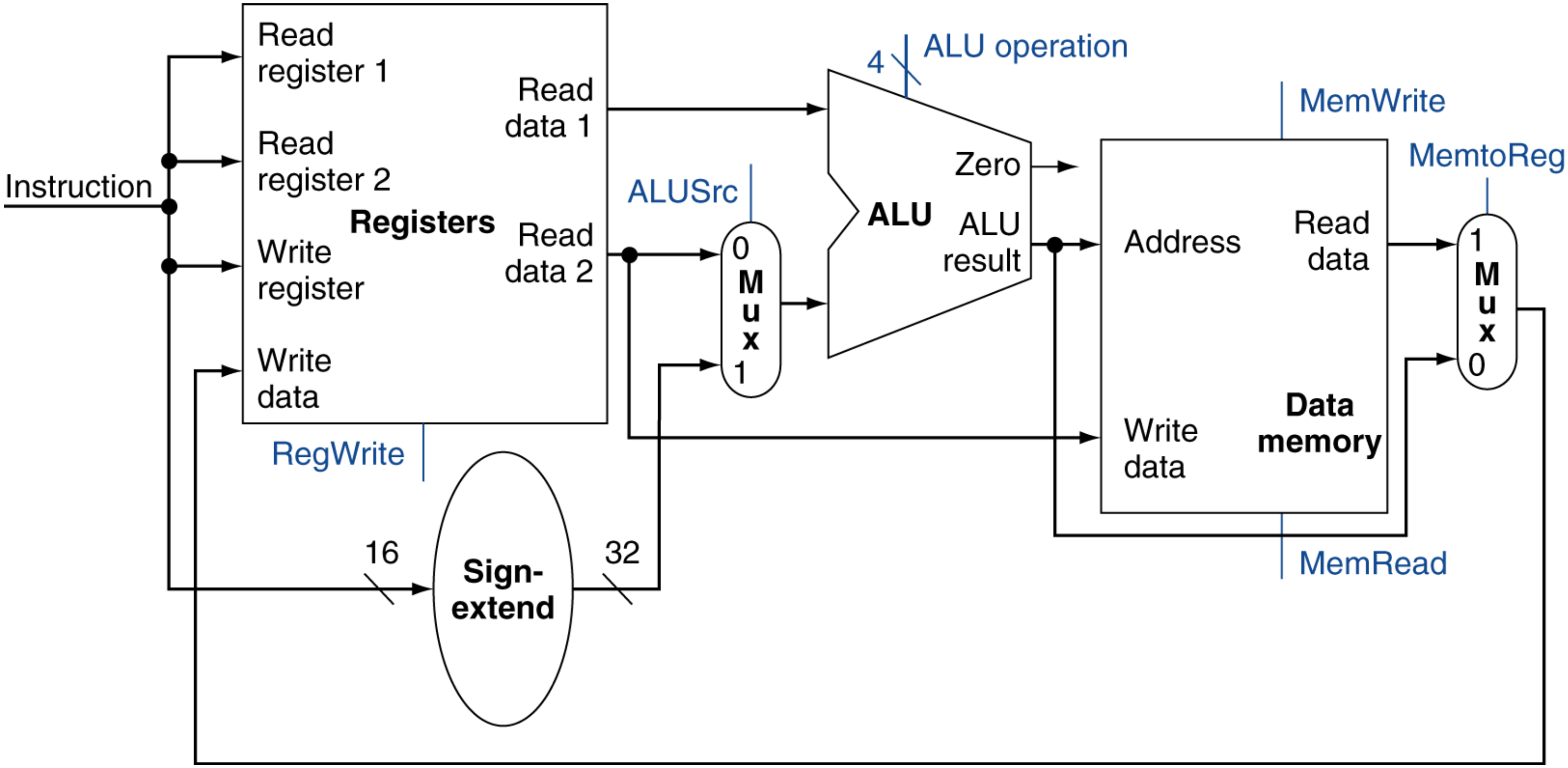


Just re-routes wires

Sign-bit wire replicated

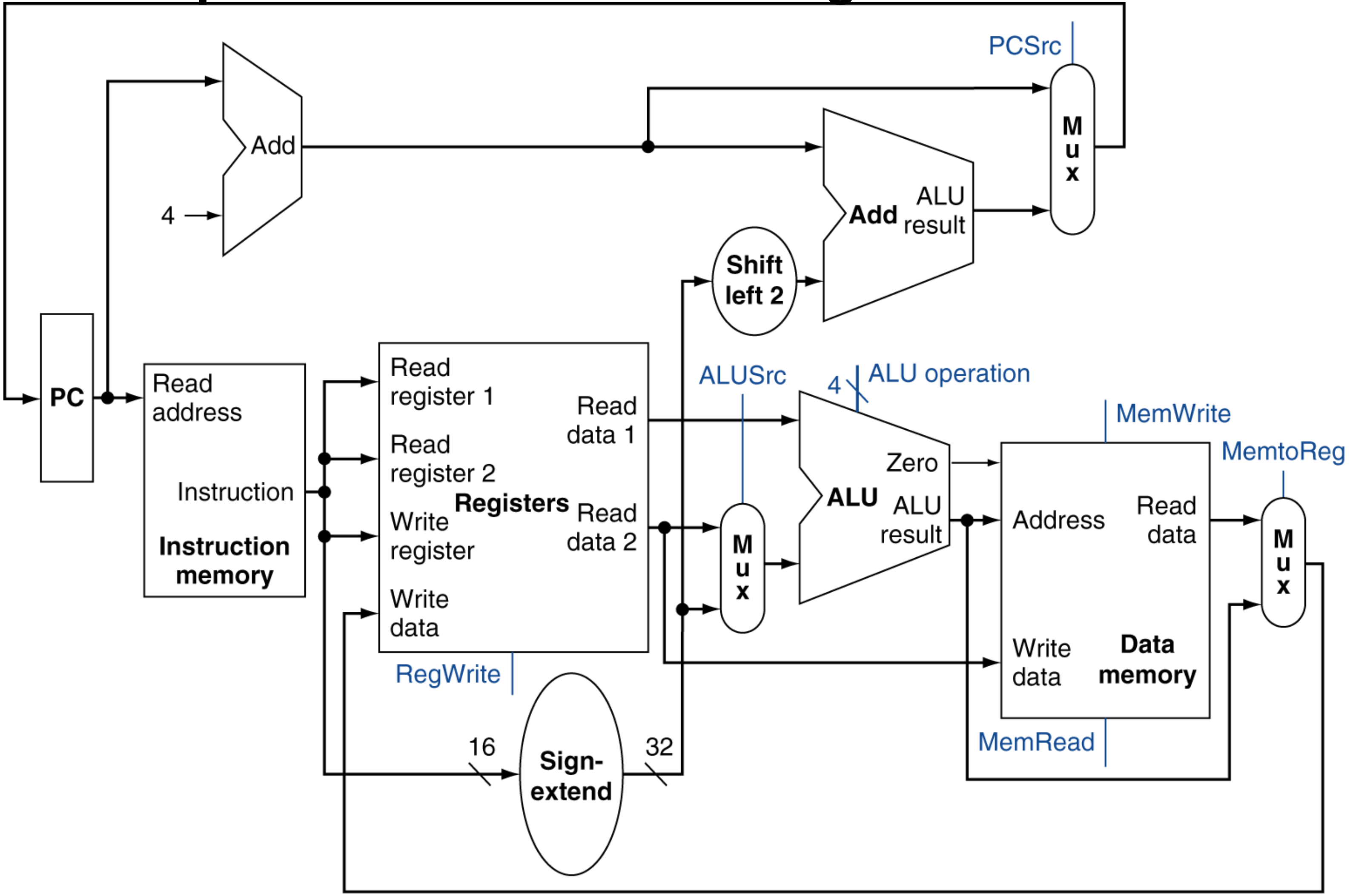
R-type + (Load or Store)

- **Need two extra multiplexers.**



R-type + (Load / Store) + Branch + PC

- **Use multiplexers to stitch them together**



Summary

- We have built datapath for single clock cycle datapath. The components are arranged so that data can flow through them to produce correct output.

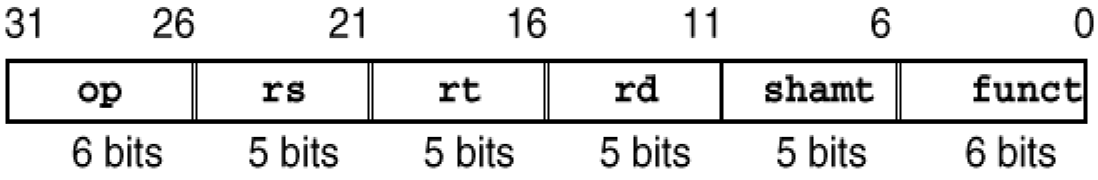
1. **Review**
    - Logic design review
    - MIPS instruction review
2. **Building Datapath**
3. **A Simple Implementation Scheme**

Introduction

- In this section we will learn control signals for datapath built in previous section.
- We will also see how data flows MIPS CPU for each MIPS instruction.
- Also we will try to measure its performance.

