Datapath

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

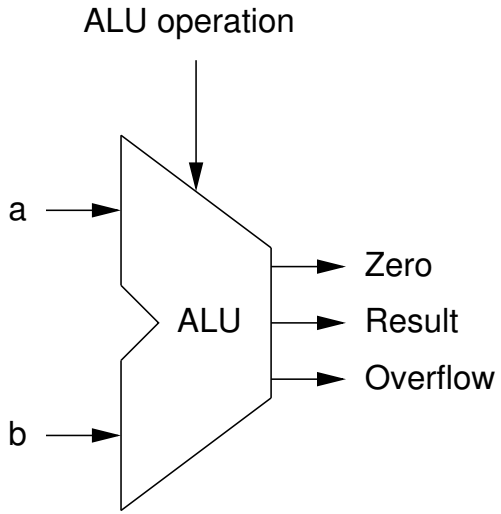
Draft

Introduction

Implementing a processor that supports the following instructions:

- **Memory-reference instructions**:
  - Load word (`lw`) and
  - Store word (`sw`)
- **Arithmetic-logical instructions**:
  - `add`,
  - `sub`,
  - `and`,
  - `or`, and
  - `slt`
- **Branch and Jump instructions**:
  - Branch on equal (`beq`) and
  - Jump (`j`)

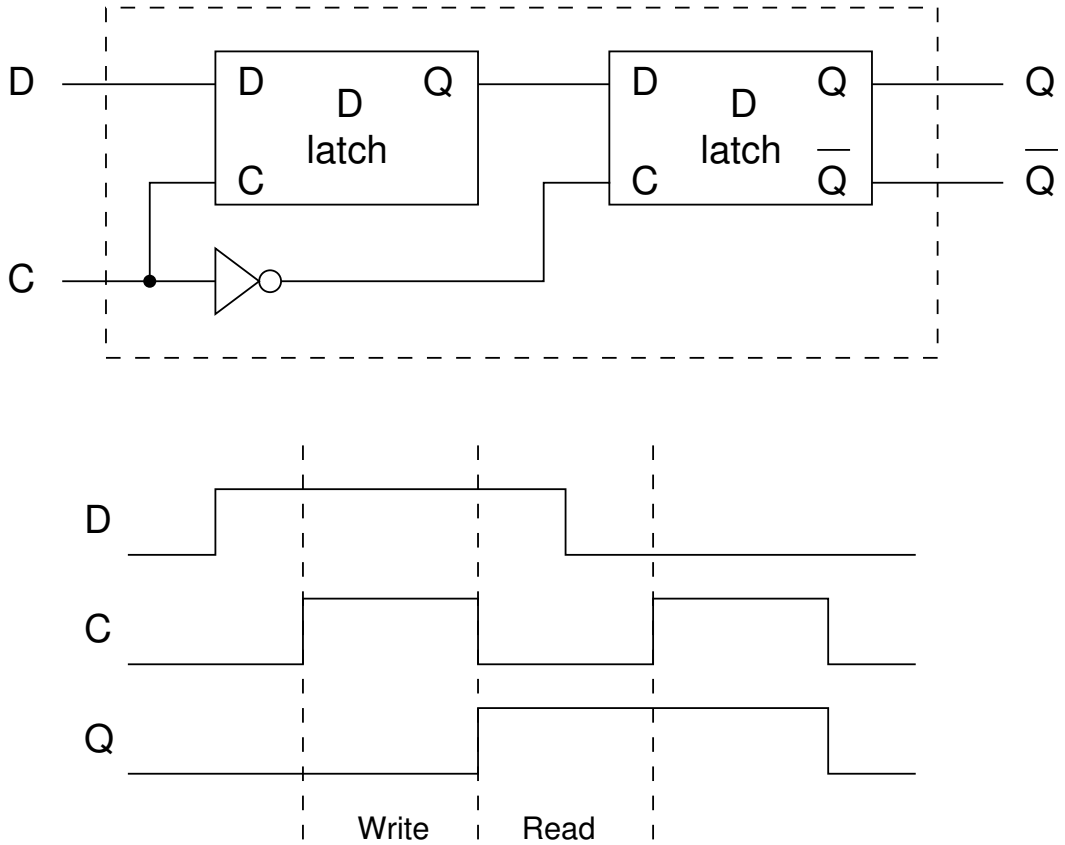
ALU



Operation	ALU operation		
	`Ainvert`	`Bnegate`	`Operation`
AND	0	0	00_2
OR	0	0	01_2
Addition	0	0	10_2
Subtraction	0	1	10_2
NOR	1	1	00_2
Set on Less Than	0	1	11_2
Branch	0	1	10_2

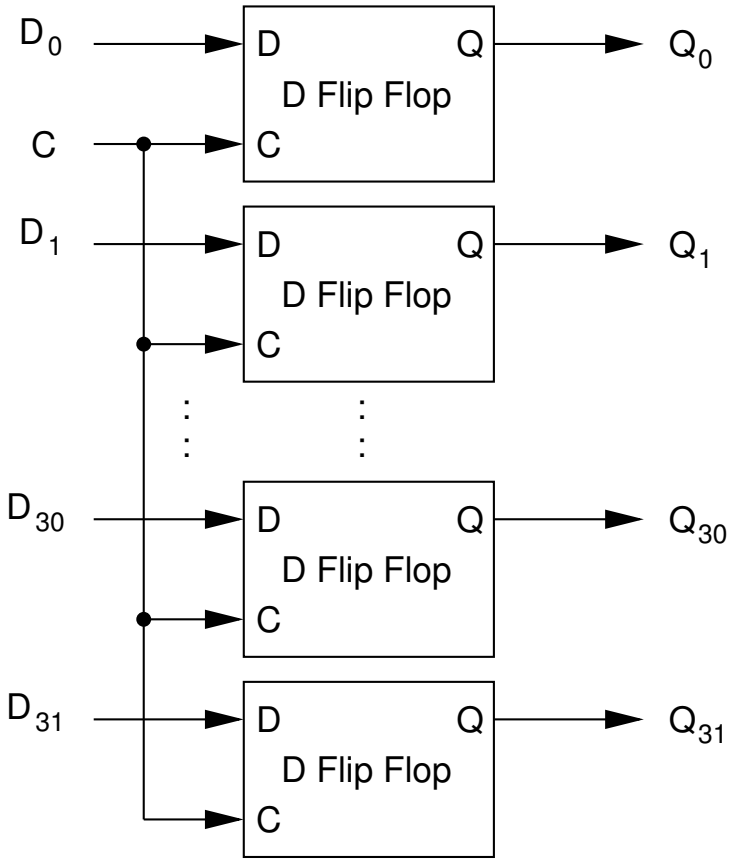
See logisim

D Flip Flop



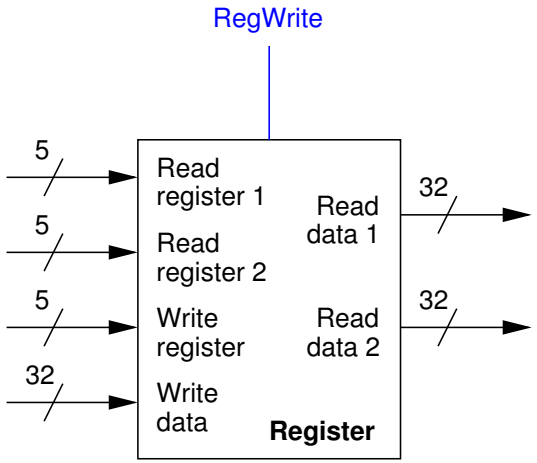
- When C is asserted (1), D is written to the first D latch
- When C is deserted (0), value store in the first D latch goes to the output
- See logisim

A 32-Bit Register



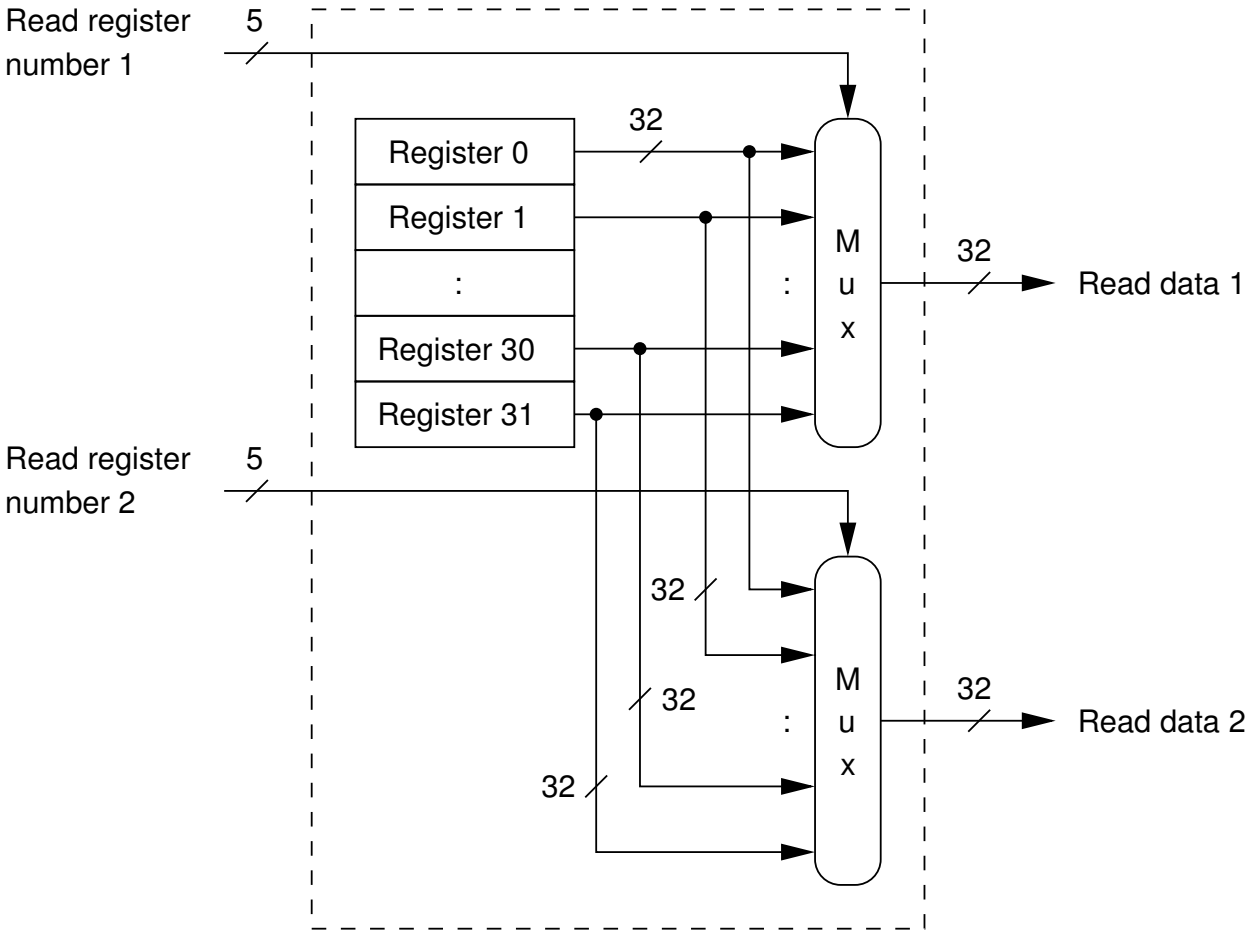
- Use one D flip flop to store 1-bit number
- We need 32 D flip flops to store 32-bits number
- See logisim

Register File



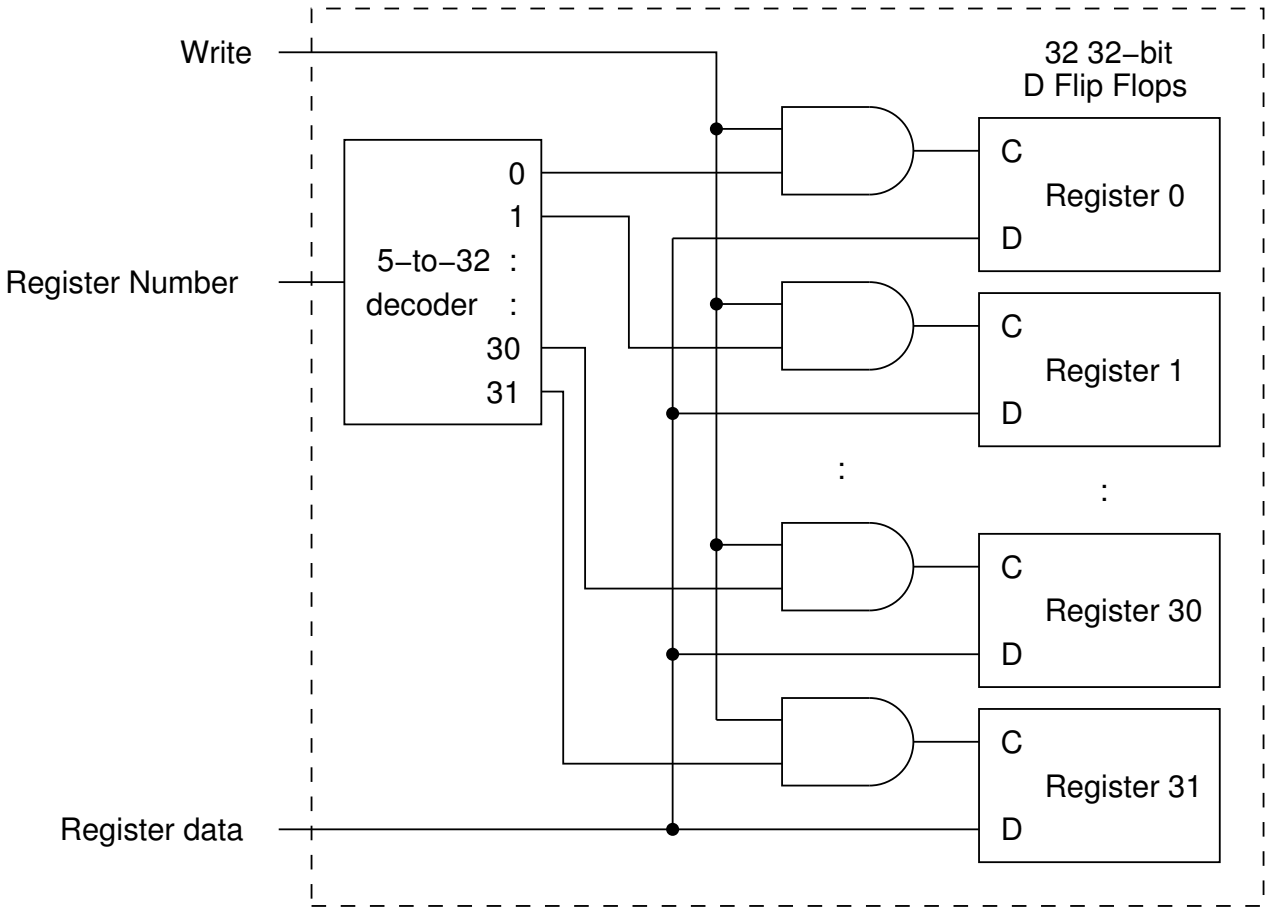
- Our MIPS has 32 general purpose registers
- Instruction such as `add` needs to read from two registers and write to one register
  - we need 5 bits to identify which register is the first operand
  - we need 5 bits to identify which register is the second operand
  - we need 5 bits to identify which register to be written
- The register file produces two 32-bit numbers as outputs and take one 32-bit number as an input data to write to the designated register
- Use `RegWrite` signal to enable write to a register

Output of A Register File



- All registers constantly produce their stored values
- Use two multiplexers to select which values to go out
- Each multiplexer has 32 32-bit inputs and one 32-bit output

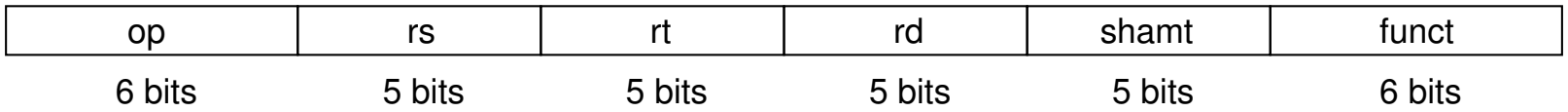
Write Port of A Register File



- Only one out of 32 registers will be written, if `write` is asserted.
- See logisim

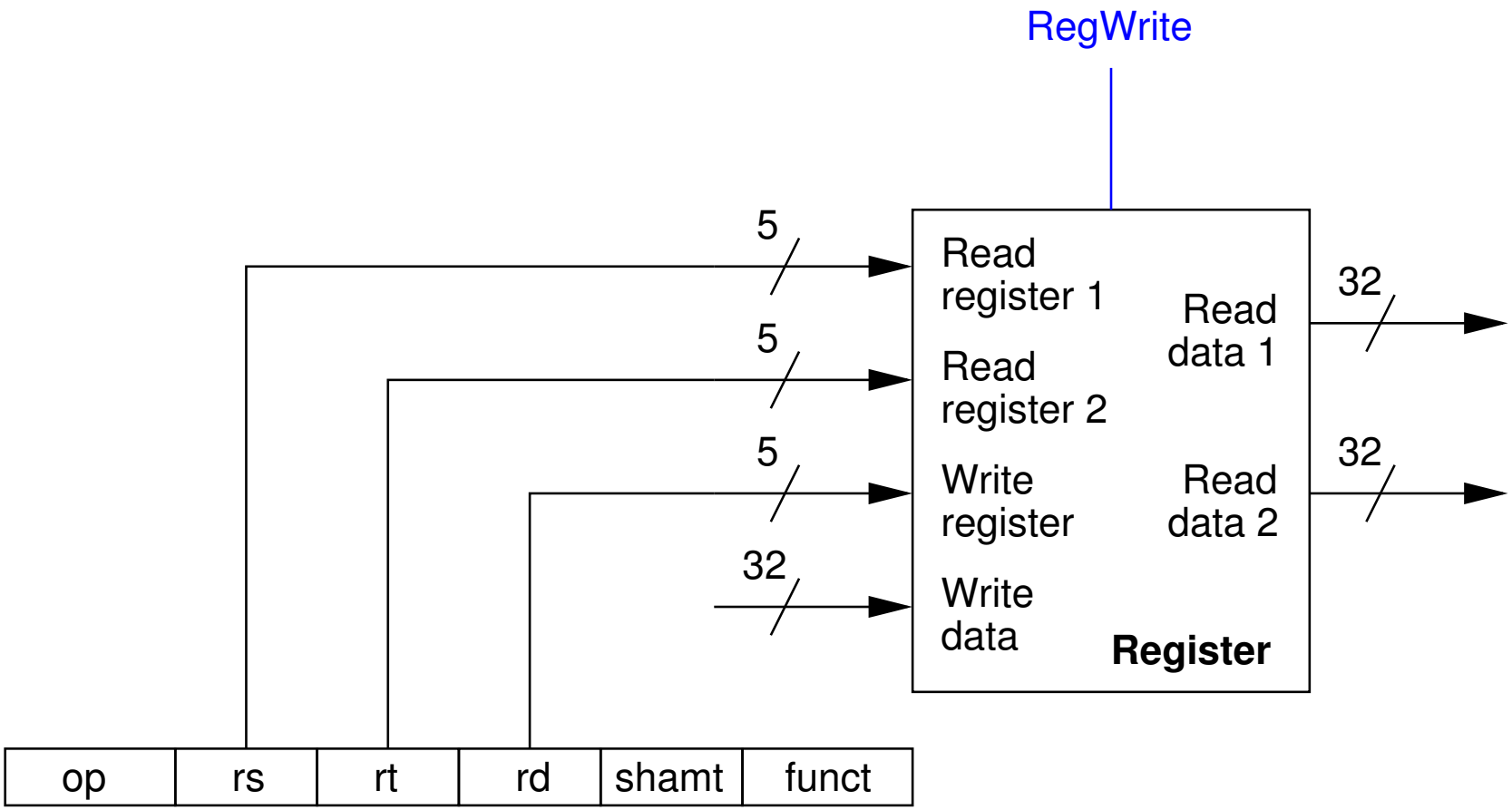
Arithmetic/Logic Instructions

- We will focus only on instructions `add`, `sub`, `and`, `or`, and `slt`
- All instructions use R-type



- For these instructions:
  - `rs` indicates the register number of the first operand
  - `rt` indicates the register number of the second operand
  - `rt` indicates the register number of the register to store the result (to be written)
- We can simply (for now)
  - wire the 5-bit `rs` to `Read Data 1`,
  - wire the 5-bit `rt` to `Read Data 2`, and
  - wire the 5-bit `rd` to `Write register`.

Arithmetic/Logic Instructions

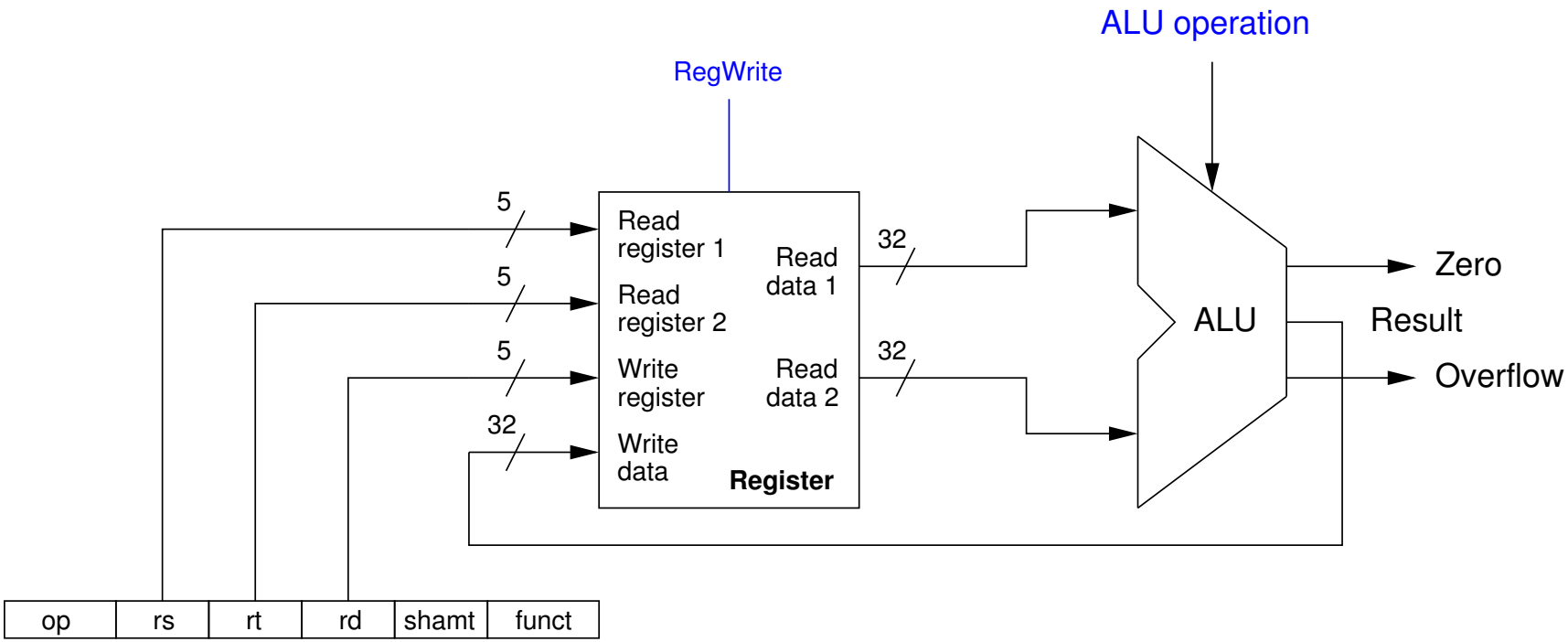


- Value stored in the register number designated by `rs` will be at `Read data 1`
- Value stored in the register number designated by `rt` will be at `Read data 2`

Arithmetic/Logic Instructions

- All arithmetic/logic instructions use ALU to calculate the result
- Operands for the ALU come from:
  1. Value stored in register number designated by `rs` which will be at `Read data 1`
  2. Value stored in register number designated by `rt` which will be at `Read data 2`
- The result produced by ALU must be stored in a register number designated by `rd`
- The `RegWrite` signal should be asserted (1) and then deserted (0)
  - When `RegWrite` is 1, it allows the result to be written to a designated register
  - Change `RegWrite` to 0 to stop writing process

Arithmetic/Logic Instructions



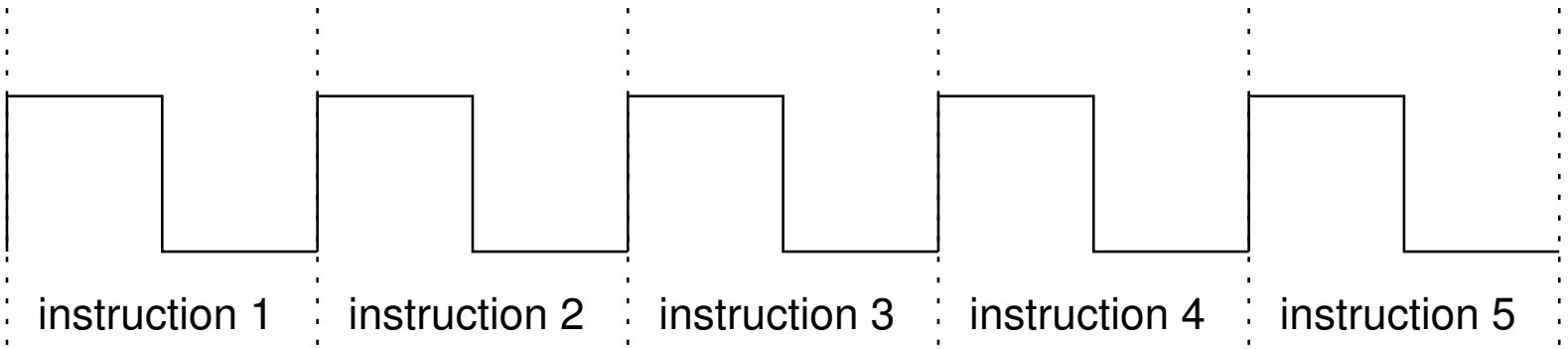
- Note that there is no feedback or infinite loop
  - We use D Flip Flop which is written when `RegWrite` is 1
  - New value will come out when `RegWrite` is 0

Clocking Methodology

- Defines when signals can be written and when they can be read
- For simplicity, we use **edge-triggered clocking** methodology
- Allow memory read and write in the same clock cycle.
- Data will be written only when clock is asserted (changed from 0 to 1)
- Output from memory will be updated only on a clock is deserted (changed from 1 to 0)
- Thus, no feedback