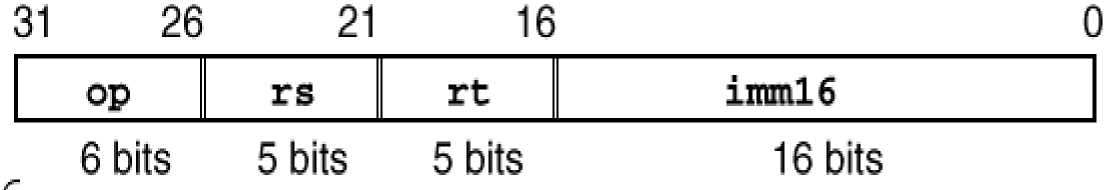
Operation/Instruction Format Summary

- Add, sub, and, or
  - `add rd, rs, rt`
  - `and rd, rs, rt`



31	26	21	16	11	6	0
op	rs	rt	rd	shamt	funct	
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

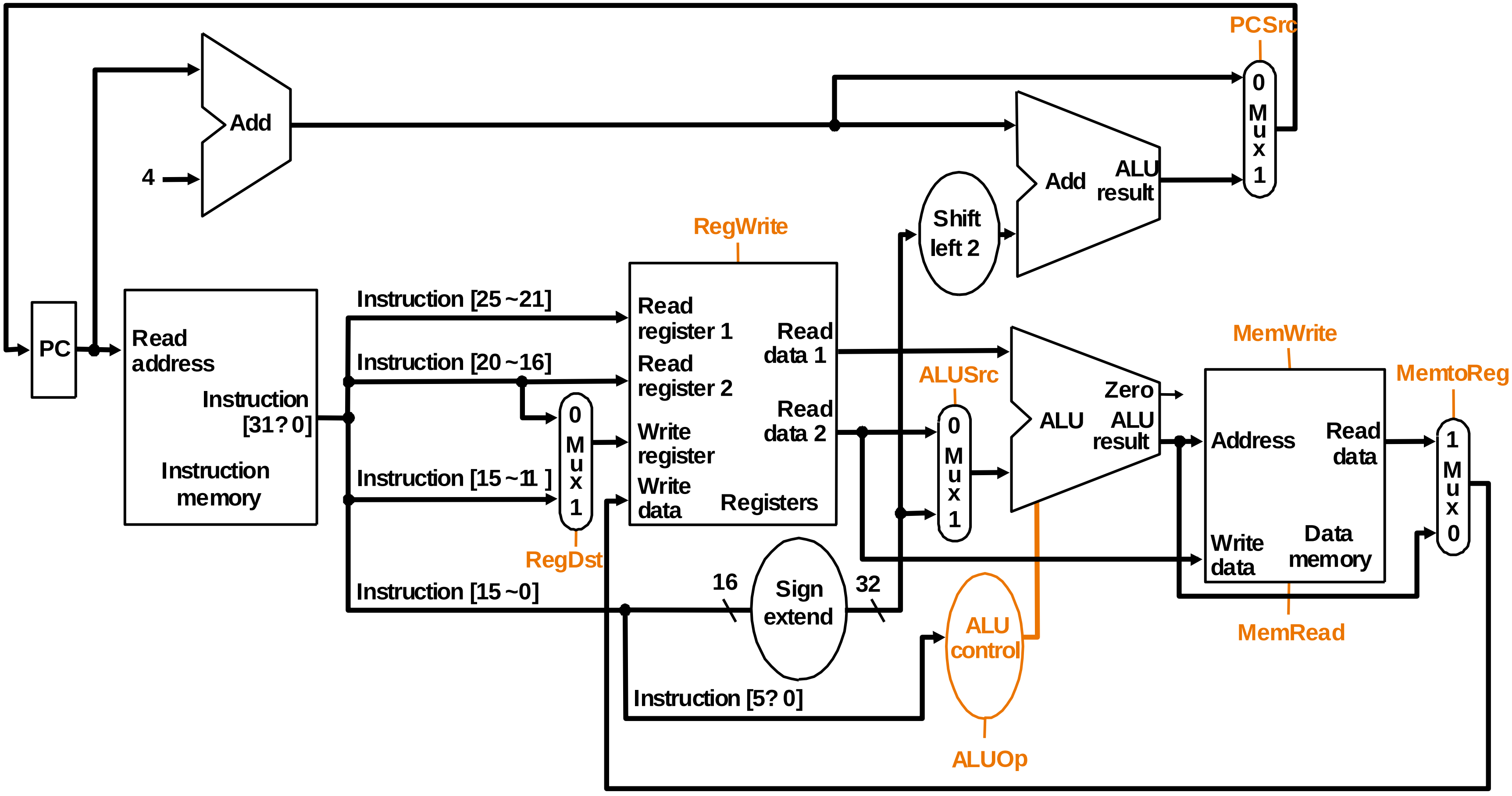
- Load, store
  - `lw rt, rs, imm16`
  - `sw rt, rs, imm16`
- Branch
  - `beq rs, rt, imm16`



31	26	21	16	0
op	rs	rt	imm16	
6 bits	5 bits	5 bits	16 bits	

R-type + (Load or Store) + Branch + PC

- **More multiplexer…**



Detailed Design of Control

- Control Signal Types
  - Selecting the operations to perform (ALU, read/write, etc.)
  - Controlling the flow of data (selecting multiplexer inputs)
- Information comes from the 32 bits of the instruction
- Example: add $8, $17, $18   Instruction Format:

add	000000	10001	10010	01000	00000	100000
	op	rs	rt	rd	shamt	funct
and	000000					100100
or	000000					100101
sub	000000					100010
slt	000000					101010

- ALU operation is based on op. code and function code

Detailed Design of Control : ALU Control

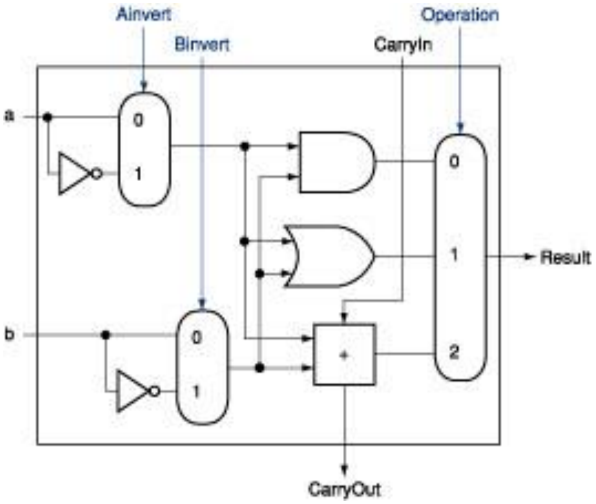
- **e.g., what should the ALU do with this instruction**

Example: lw $1, 100($2)

35	2	1	100
op	rs	rt	16 bit offset

lw	100011				Need 'add' op.
sw	101011				Need 'add' op.
beq	000100				Need 'sub' op.

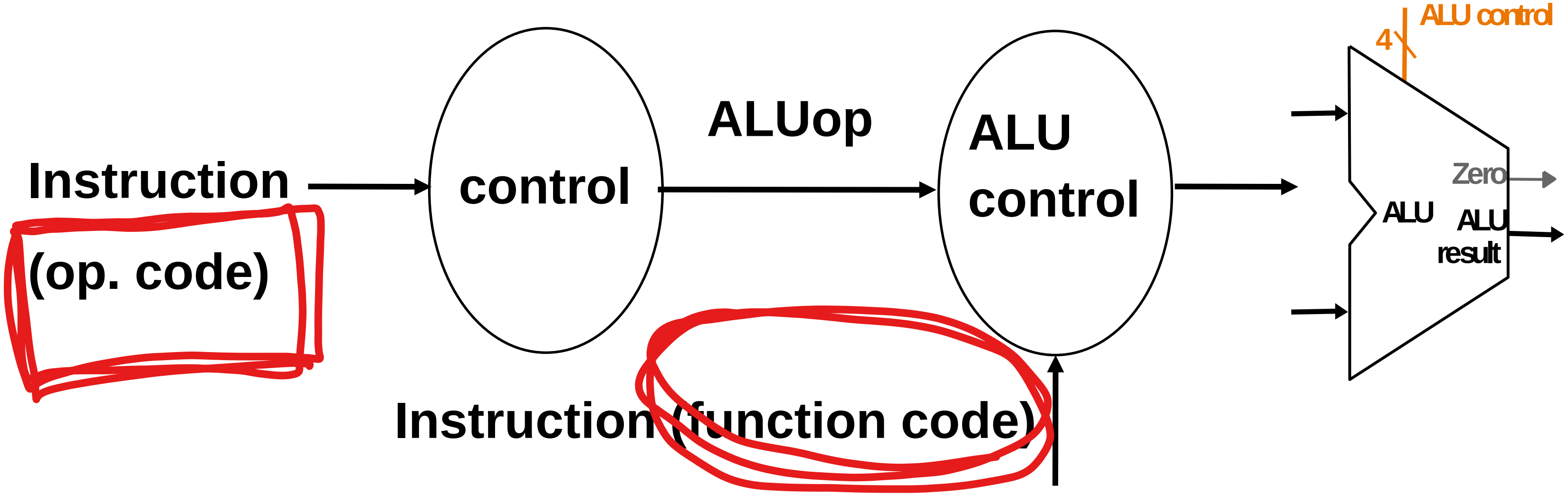
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR



Detailed Design of Control : ALU Control

- output : 4-bit ALU control input
- input :
  - ALUop from 'control' component
    - 00 = lw, sw
    - 01 = beq,
    - 10 = arithmetic

    ALUOp computed from op. code in instruction
  - function code for arithmetic



Instruction (op. code)

control

ALUop

ALU control

Instruction (function code)

Detailed Design of Control : ALU Control

- **Describe it using a truth table (can be turned into gates):**

	ALUOp		Function field						Operation
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
lw, sw	0	0	X	X	X	X	X	X	0010
beq	X	1	X	X	X	X	X	X	0110
R-format (add,sub, and,or,slt)	1	X	X	X	0	0	0	0	0010
	1	X	X	X	0	0	1	0	0110
	1	X	X	X	0	1	0	0	0000
	1	X	X	X	0	1	0	1	0001
	1	X	X	X	1	0	1	0	0111