Clocking Methodology

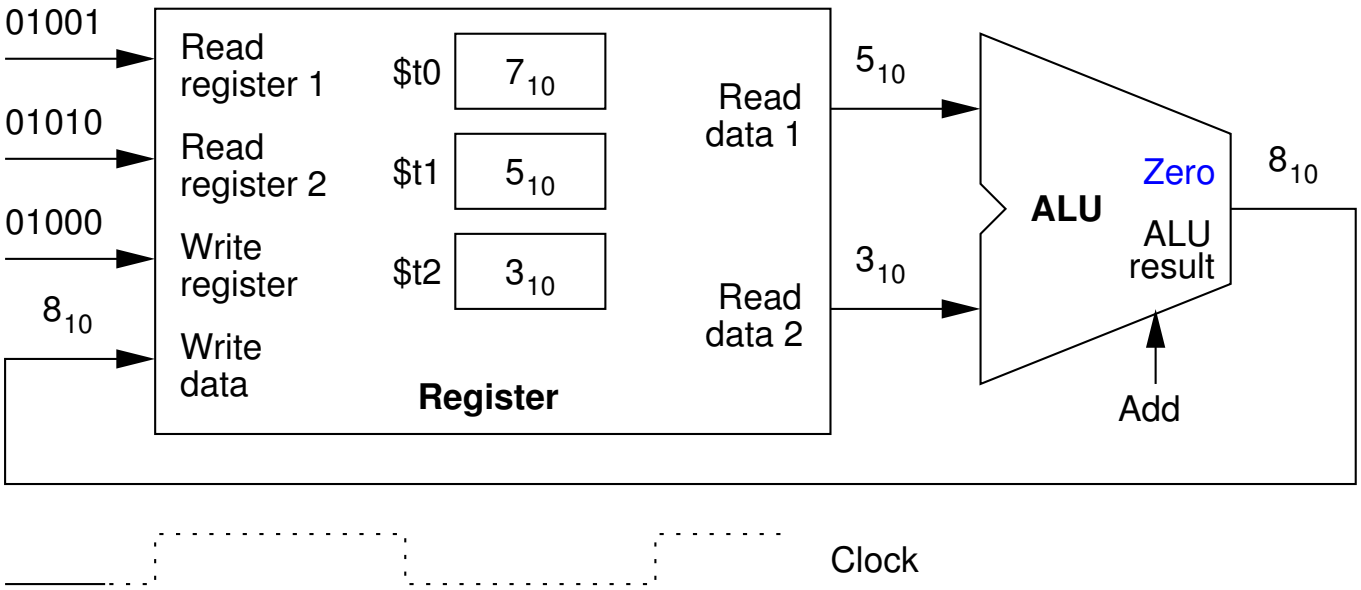
- Our design, every instruction will take one clock cycle to execute



- Every clock cycle, a new instruction is loaded from instruction memory
- The location of the instruction to be loaded is specified by program counter (PC)

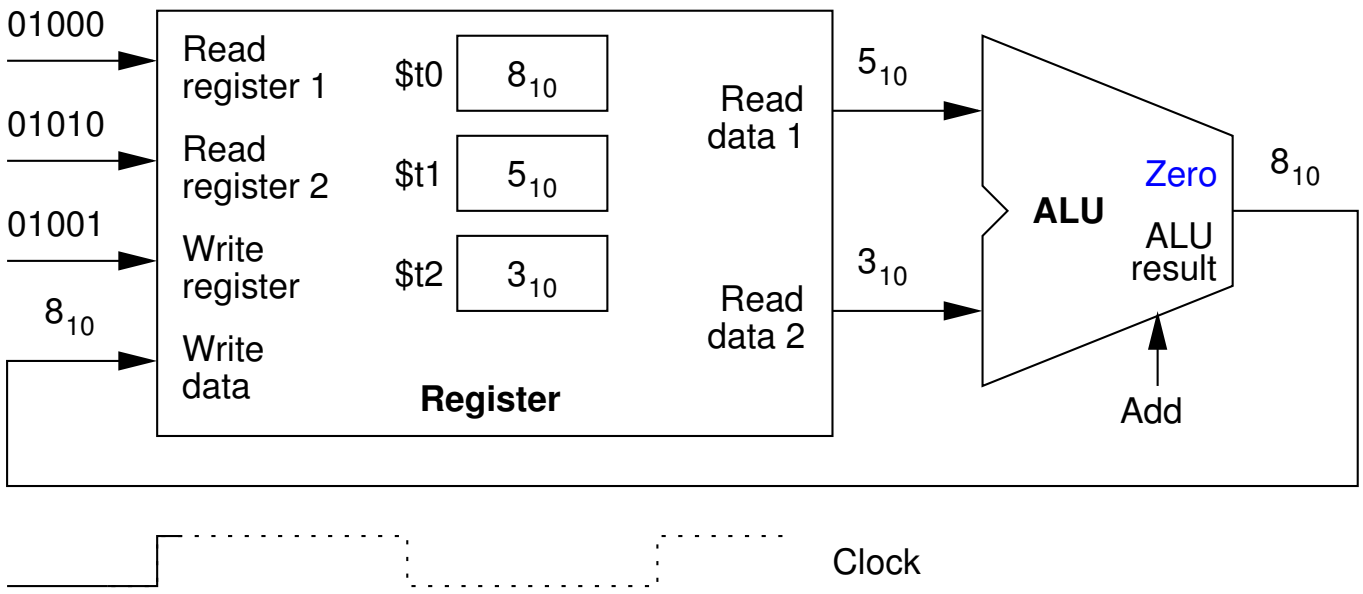
Clocking Methodology

* `add $t0, $t1, $t2`
  `add $t1, $t0, $t2`



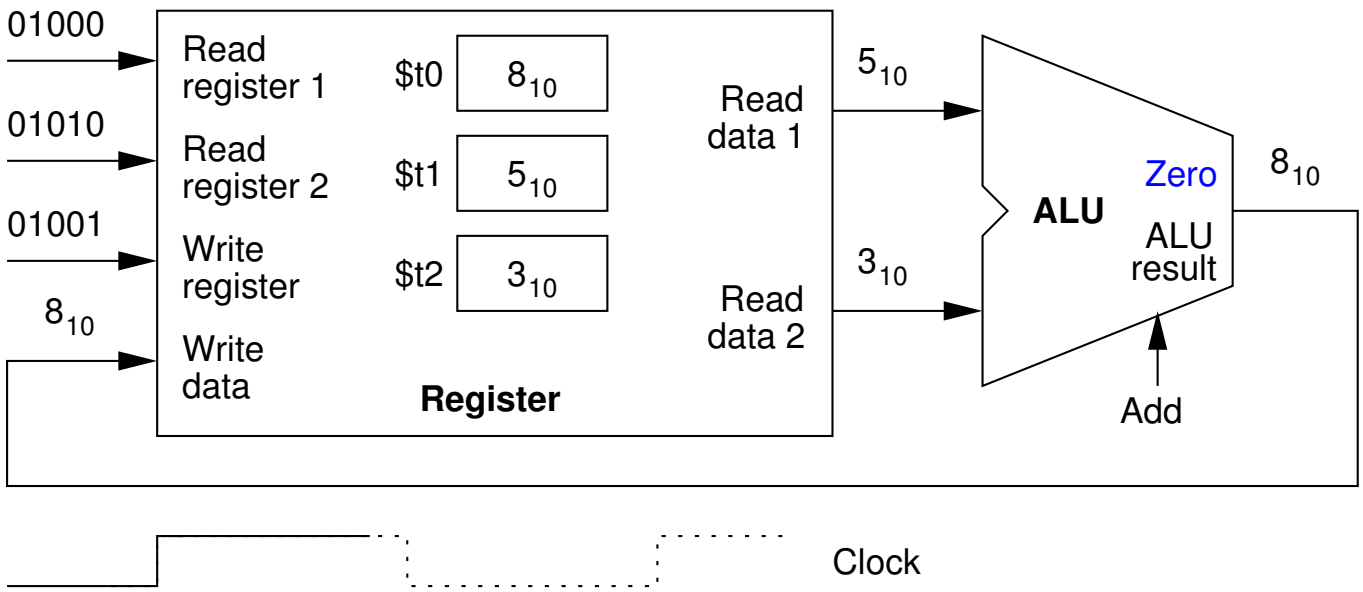
Clocking Methodology

```
  add $t0, $t1, $t2
* add $t1, $t0, $t2
```



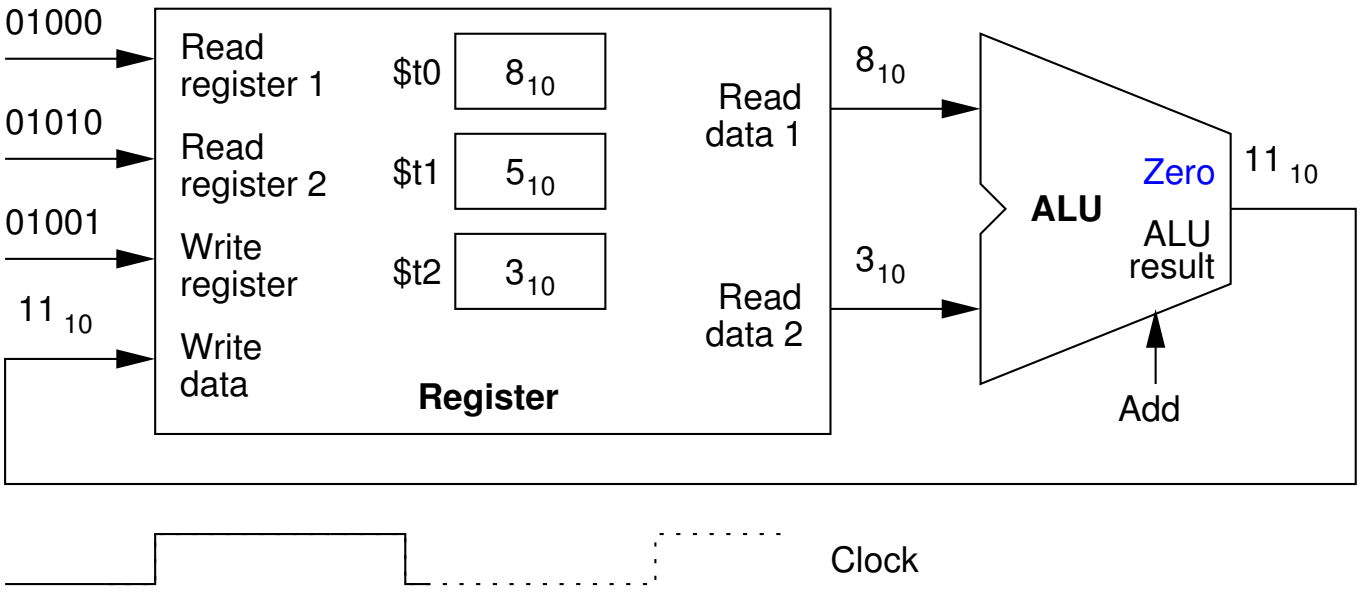
Clocking Methodology

```
  add $t0, $t1, $t2
* add $t1, $t0, $t2
```



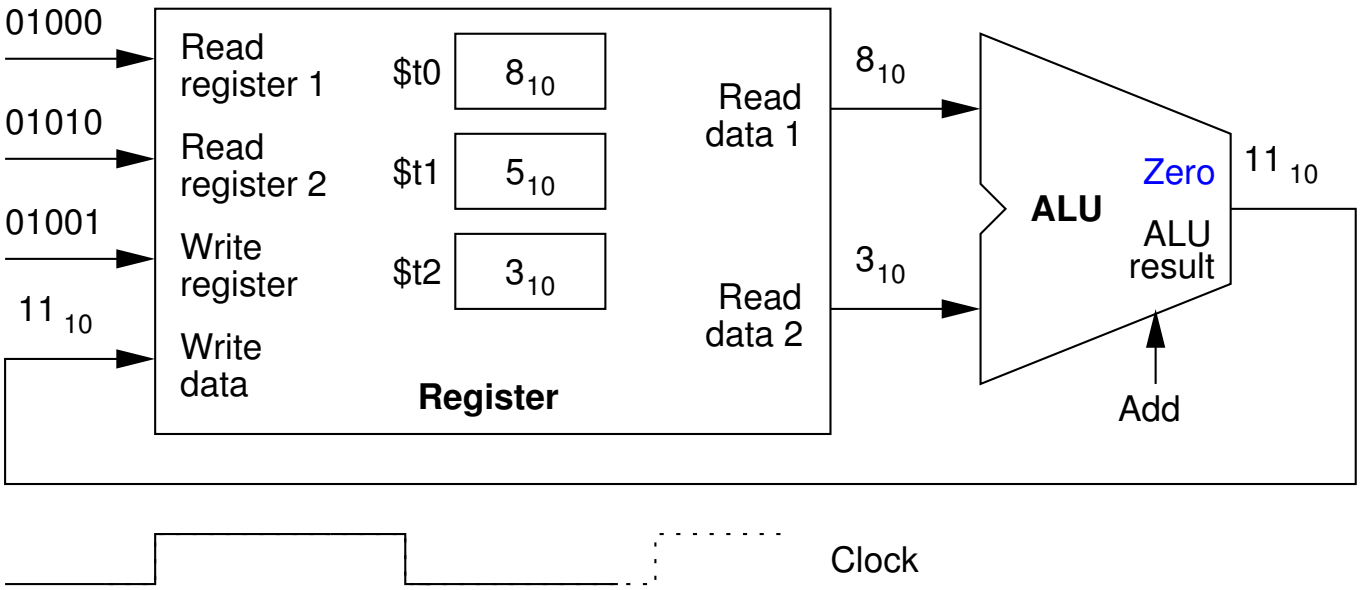
Clocking Methodology

```
  add $t0, $t1, $t2
* add $t1, $t0, $t2
```



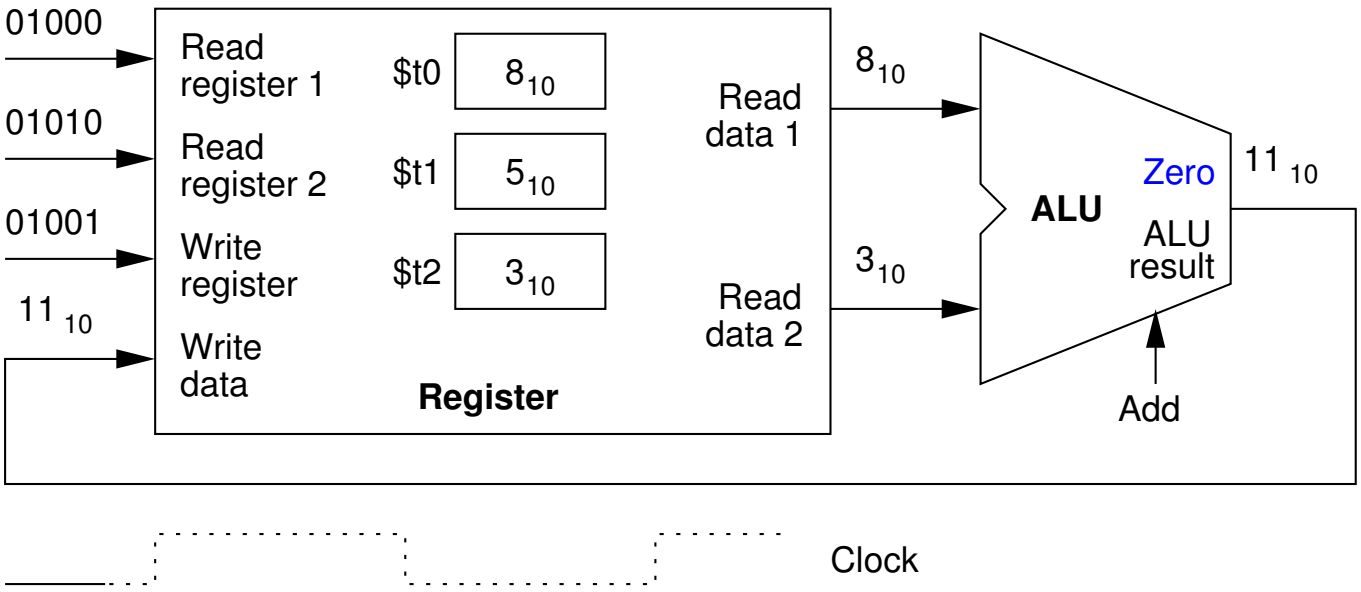
Clocking Methodology

```
  add $t0, $t1, $t2
* add $t1, $t0, $t2
```



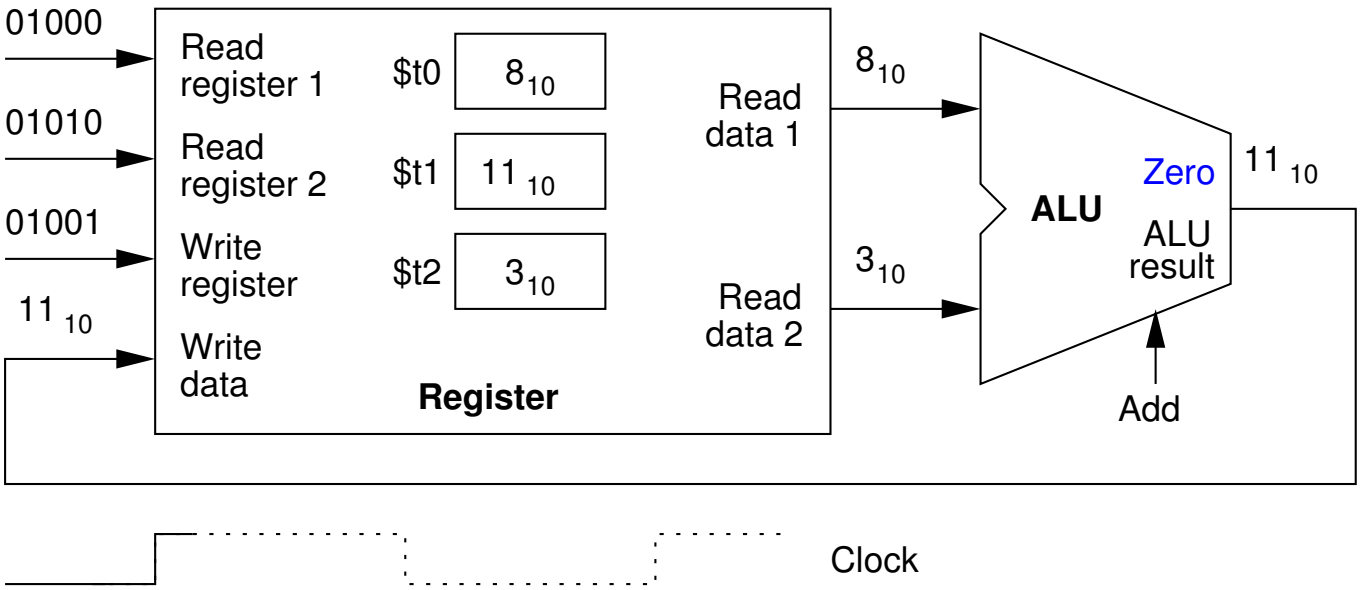
Clocking Methodology

```
* add $t1, $t0, $t2
  add $t0, $t0, $t1
```



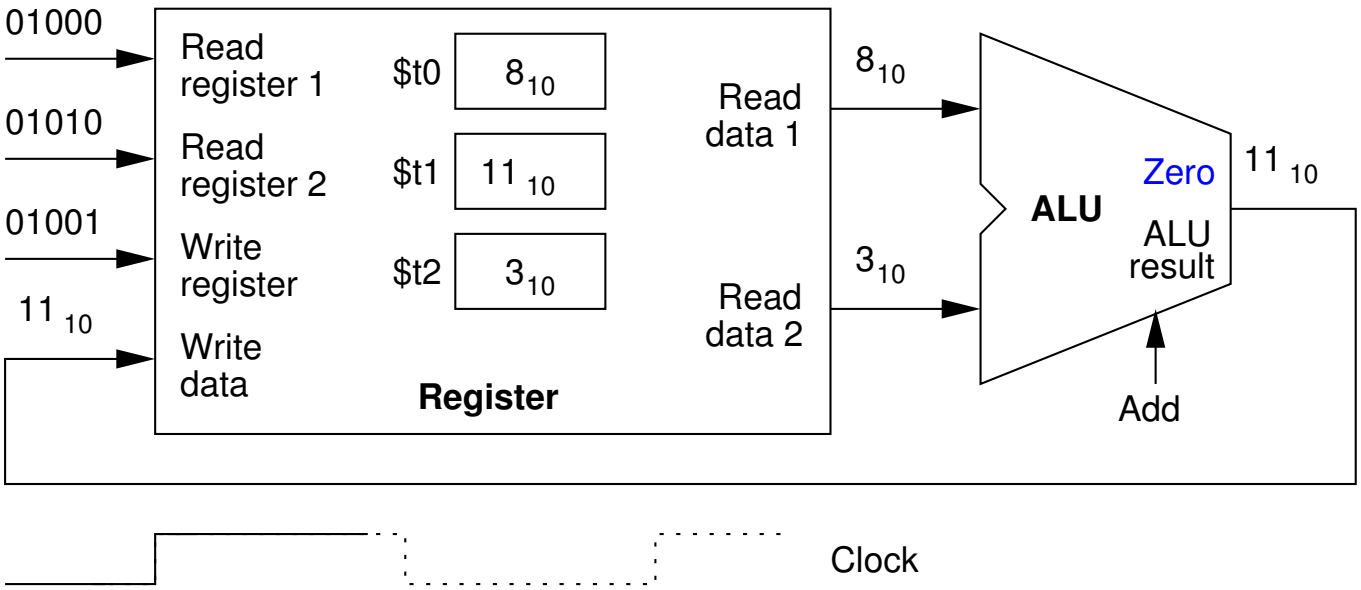
Clocking Methodology

```
  add $t1, $t0, $t2
* add $t0, $t0, $t1
```



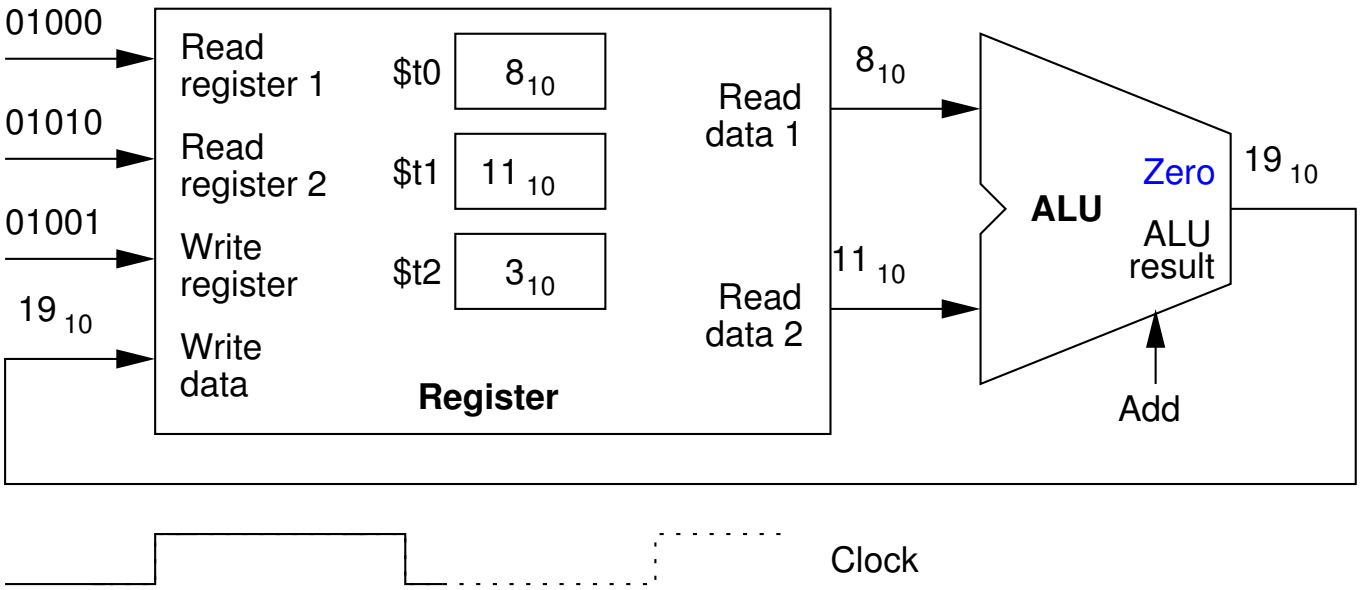
Clocking Methodology

```
  add $t1, $t0, $t2
* add $t0, $t0, $t1
```



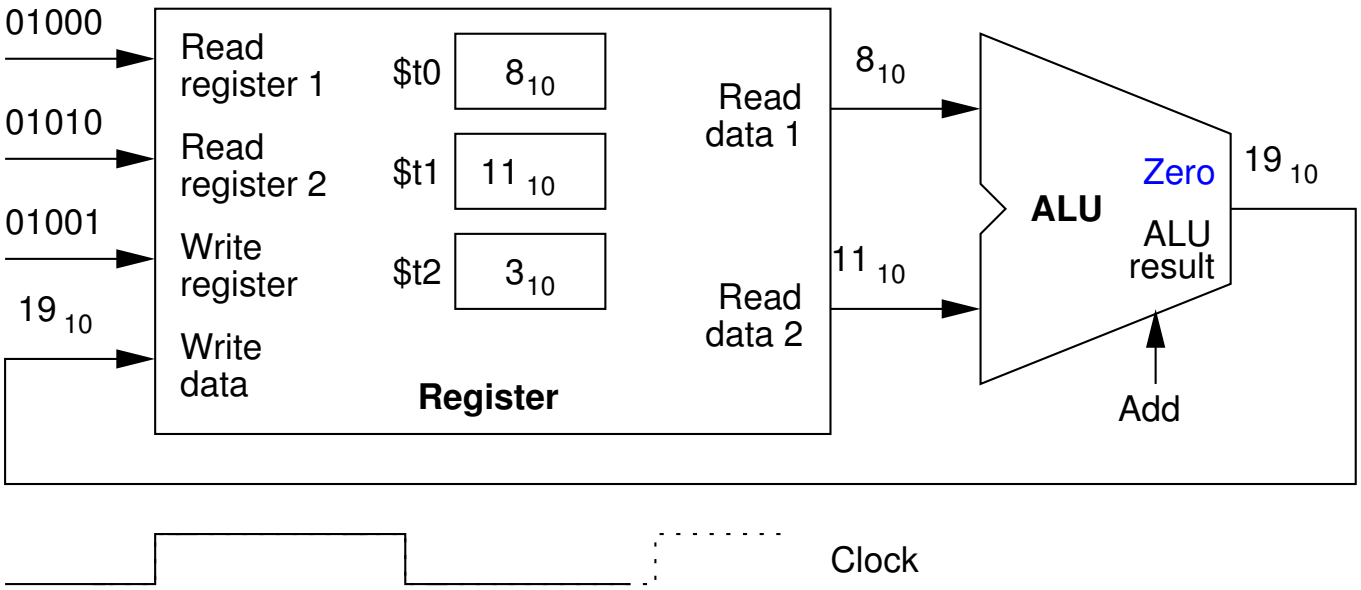
Clocking Methodology

```
  add $t1, $t0, $t2
* add $t0, $t0, $t1
```