ALU control signal and its action

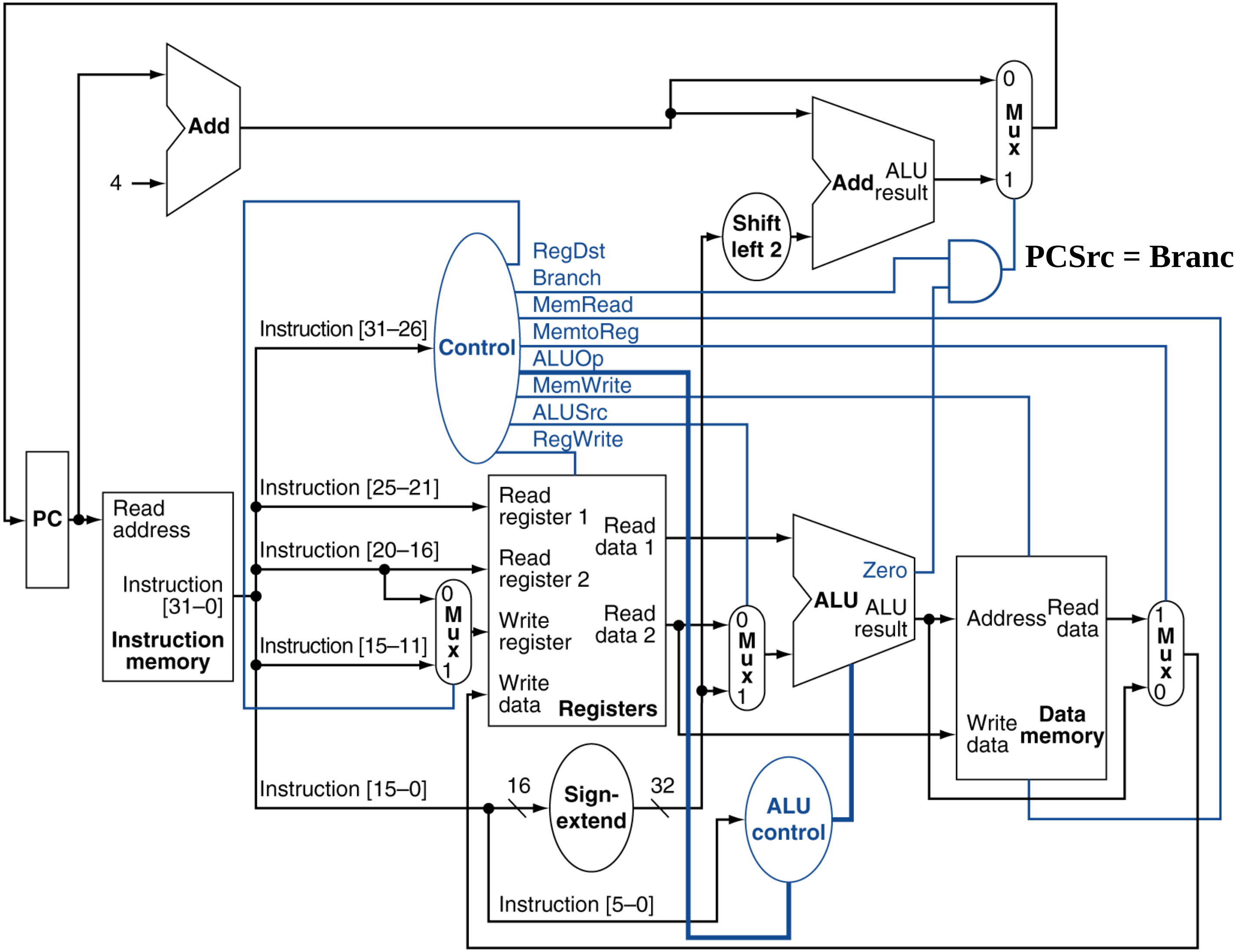
Arithmatic

Instruction opcode	ALUOp	Instruction Operation	Function field	Desired ALU action	ALU control Input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	banch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Control Signals

Signal name	Effect when disserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16)	The register destination number for the Write register comes from the rd field (bits15:11)
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 data2)	The second ALU operand is the sign-extended, lower 16 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

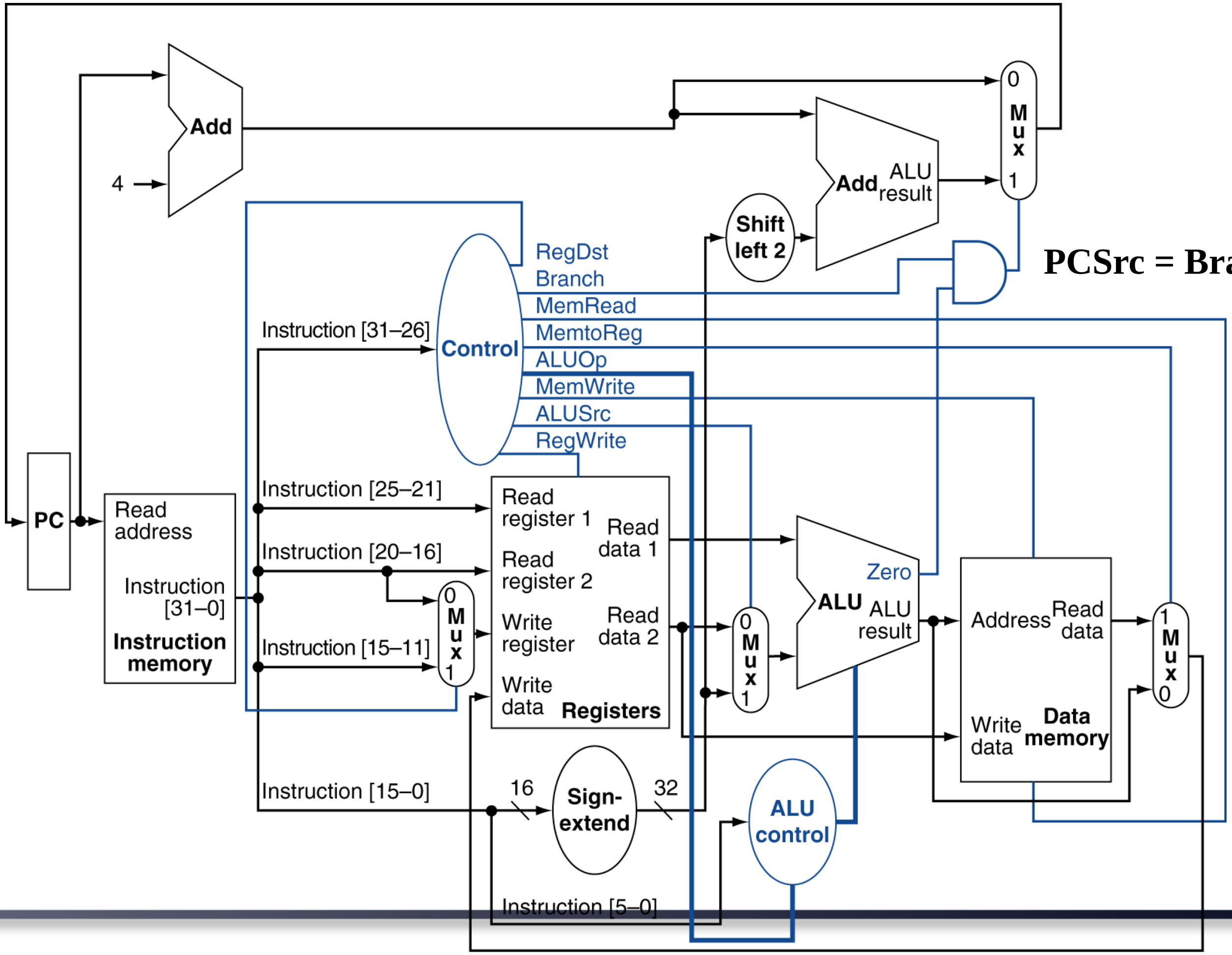
With Control Unit



PCSrc = Branch AND Zero

Detailed Design of Control

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp 1	ALUOp 0
R-format									
lw									
sw									
beq									



PCSrc = Branch. Zero

A Simple Cycle Implementation

= A single clock implementation

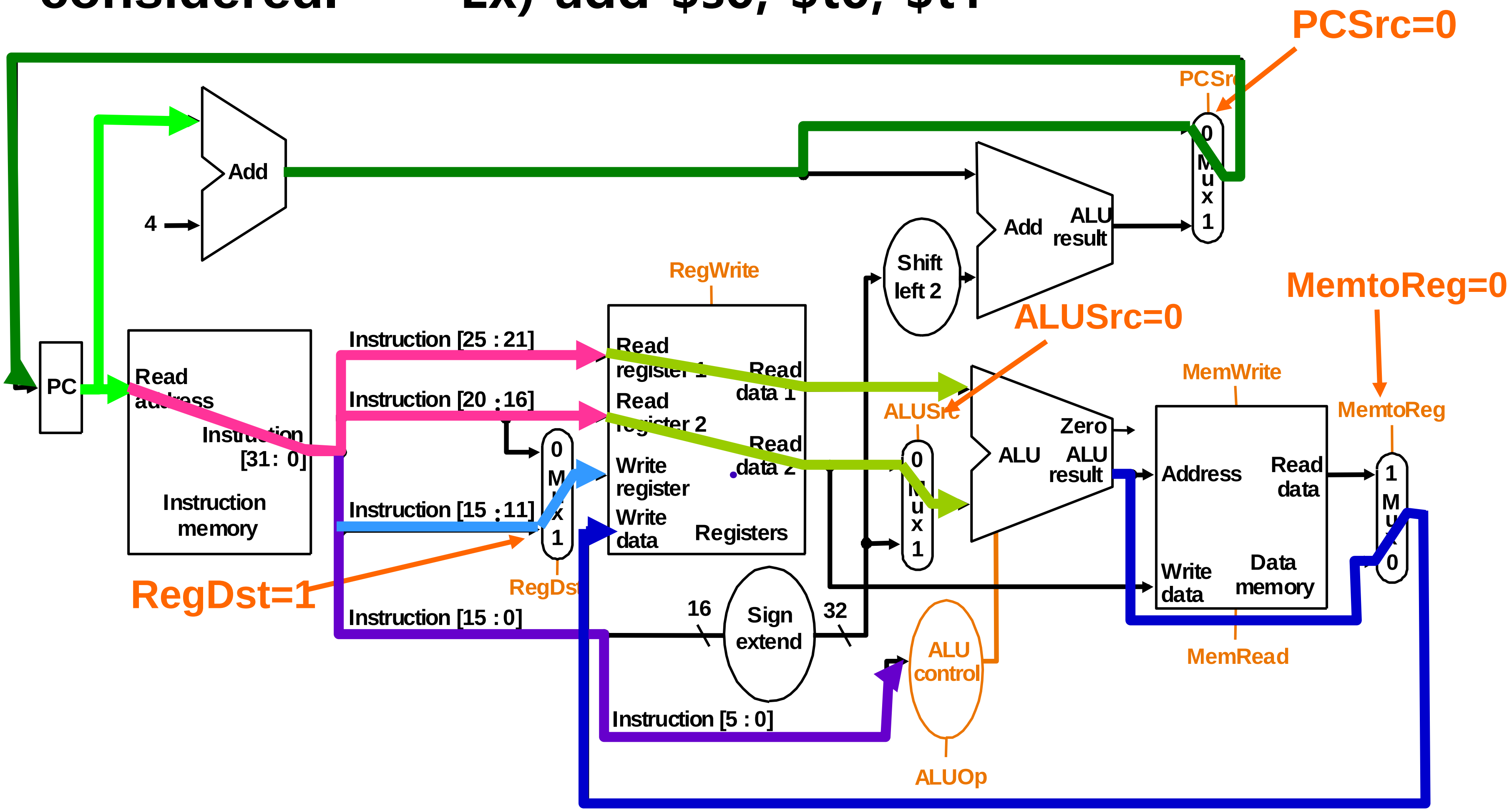
- All instructions are executed in one clock cycle. Thus, no resource can be used more than once per instruction.

➔ need separate instruction and data memory.

need several adders.