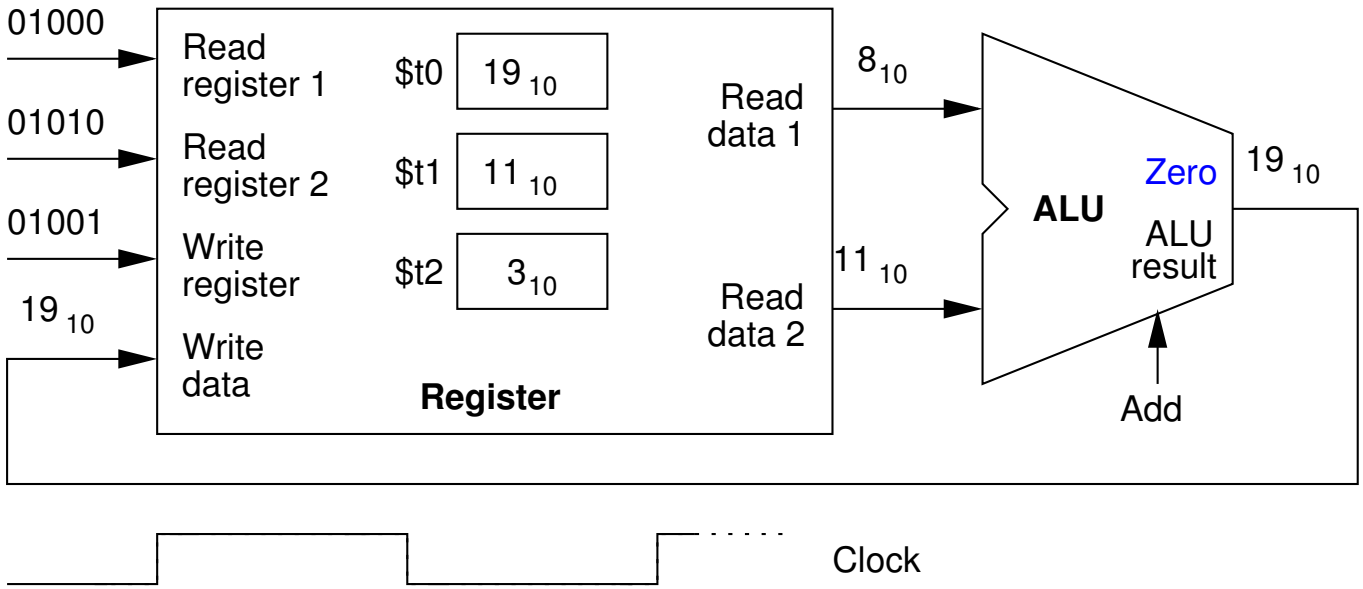
Clocking Methodology

```
  add $t1, $t0, $t2
* add $t0, $t0, $t1
```

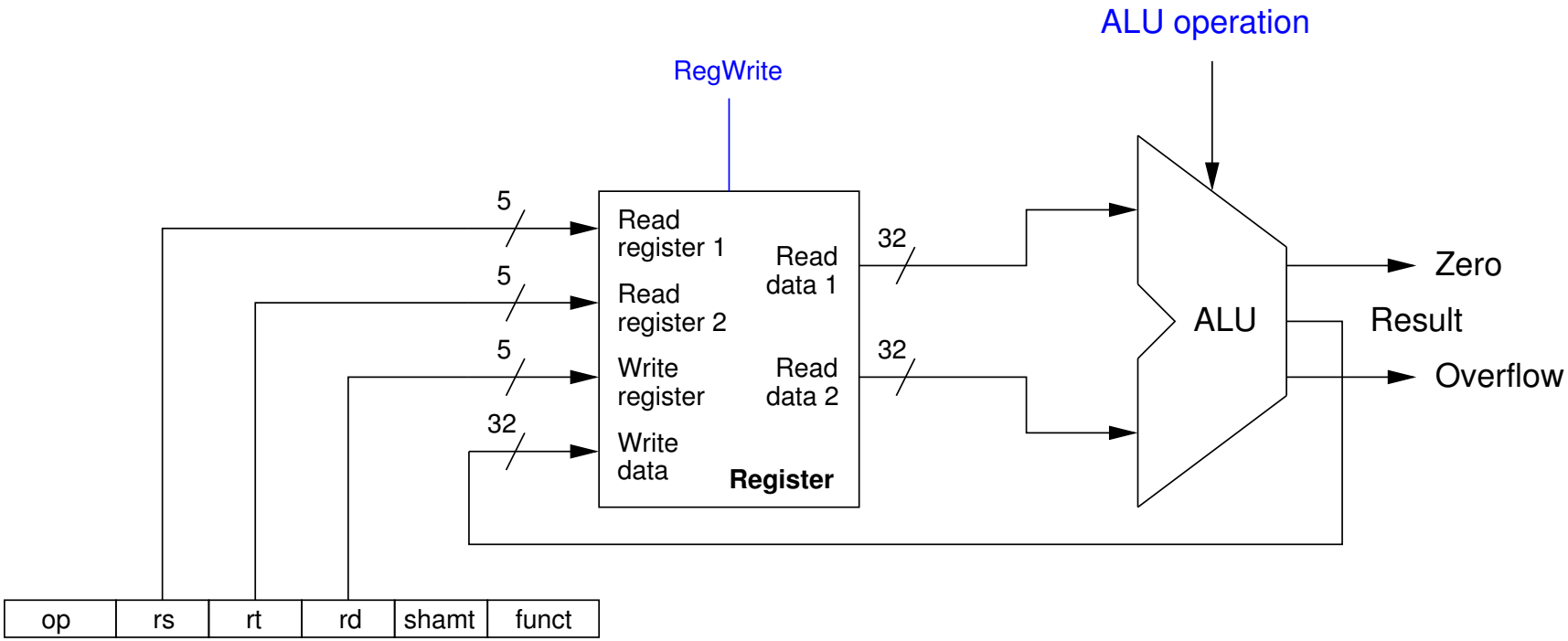


Clocking Methodology

```
  add $t0, $t0, $t1
* next instruction
```



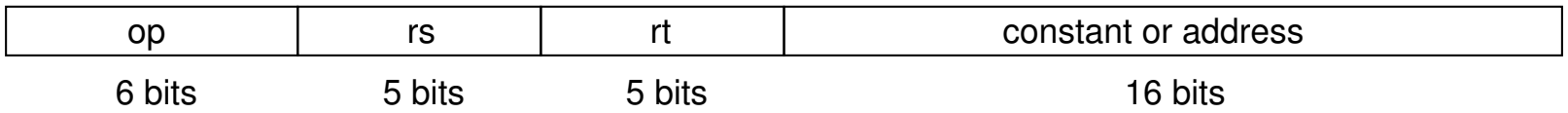
Arithmetic/Logic Instructions

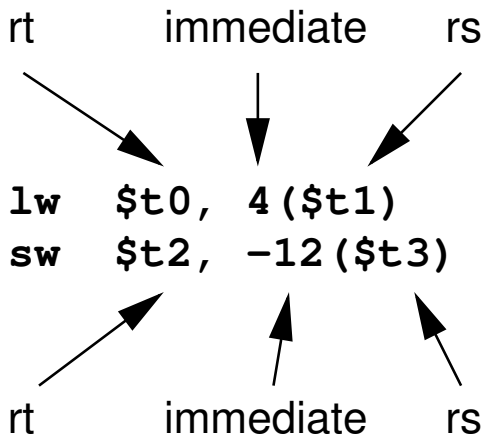


- The above processor supports arithmetic/logic instructions
- No feedback
- Ignore `RegWrite` and `ALU Operation` for now.

Memory-Reference Instruction

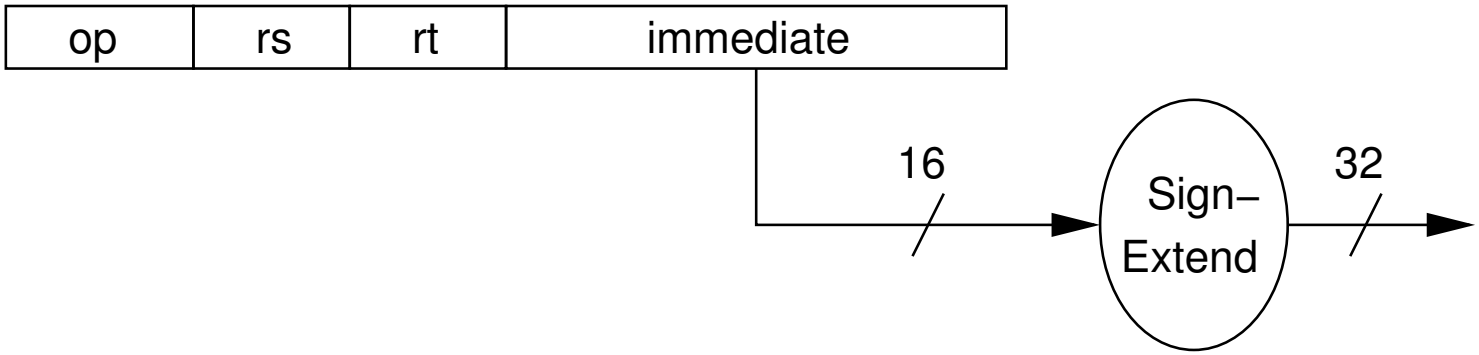
- We will focus only on Load Word (`lw`) and Store Word (`sw`) instructions.
- Both instruction uses I-type





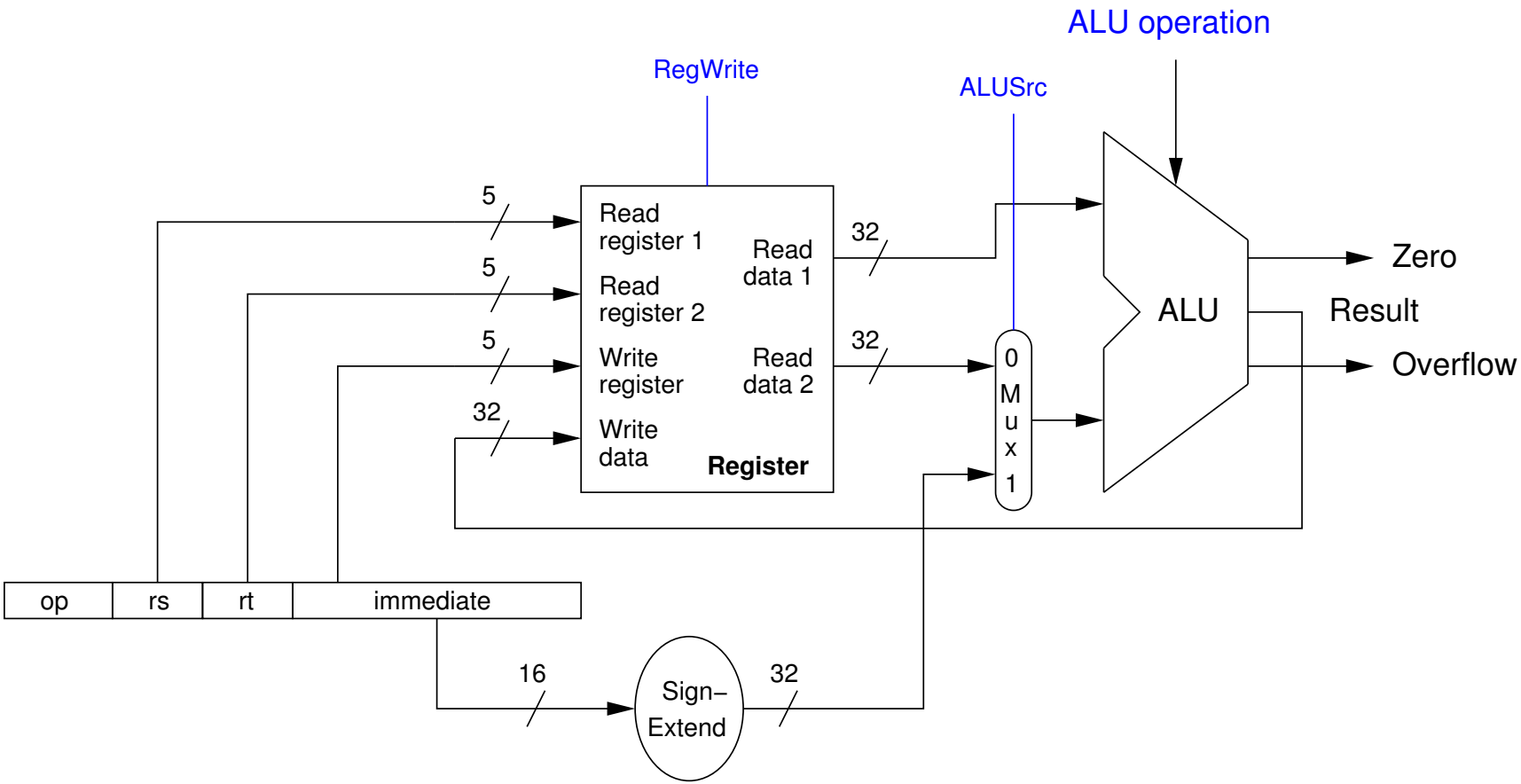
Memory-Reference Instruction

- Need to calculate the actual memory address to be read or written
  - The actual address is the value stored in the register number designated by `rs` plus the immediate value
- The immediate value from the instruction is a 16-bit number
  - We need to perform sign-extension to a 32-bit number



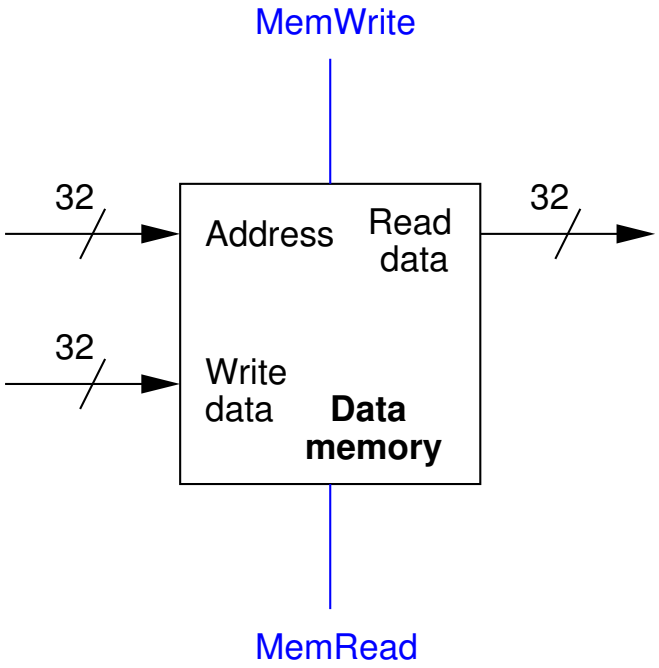
- We can use our ALU to perform addition
  - But the second input to ALU has been used by arithmetic/logic instructions
  - Need a multiplexer to select the correct input

Memory-Reference Instruction



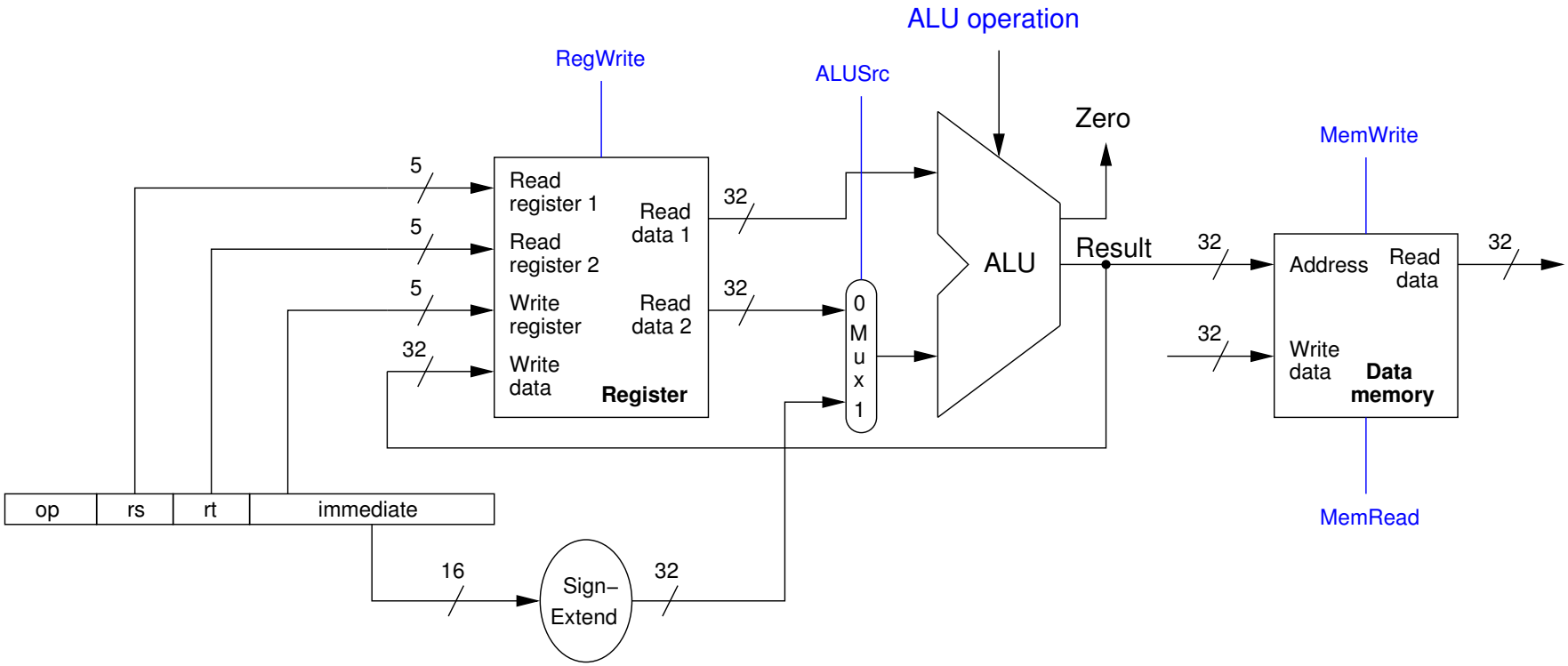
- `ALUSrc` is used to select the second operand of the ALU
- Note that we just took care of calculating the actual address, not load or store yet.

Data Memory



- Address is 32-bit wide
- Input and Output are 32-bit wide
- `MemWrite` is asserted when a data is needed to be written to a memory location
- `MemRead` is asserted when a data is needed to be read from a memory location

Processor with Data Memory



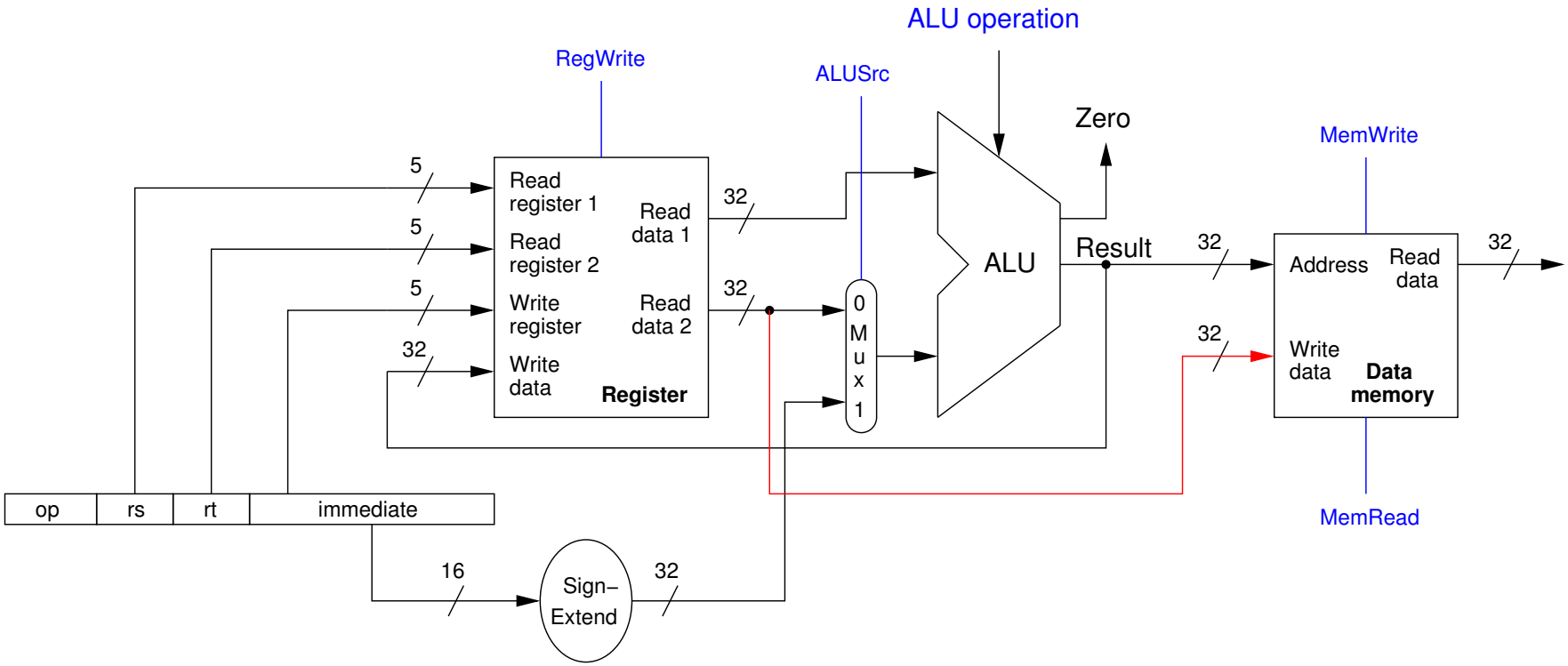
Store Word (sw) Instruction

- Recall a store word instruction

```
sw  $t0, 4($s0)
```

- The value to be stored into the data memory is the value stored in the register $t0
- The register $t0 is specified in rt of the instruction
- We need to use the value from Read data 2 to write to the data memory
- Thus, simply wire Read data 2 from register file to Write data of data memory