R-format instructions dataflow

- Only 'add', 'sub', 'and', 'or', and 'slt' instructions are considered. Ex) add $s0, $t0, $t1



R-format Operation Control

- We thus have the following control signals for these ops:

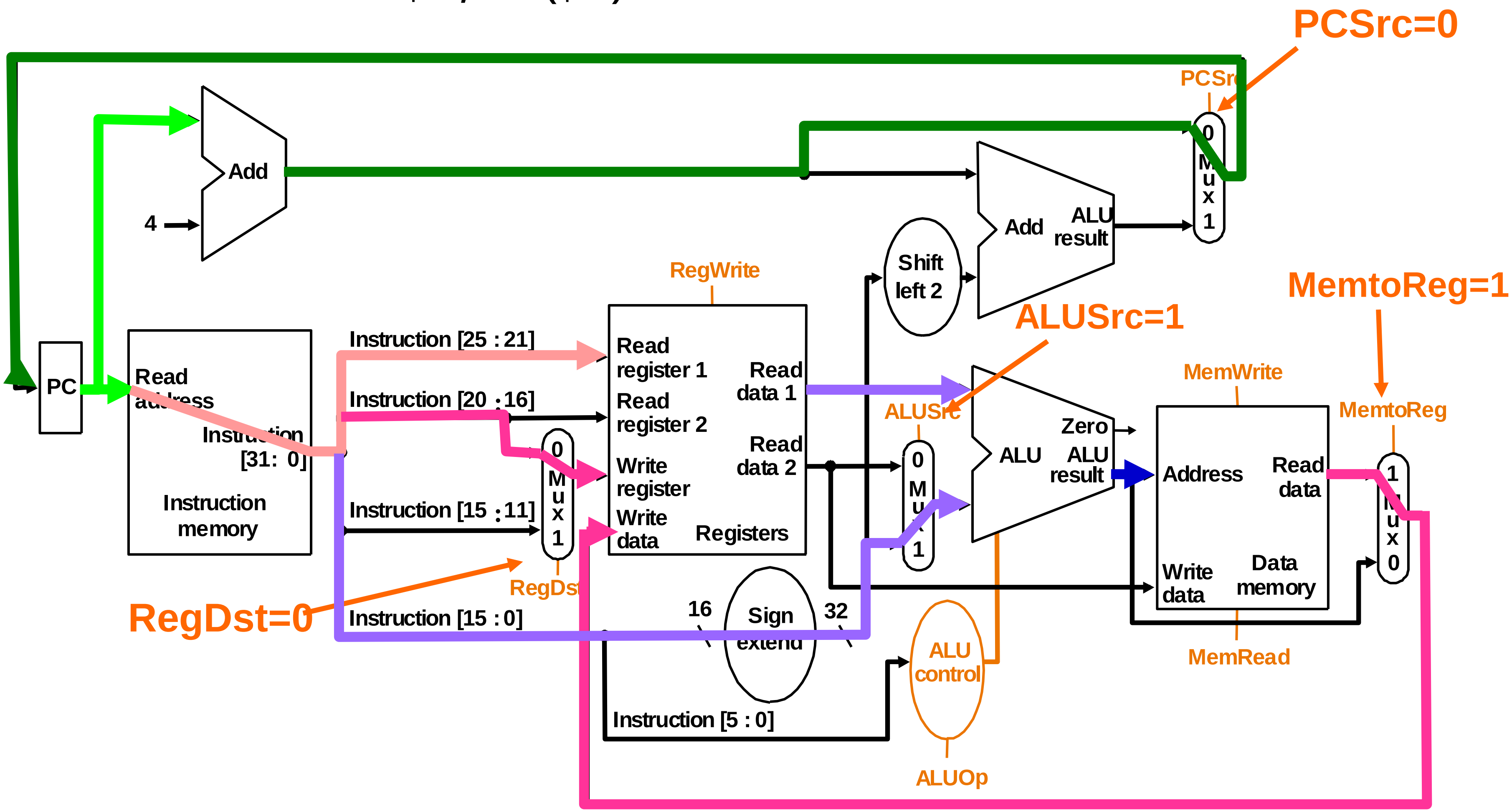
`RegDst`	1 to select `Rd`
`RegWrite`	1 to enable writing `Rd`
`ALUSrc`	0 to select `Rt` value from register file
`ALUOp`	*Dependent on operation* (see below)
`MemWrite`	0 to disable writing memory
`MemRead`	0 to disable reading memory
`MemtoReg`	0 to select ALU output to register
`PCSrc`	0 to select next `PC`

- The ALU requires control based on the op:

`add`	OP(add)	`and`	OP(and)
`sub`	OP(sub)	`or`	OP(or)
`slt`	OP(slt)		

I-format 'Load' instruction dataflow

lw $s0, 100($t0)



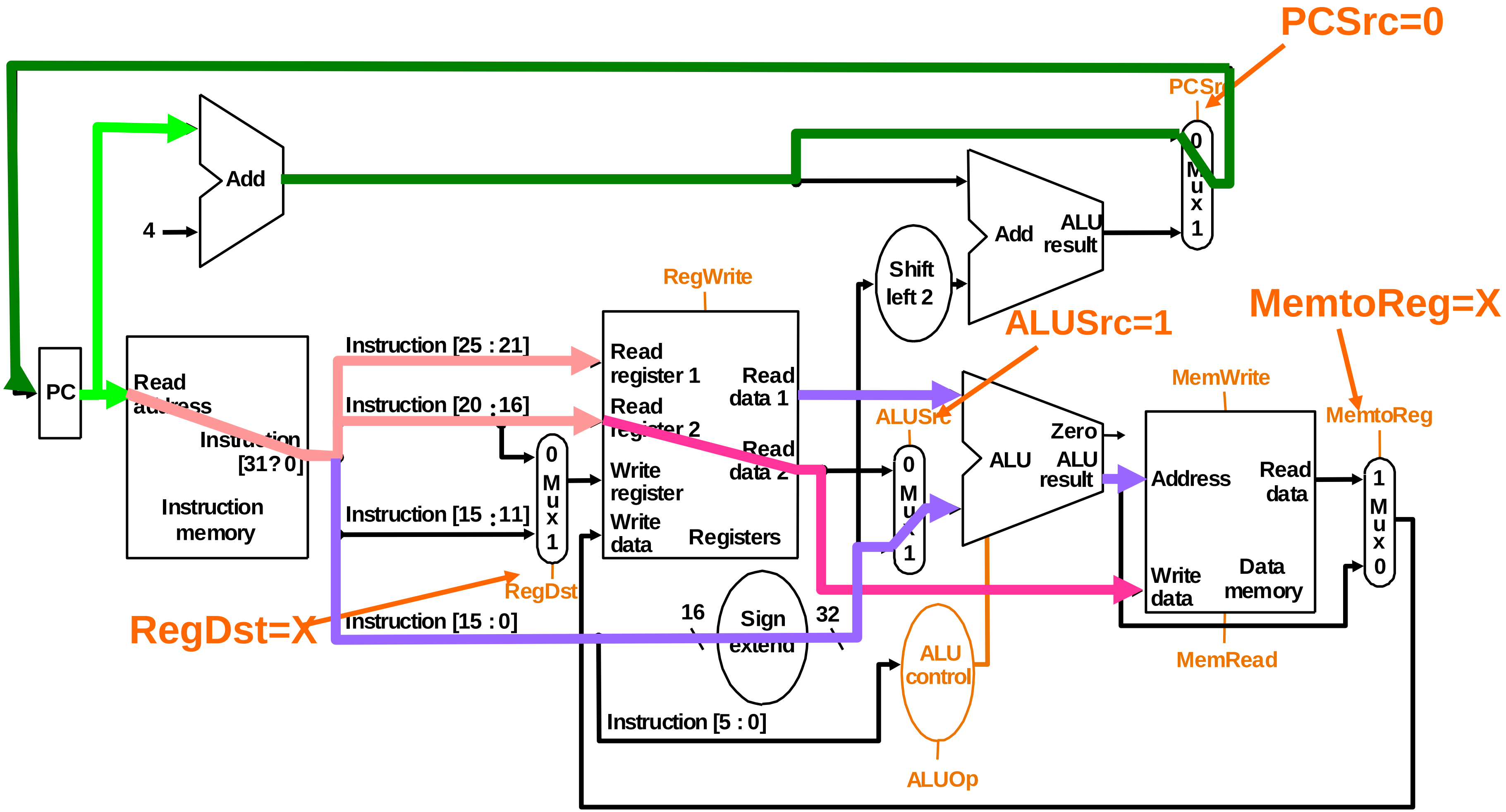
I-format Load Operation Control

- **We thus have the following control signals for these ops:**

`RegDst`	**0 to select `Rt`**
`RegWrite`	**1 to enable writing `Rt`**
`ALUSrc`	**1 to select immediate field value from instruction**
`ALUOp`	**add**
`MemWrite`	**0 to disable writing memory**
`MemRead`	**1 to enable reading memory**
`MemtoReg`	**1 to select memory output to register**
`PCSrc`	**0 to select next `PC`**

I-format ‘Store’ instruction dataflow

sw $s0, 100($t0)