Store Word (sw) Instruction



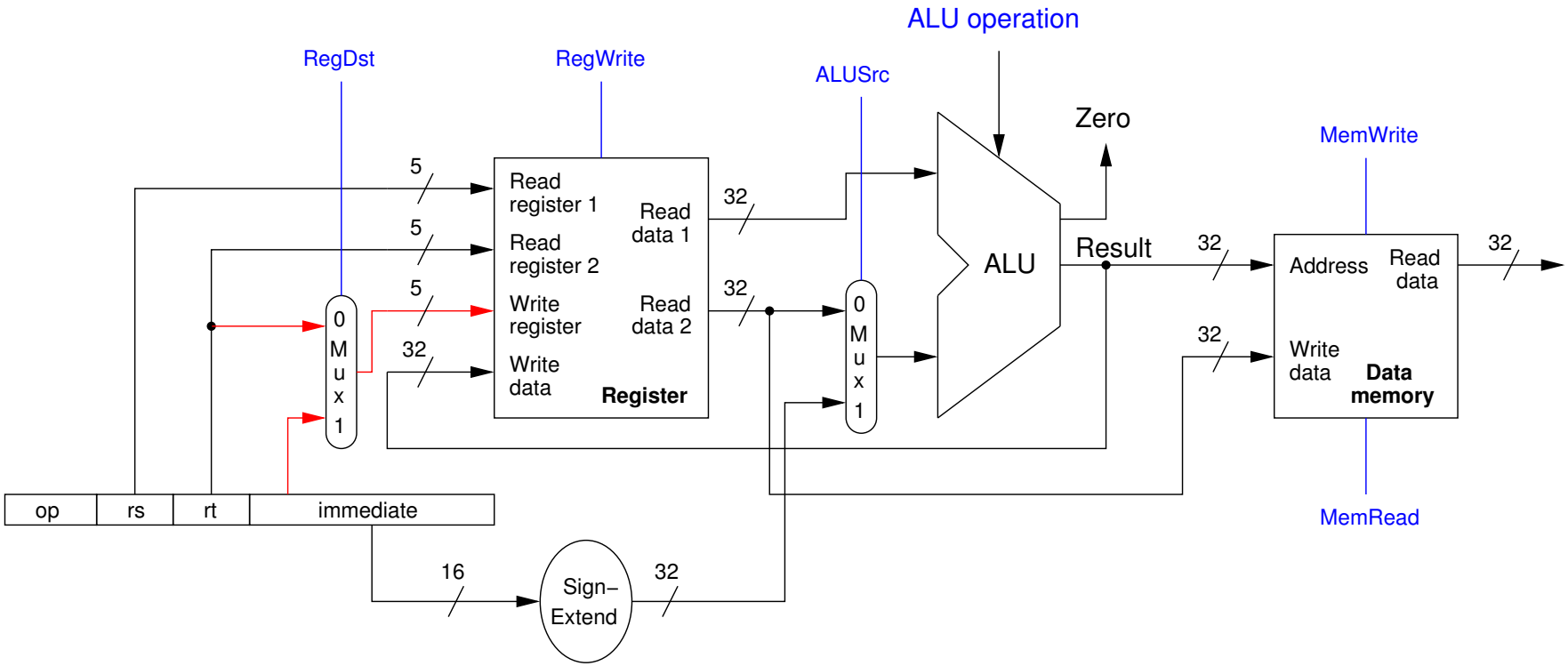
Load Word (sw) Instruction

- Recall a load word instruction

```
lw  $t0, 4($s0)
```

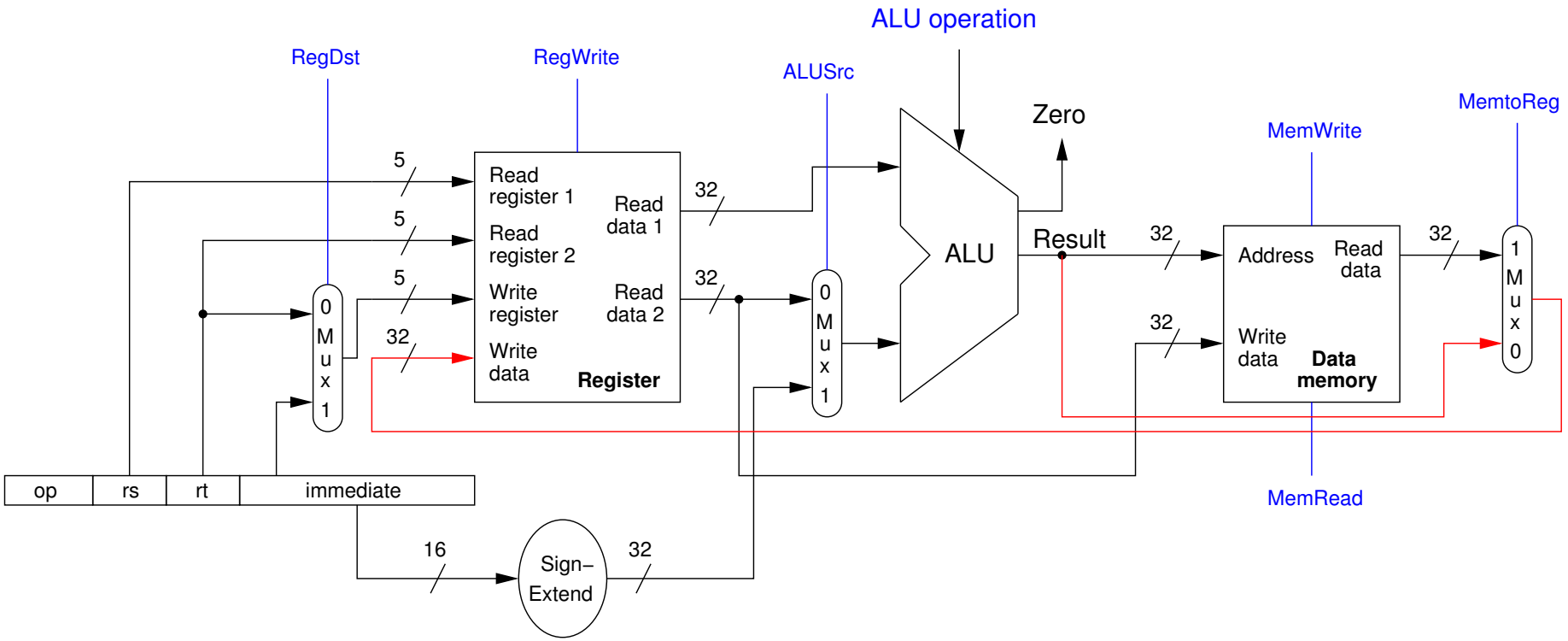
- The value read from data memory is needed to be stored in the register `$t0`
- In this case, the register `$t0` is specified in `rt` of the instruction
  - The value of `rt` is needed to go to `Write register`
  - But the `Write register` has been used by R-type instructions
  - Thus, we need a multiplexer to select the source of `Write register`
  - Sources can come from either `rt` or `rd`

Load Word (`lw`) Instruction



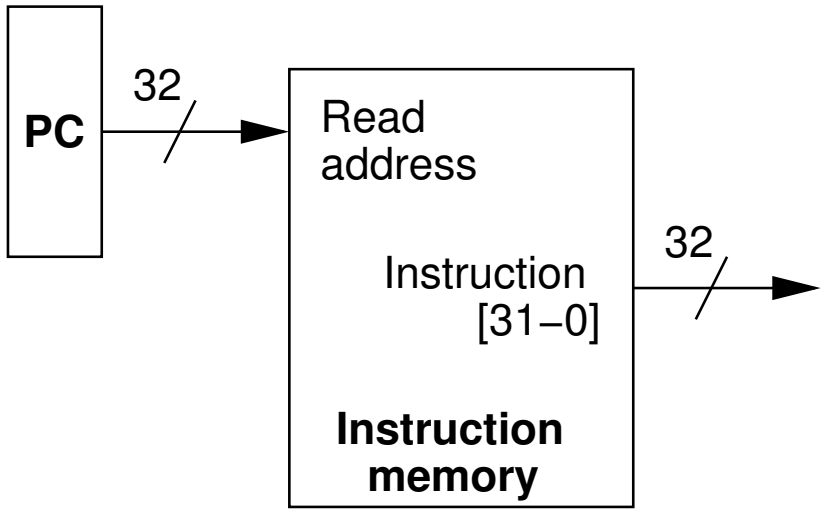
- `RegDst` is used to select the source of `Write register`
- We are not done yet! For Load Word, data from data memory must go to `Write Data` of the register file
- But it has been used by R-type instructions
- Thus, need another multiplexer

Load Word (`lw`) Instruction



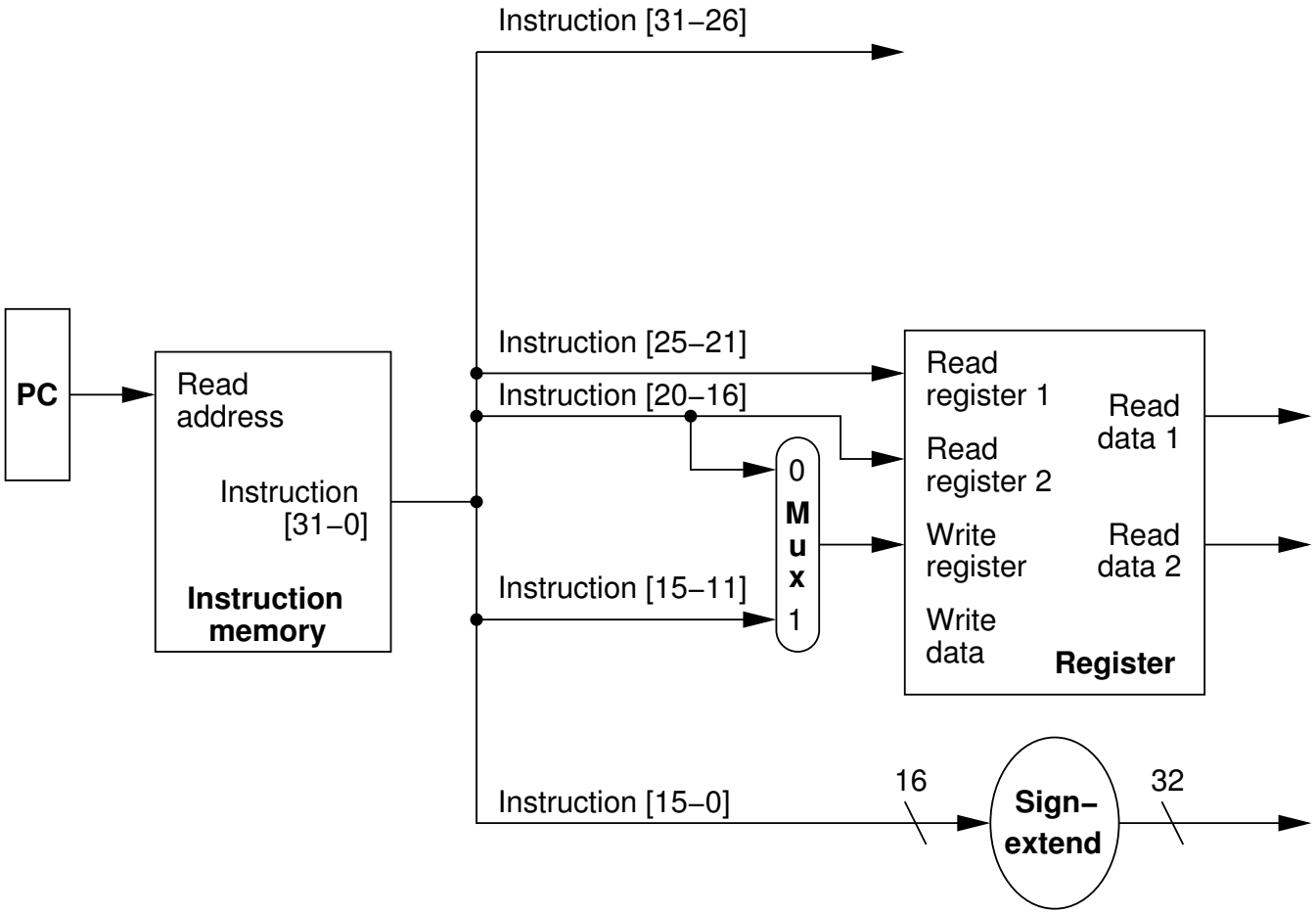
- `MemtoReg` is used to select the data to be written to the register file

Program Counter and Instruction Memory



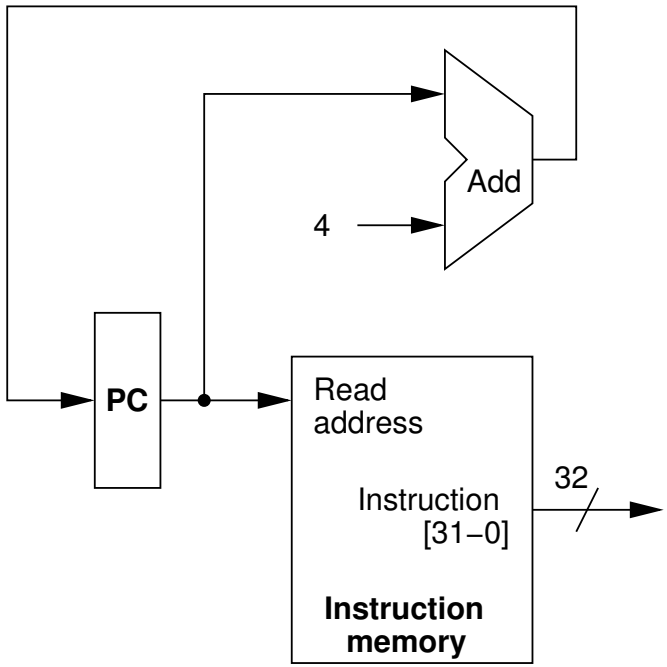
- Output from $pc is 32-bit wide (address)
- Output from instruction memory is 32-bit wide (machine instruction)

Instruction Memory to Processor



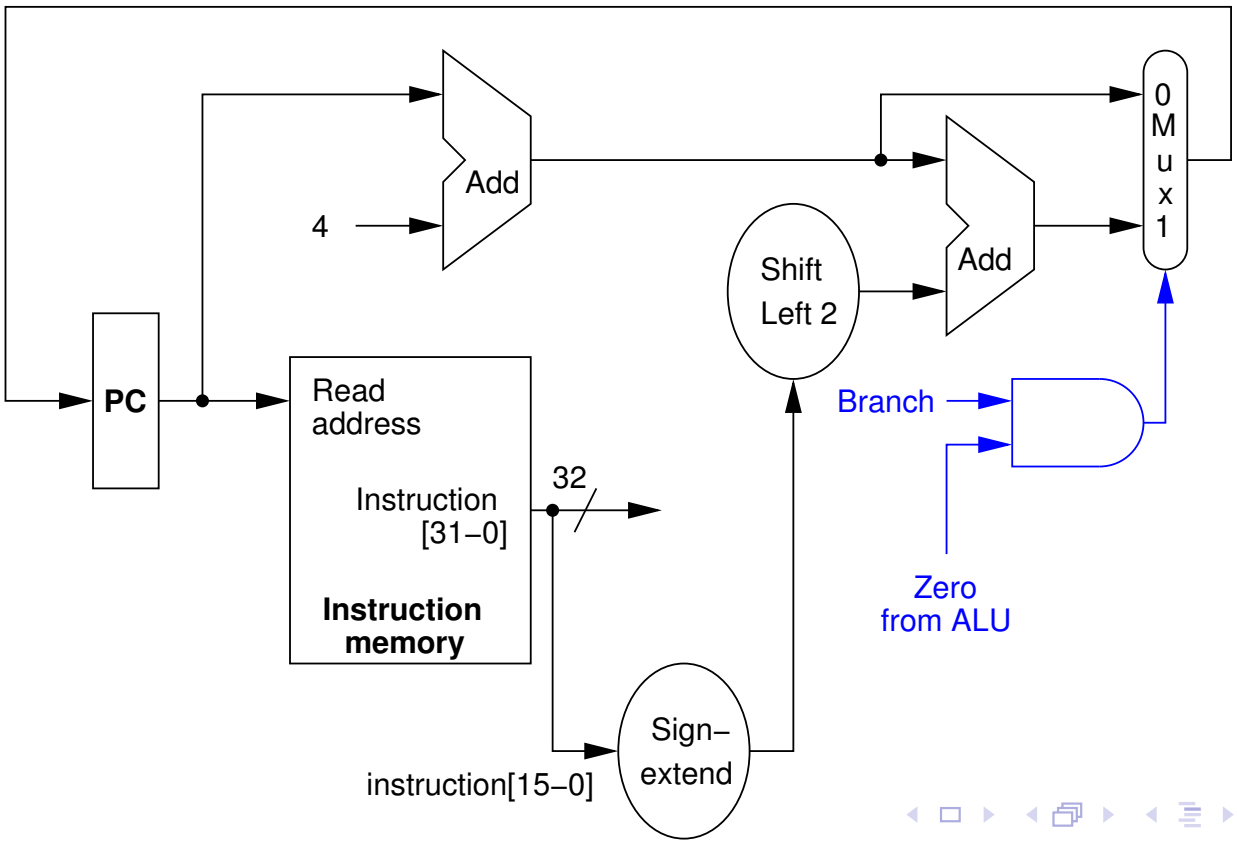
Program Counter

- After an instruction is fetched, the program counter will be increased by 4
  - Ready to fetch the next instruction