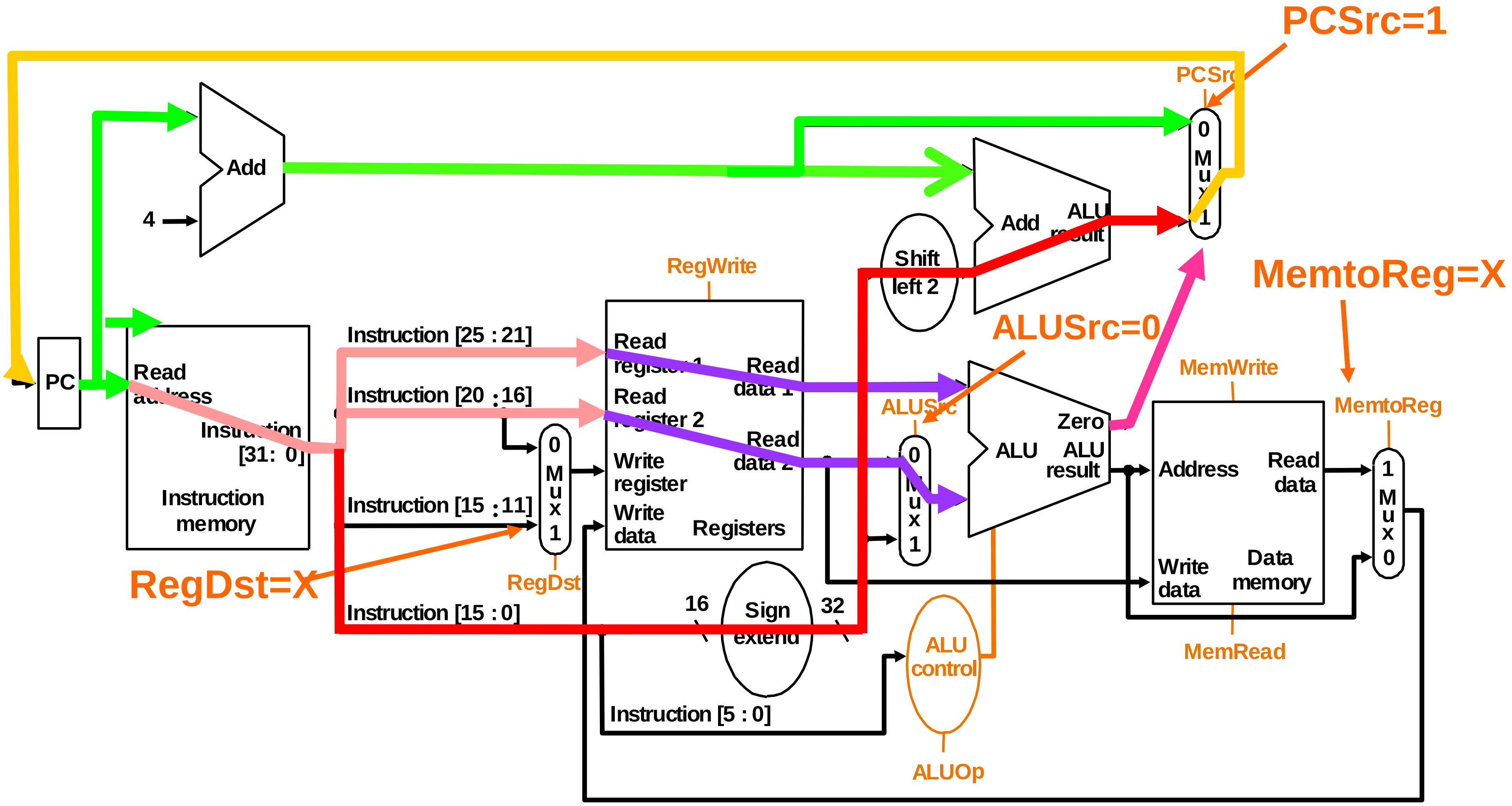
I-format Store Operation Control

- We thus have the following control signals for these ops:

`RegDst`	X (don’t care)
`RegWrite`	0 to disable writing a register
`ALUSrc`	1 to select immediate field value from instruction
`ALUOp`	add
`MemWrite`	1 to enable writing memory
`MemRead`	0 to disable reading memory
`MemtoReg`	X (don’t care)
`PCSrc`	0 to select next `PC`

I-format ‘Beq’ instruction dataflow

beq $s0, $s1, loop



I-format Branch Operation Control

- **We thus have the following control signals for these ops:**

`RegDst`	**X (don’t care)**
`RegWrite`	**0 to disable writing a register**
`ALUSrc`	**0 to select `Rt` value from register file**
`ALUOp`	**sub**
`MemWrite`	**0 to disable writing memory**
`MemRead`	**0 to disable reading memory**
`MemtoReg`	**X (don’t care)**
`PCSrc`	**zero AND branch**

Single-cycle Control Signals Summary

- From the preceeding analyses, we have the following control signals required:

Signal	R-fmt	I-fmt (lw)	I-fmt (sw)	I-fmt (beq)
RegDst	1	0	x	x
RegWrite	1	1	0	0
ALUSrc	0	1	1	0
ALUOp	OP	add	add	sub
MemWrite	0	0	1	0
MemRead	0	1	0	0
MemtoReg	0	1	x	x
PCSrc	0	0	0	zero

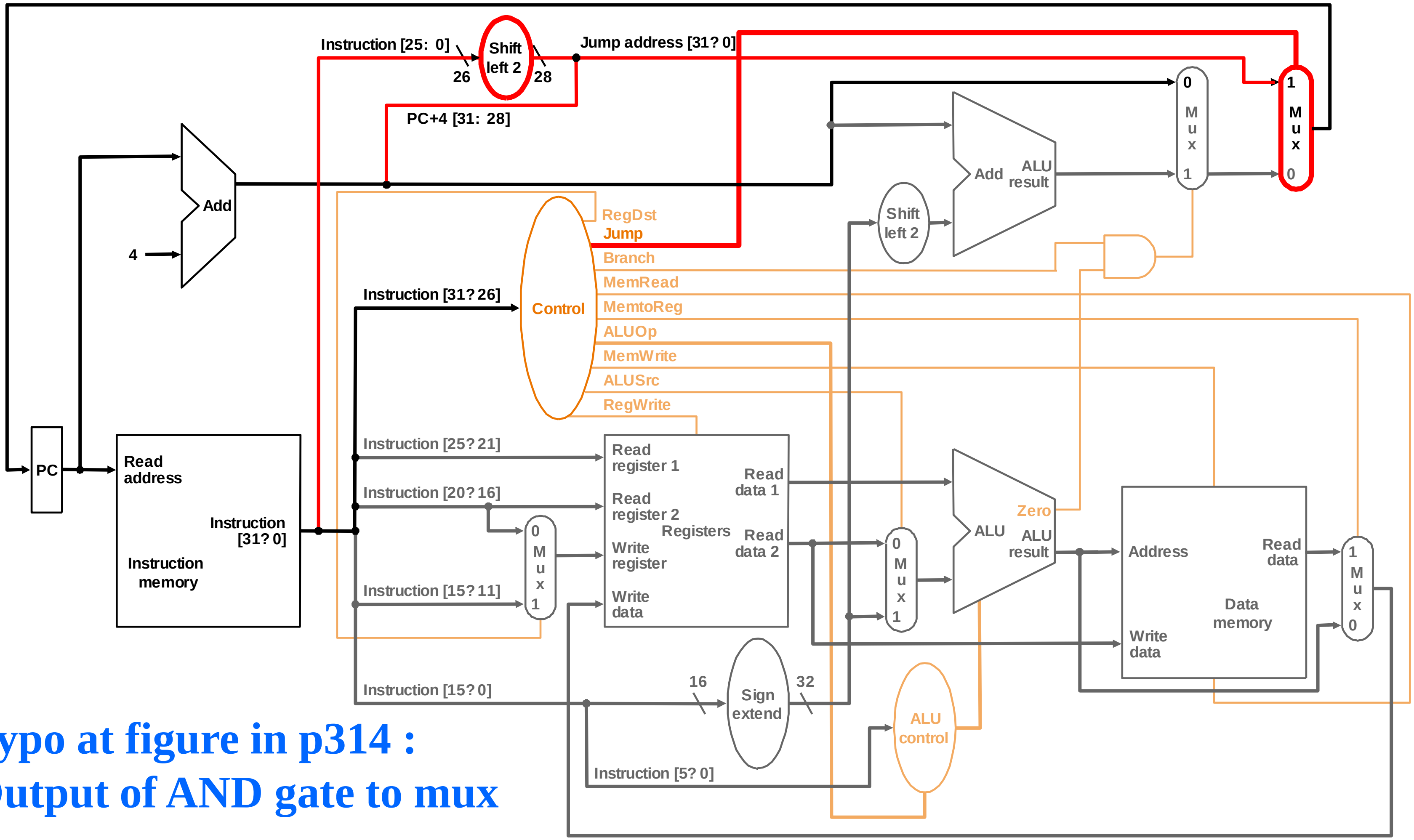
- Note that the only time that the PCSrc signal is active is during a branch, thus

PCSrc = (Operation is a branch) $\wedge$ zero

Adding Jump Implementation ?

000010	address(26-bits)

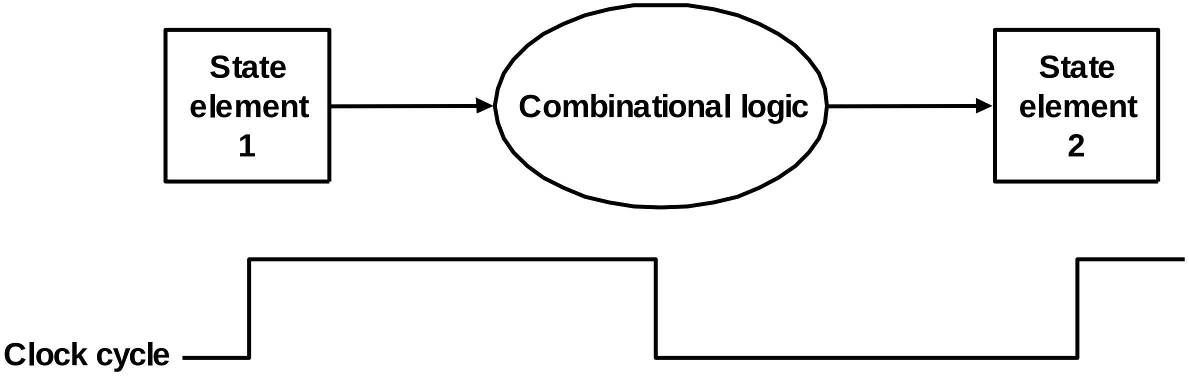
Target = PC(31:28) && (address << 2)



Typo at figure in p314 :
Output of AND gate to mux

Our Single Cycle Control Structure

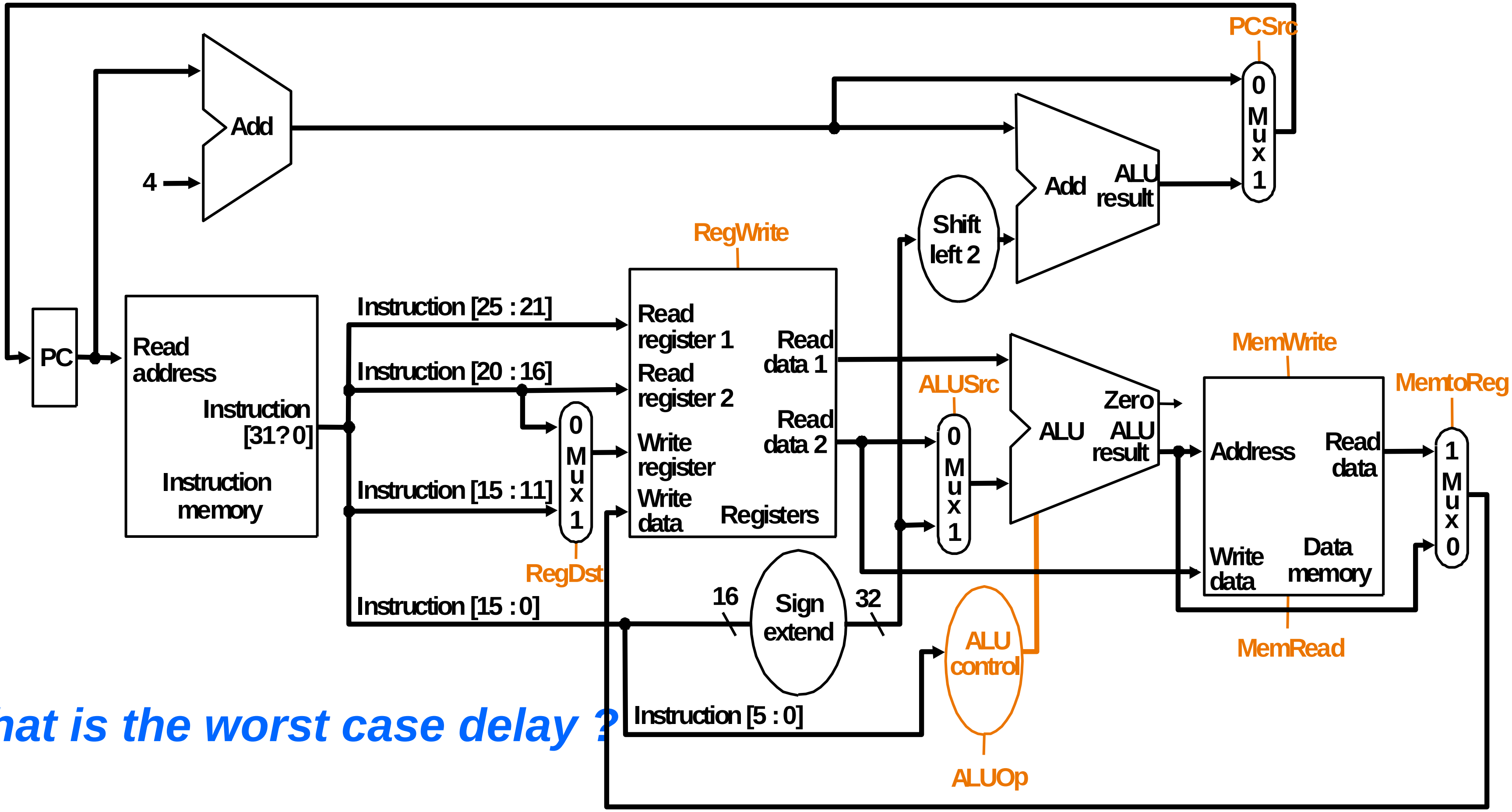
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer right away
  - We use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

Single Cycle Implementation Performance

- **Calculate cycle time assuming negligible delays except:**
  - **memory (200ps), ALU and adders (100ps), register file access(50ps)**



What is the worst case delay ?

Single Cycle Implementation Performance

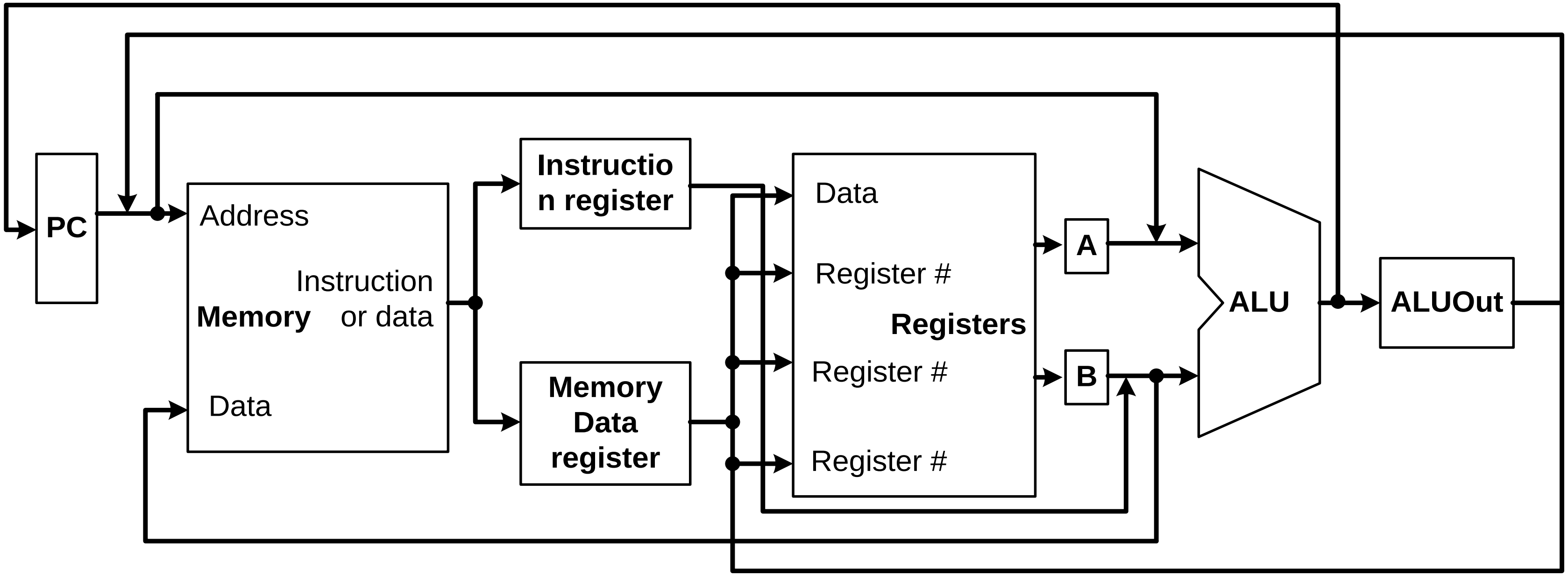
instruction class	Functional units used by the instruction class				
ALU type	Instr. Fetch (200ps)	register access (50ps)	ALU (100ps)		register access (50ps)
Load word	Instr. Fetch (200ps)	register access (50ps)	ALU (100ps)	Memory access(200ps)	register access (50ps)
Store word	Instr. Fetch (200ps)	register access (50ps)	ALU (100ps)	Memory access(200ps)	
Conditional Branch	Instr. Fetch (200ps)	register access (50ps)	ALU (100ps)		
Jump	Instr. Fetch (200ps)				

- The Clock Cycle time with single clock will be determined by the longest instruction, which is 600 ps
- CPI = 1
- Execution Time = instruction# * CPI * Clock Cycle Time

= instruction# * 600 ps

Where we are headed

- **Single Cycle Problems:**
  - **what if we had a more complicated instruction like floating point?**
  - **waste of chip area**
- **One Solution:**
  - **Make cycle time smaller & have different instructions take different numbers of cycles ➔ a multicycle datapath:**



Summary

- **We verified the MIPS datapath built in previous section executes MIPS instruction correctly.**
- **Different instruction takes different amount of time to be executed, which gives a room for performance improvement.**

1. Review

    - Logic design review

    - MIPS instruction review

2. Building Datapath

3. A Simple Implementation Scheme

4. **Multicycle Implementation**

Introduction

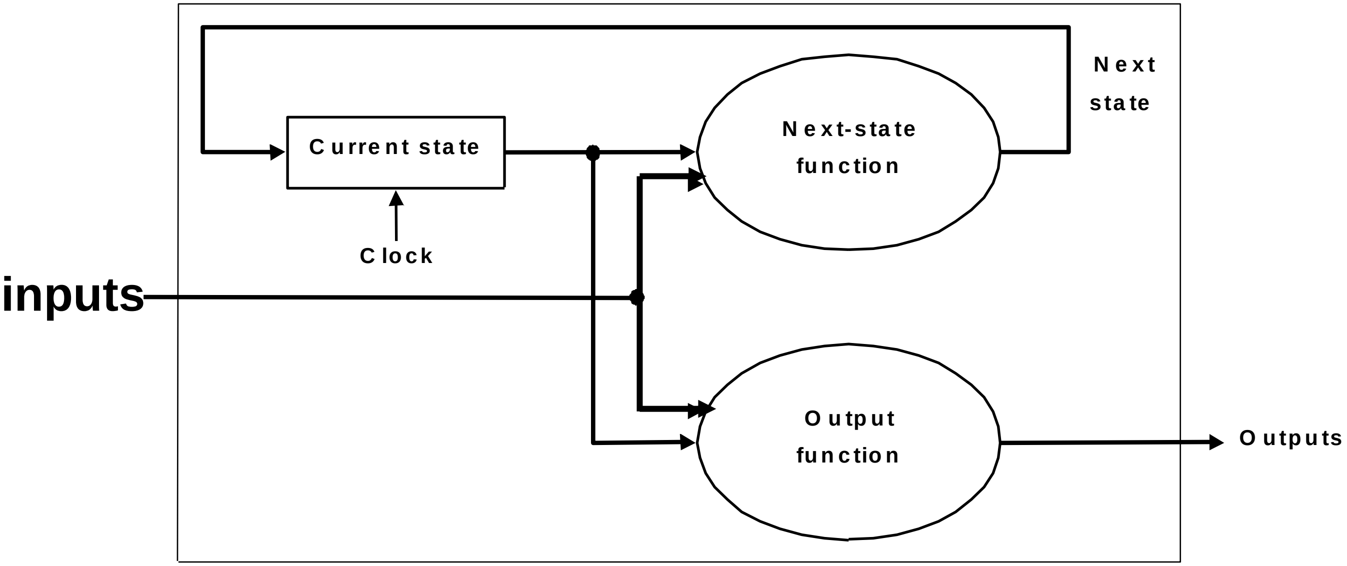
- In this section the datapath in previous section is modified to build multi-clock cycle datapath.
- In multicycle datapath, it takes several clock cycles to execute one instruction.
- We need extra components to build multicycle datapath.

Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- We'll use a finite state machine for control

Review: finite state machines

- **Finite state machines:**
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)



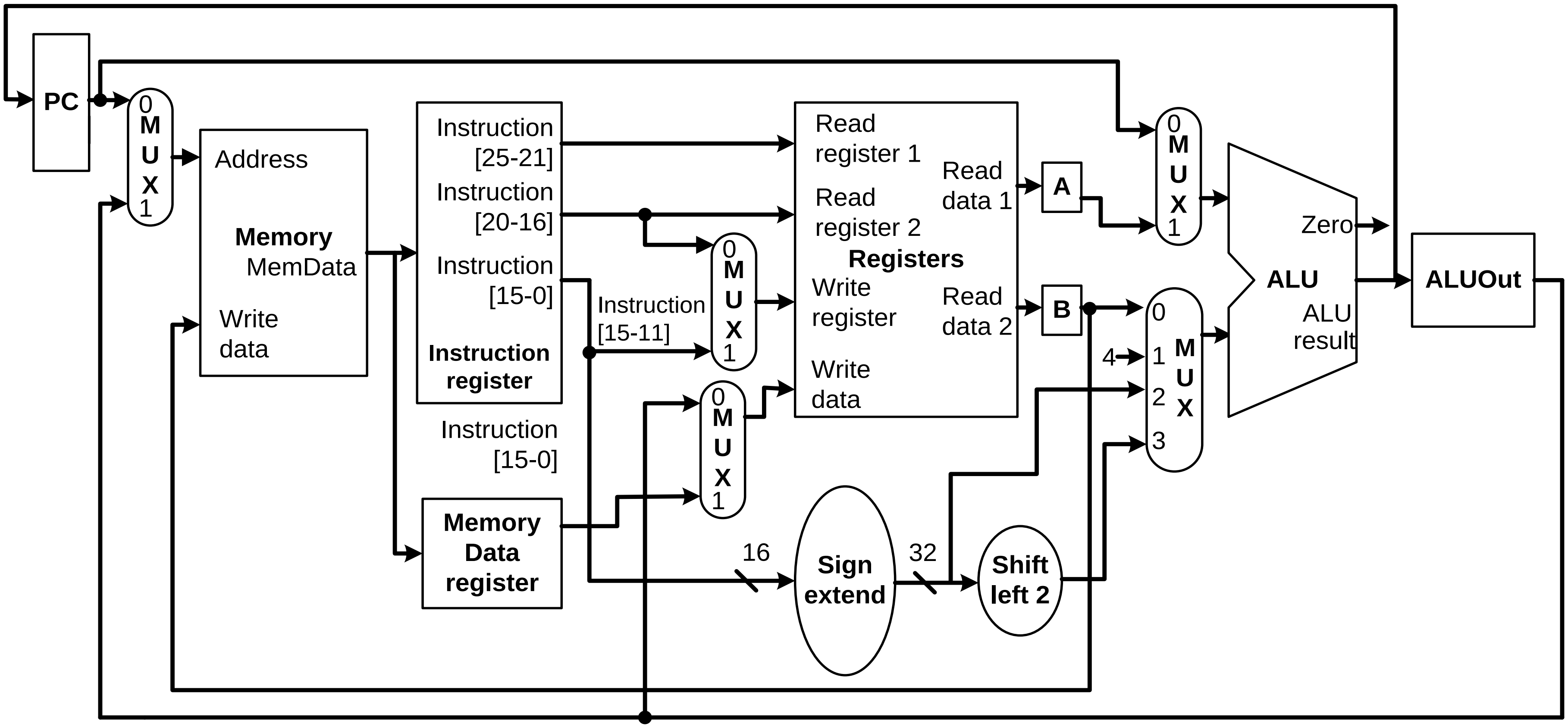
  - We'll use a Moore machine (output based only on current state)

MultiCycle Datapath for MIPS Basic Instructions

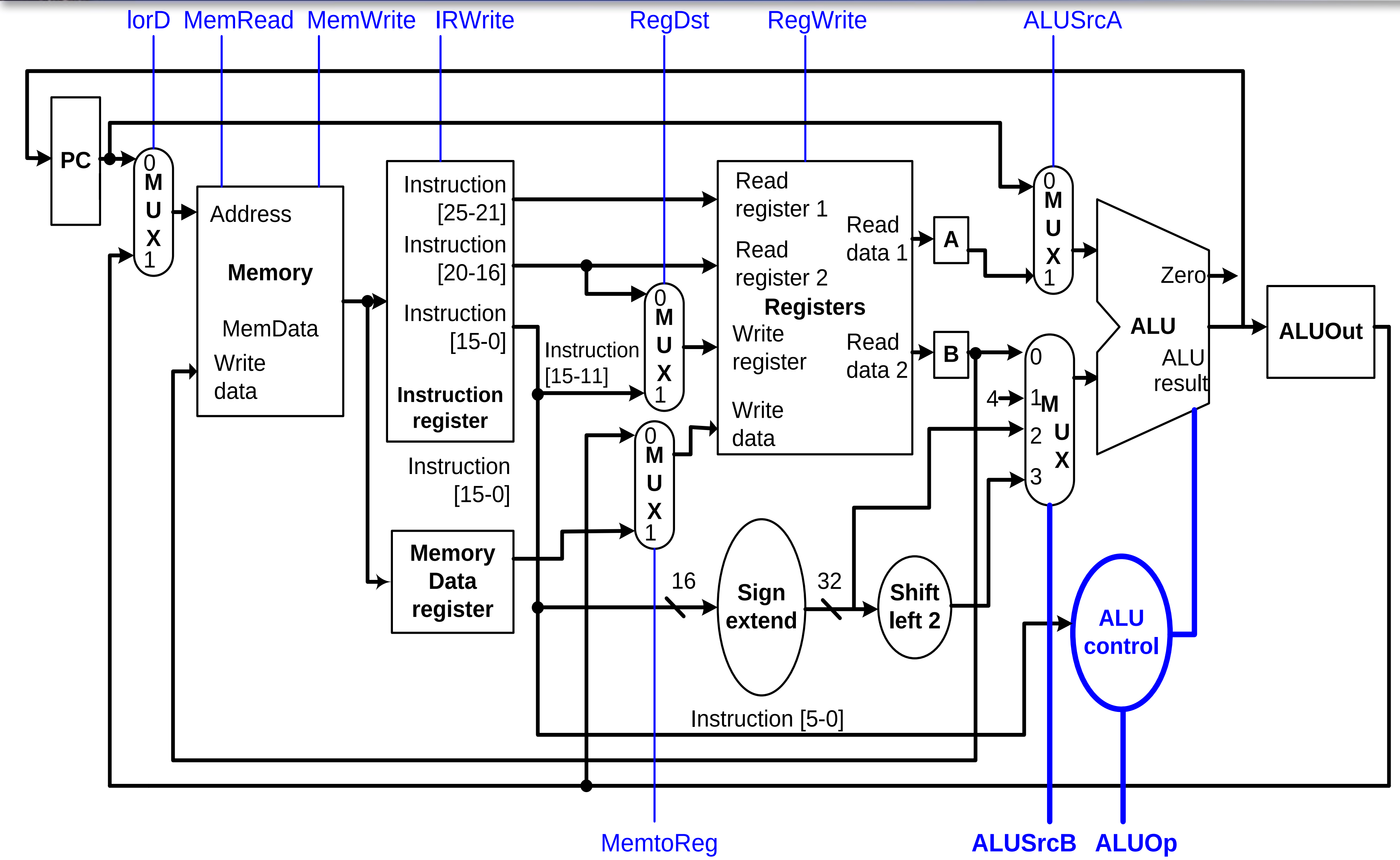
- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit

- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional internal registers

MultiCycle Datapath for MIPS
Basic Instructions



Multicycle datapath



Comparing simplecycle datapath with multicycle datapath

- **Single Memory**
- **Single ALU**
- **Several registers are added to store intermediate results**

  - Instruction Register

  - A, B

  - ALUout

  - Memory Data Register   cf) Memory Address Register
- **We need more multiplexers to share functional units.**

  - Mux in front of memory

  - Muxes in front of ALU

Five Execution Steps

1. Instruction Fetch

2. Instruction Decode and Register Fetch

3. Execution, Memory Address Computation, or Branch Completion

4. Memory Access or R-type instruction completion

5. Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- **Use PC to get instruction and put it in the Instruction Register.**
- **Increment the PC by 4 and put the result back in the PC.**
- **Can be described succinctly using RTL "Register-Transfer Language"**

```
IR <= Memory[PC];
PC <= PC + 4;
```

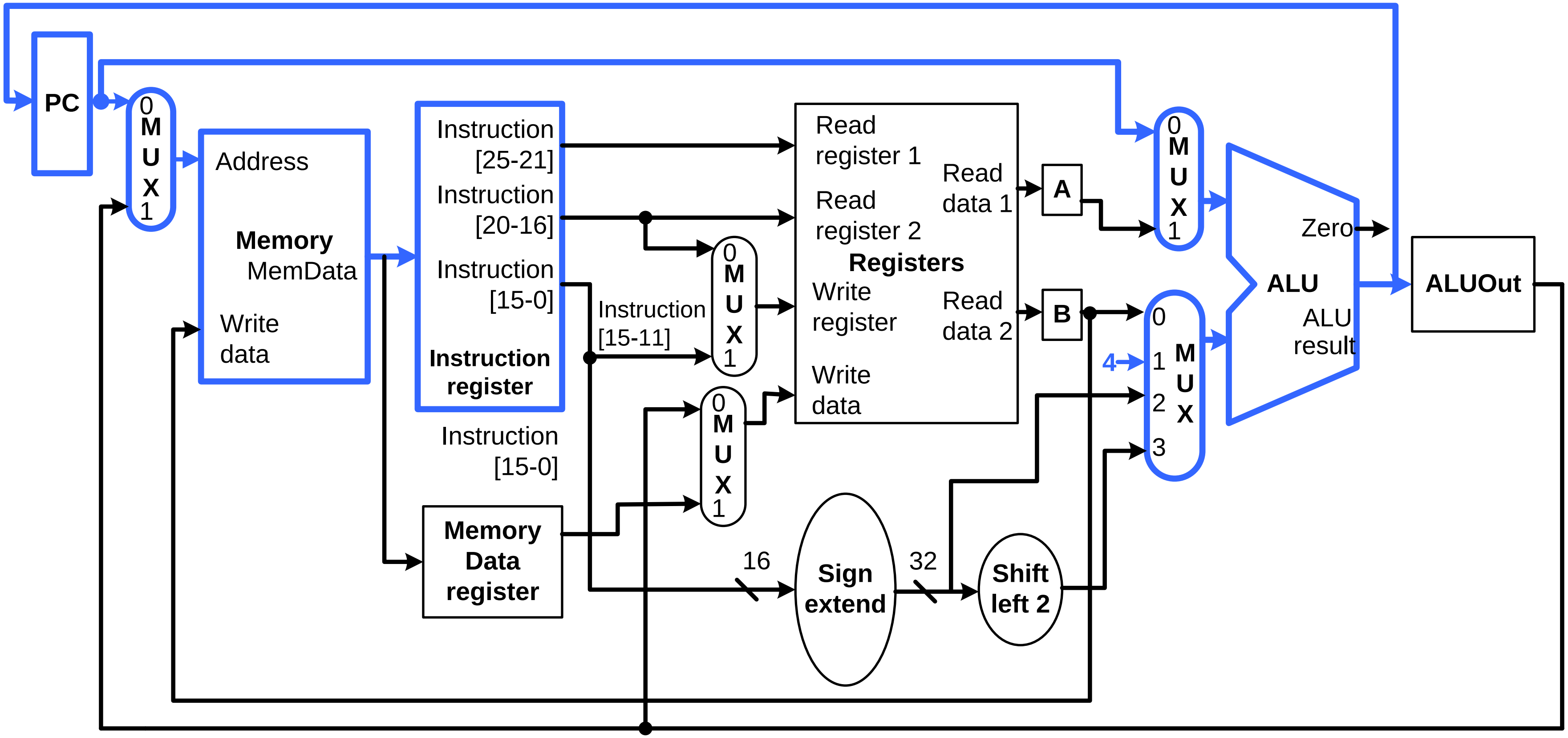
Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Instruction Fetch

- **RTL Description**

```
IR ← Memory[PC];
PC ← PC + 4;
```



Step 2: Instruction Decode and Register Fetch

- **Read registers rs and rt in case we need them**
- **Compute the branch address in case the instruction is a branch**
- **RTL:**

```
A <= Reg[IR[25-21]];
B <= Reg[IR[20-16]];
ALUOut <= PC + (sign-extend(IR[15-0]) << 2);
```

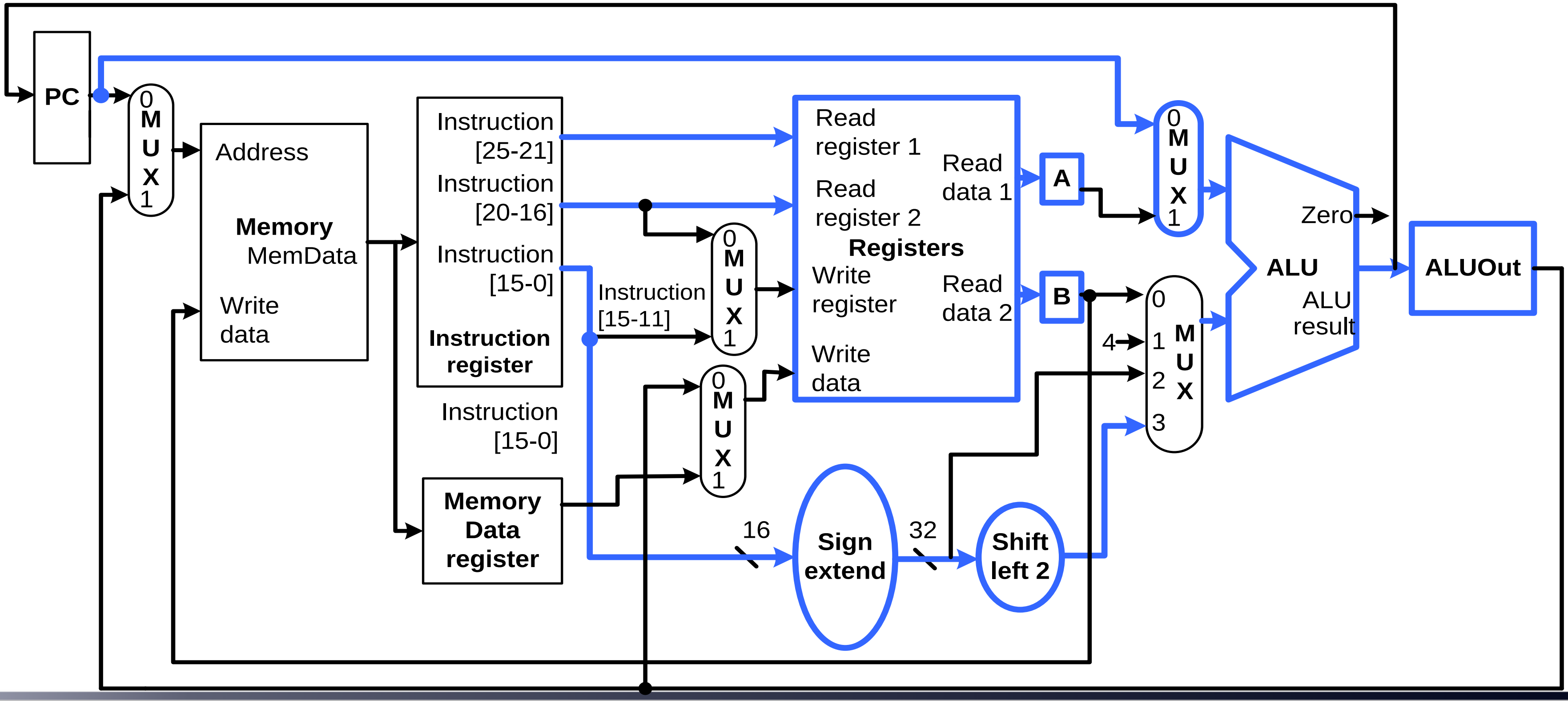
- **We aren't setting any control lines based on the instruction type**
  (we are busy "decoding" it in our control logic)

Instruction Decode and Register Fetch

- **RTL Description**

```
A ← Reg[IR[25-21]];
B ← Reg[IR[20-16]];
ALUOut ← PC + (sign-extend(IR[15-0]) << 2);
```

—The ALU is used to calculate the branch destination speculatively in case the operation is, in fact, a branch



Step 2: Instruction Decode and Register Fetch

- Computing the branch target address is not going to be used in next step if the instruction is not branch. However, it is not harmful doing so at this stage. : Optimistic approach

- Control signals are not determined by instruction in this step. (The control signals in step 1 & 2 are same regardless of instruction type.)

Step 3: Instruction dependent

- ALU is performing one of three functions, based on instruction type

- Memory Reference:

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

- R-type:

```
ALUOut <= A op B;
```

- Branch:

```
if (A==B) PC <= ALUOut;
```

Step 4: R-type or memory-access

- Loads and stores access memory

```
MDR <= Memory[ALUOut];
      or
Memory[ALUOut] <= B;
```

- R-type instructions completion

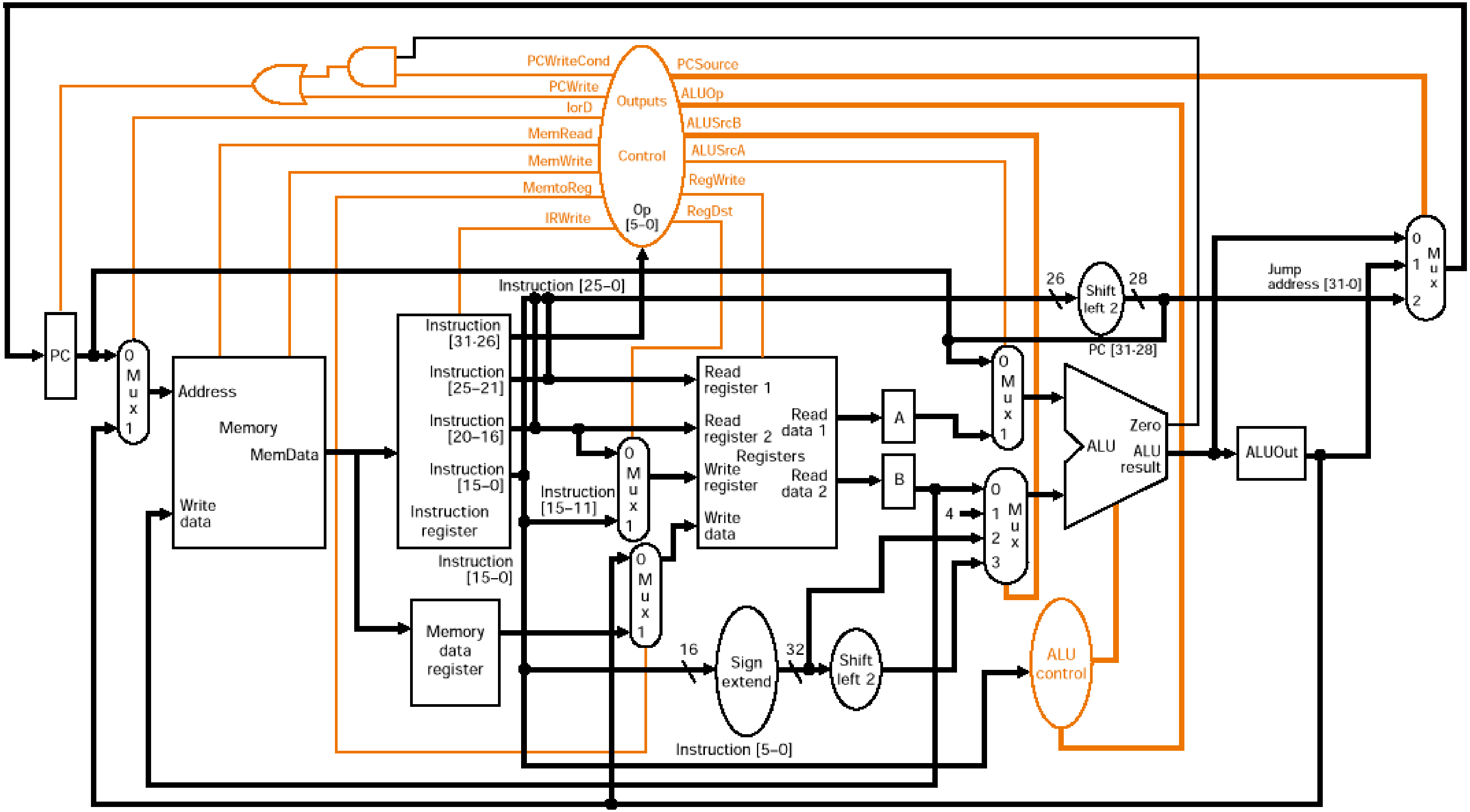
```
Reg[IR[15-11]] <= ALUOut;
```

Step 5: Memory read completion step

- `Reg[IR[20-16]]<= MDR;`

What about all the other instructions?

Multicycle Implementation
(including 'jump' instruction & Control Unit)



Summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode/register fetch	A = Reg [IR[25-21]]
	B = Reg [IR[20-16]]
	ALUOut = PC + (sign-extend-to32 (IR[15-0]) << 2)			

Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Summary

- In multicycle datapath it takes different number of clock cycles for different instructions.
- We need several registers to store intermediate results in multicycle datapath.
- We need only one memory and one ALU since they can be used for different purposes in different clock cycle.