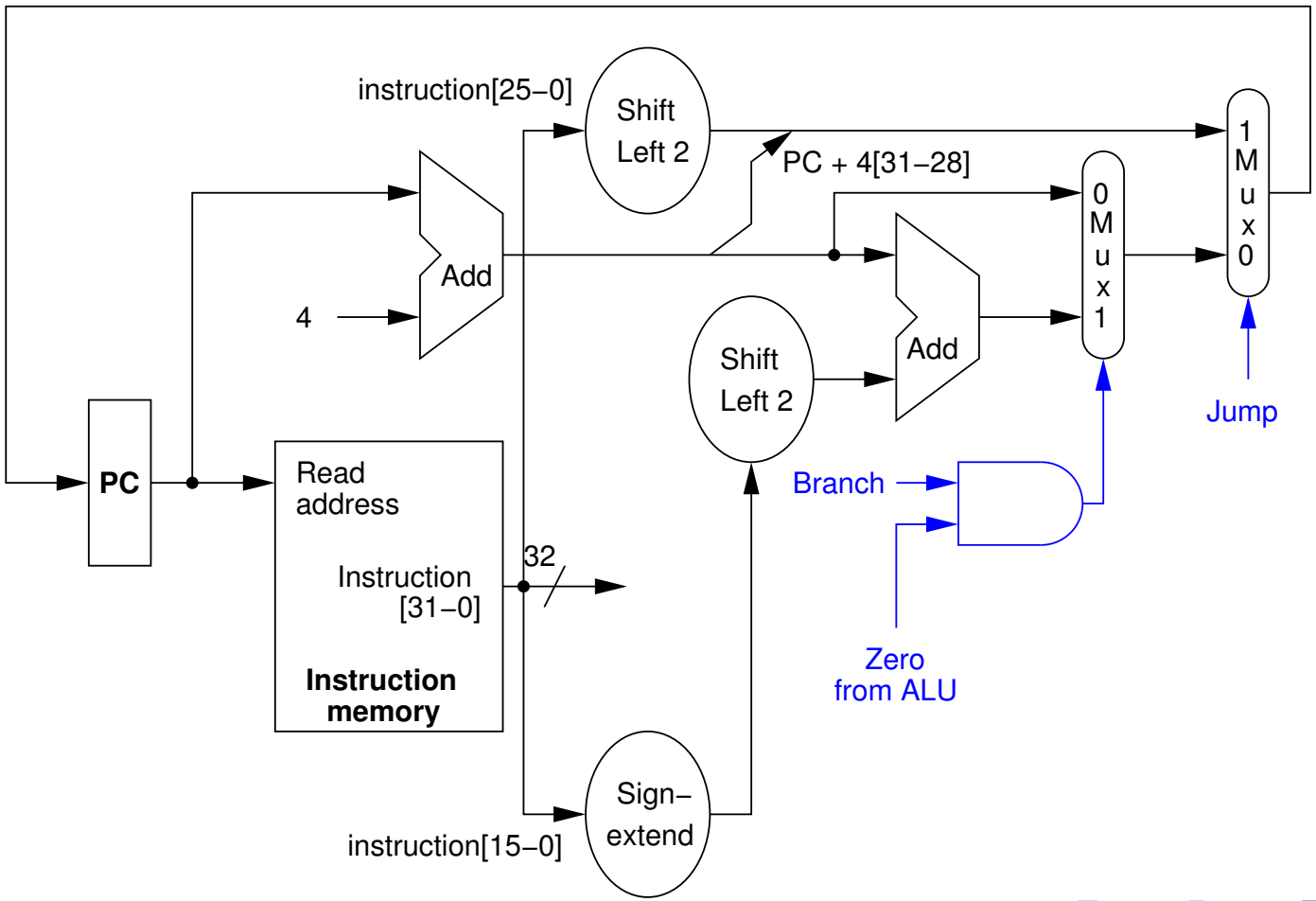
Branch Instruction

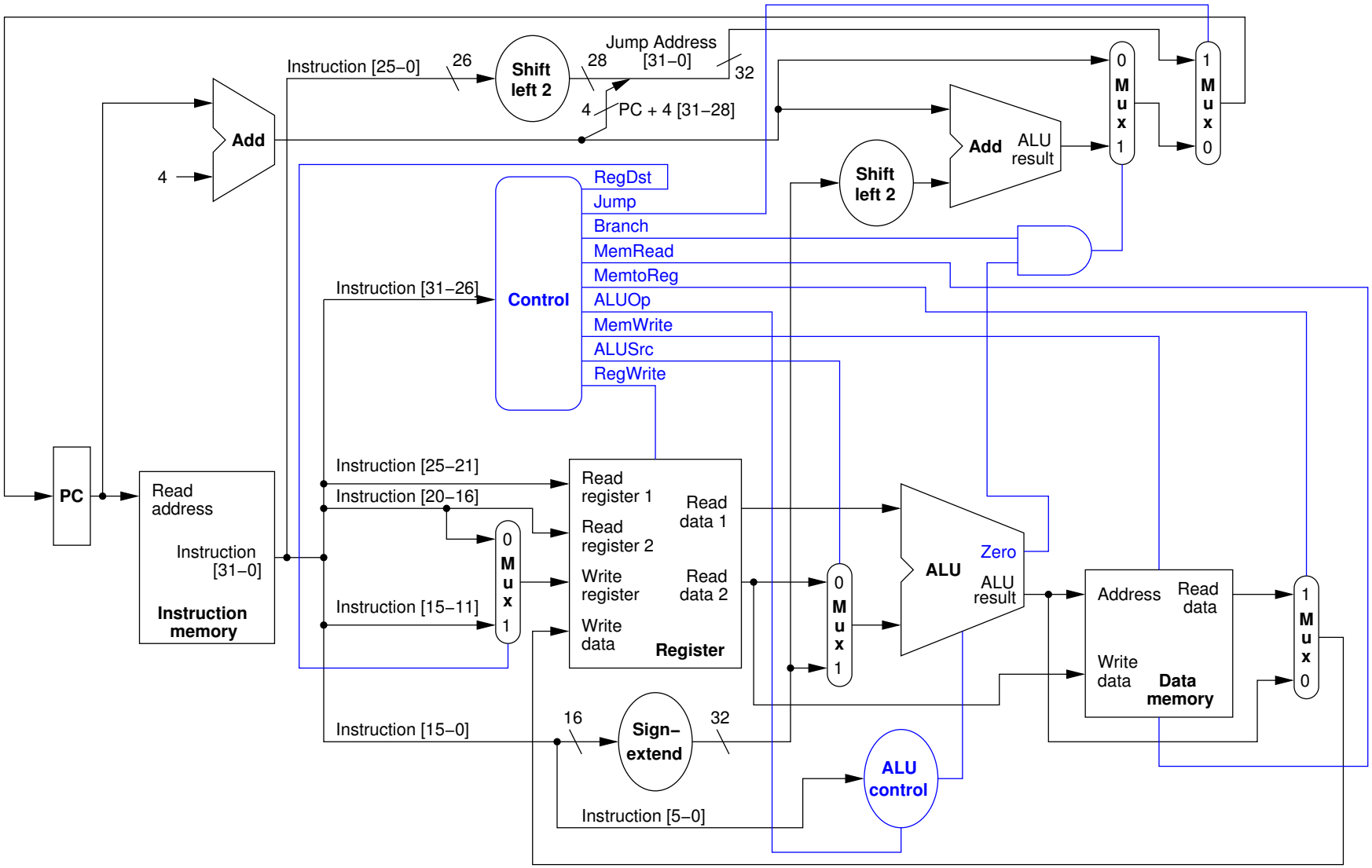
- If the instruction is a branch instruction (`beq` or `bne`), we may have to adjust PC + 4
  - Recall that we have to perform sign-extend the immediate field of branch and shift-left by 2
  - Then add it to the PC + 4 if branch is taken.
- Need a multiplexer to select which one to be used



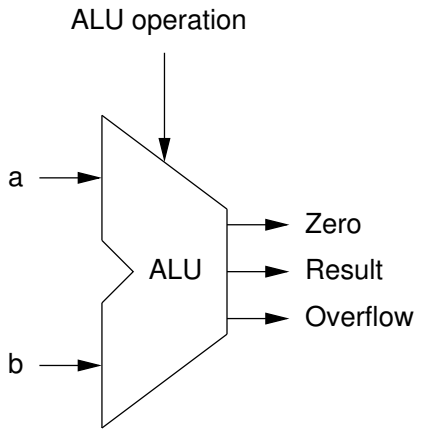
Jump Instruction

- If the instruction is the jump instruction, we have to adjust the program counter
  - Recall that we use shift the address field by 4 and use the first 4 bit of PC + 4 as a new PC




Add
4
Instruction [25–0]
26
Shift left 2
28
Jump Address [31–0]
32
4
PC + 4 [31–28]
0 M u x 1
1 M u x 0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
0 M u x 1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Register
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
16
Sign–extend
32
ALU control

ALU Control



Operation	Ainvert	Bnegate	Operation
AND	0	0	00_2
OR	0	0	01_2
Addition	0	0	10_2
Subtraction	0	1	10_2
NOR	1	1	00_2
Set on Less Than	0	1	11_2
Branch	0	1	10_2

ALU Control

- ALU operation is 4-bit wide
- ALU is needed for load word and store word
  - Addition is needed for base + offset
- ALU is needed for branch on equal
  - Subtraction is needed for comparison

Instruction	Op	funct	ALU action	ALU Control
`add`	000000	100000	add	0010
`sub`	000000	100010	subtract	0110
`and`	000000	100100	AND	0000
`or`	000000	100101	OR	0001
`slt`	000000	101010	Set on less than	0111
`beq`	000100	N/A	subtract	0110
`lw`	100011	N/A	add	0010
`sw`	101011	N/A	add	0010

ALU Control

- No `funct` field in `beq`, `lw`, and `sw`
- Just the `funct` field is not enough to control ALU operation
- Need a separate ALU control that takes 6-bit `funct` filed and 2-bit (ALUOp) from the main control
- ALUOp can be determine from Op field

Instruction	Op	ALUOp	funct	ALU action	ALU Control
`add`	000000	10	100000	add	0010
`sub`	000000	10	100010	subtract	0110
`and`	000000	10	100100	AND	0000
`or`	000000	10	100101	OR	0001
`slt`	000000	10	101010	Set on less than	0111
`beq`	000100	01	N/A	subtract	0110
`lw`	100011	00	N/A	add	0010
`sw`	101011	00	N/A	add	0010

ALUOp

- We need to decode from 6-bit `Op` to 2-bit `ALUOp`

Instruction	Op	ALUOp
`add`	000000	10
`sub`	000000	10
`and`	000000	10
`or`	000000	10
`slt`	000000	10
`beq`	000100	01
`lw`	100011	00
`sw`	101011	00

- Let $x_1 x_0$ represents 2-bit ALUOp and $o_5 o_4 o_3 o_2 o_1 o_0$ represents 6-bit Op code, what are boolean expressions representing x_1 and x_0?

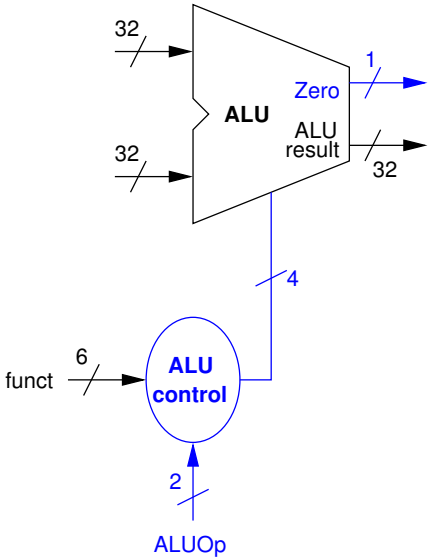
ALUOp

- We need to decode from 6-bit `Op` to 2-bit `ALUOp`

Instruction	Op	ALUOp
`add`	000000	10
`sub`	000000	10
`and`	000000	10
`or`	000000	10
`slt`	000000	10
`beq`	000100	01
`lw`	100011	00
`sw`	101011	00

- Let $x_1 x_0$ represents 2-bit ALUOp and $o_5 o_4 o_3 o_2 o_1 o_0$ represents 6-bit Op code, what are boolean expressions representing x_1 and x_0?
  - $x_1 = o_5' o_4' o_3' o_2' o_1' o_0'$
  - $x_0 = o_5' o_4' o_3' o_2 o_1' o_0'$

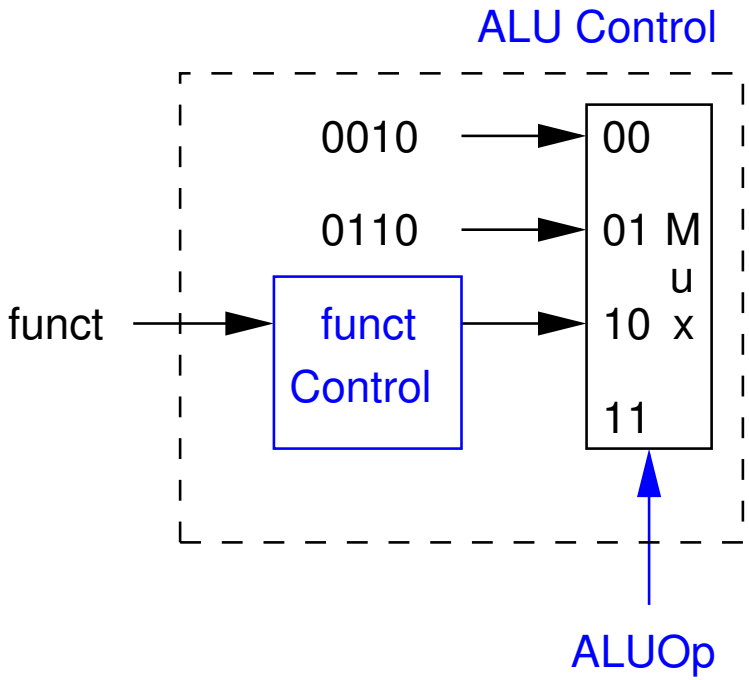
ALU Control



Instruction	ALUOp	funct	ALU Control
add	10	100000	0010
sub	10	100010	0110
and	10	100100	0000
or	10	100101	0001
slt	10	101010	0111
beq	01	N/A	0110
lw	00	N/A	0010
sw	00	N/A	0010

ALU Control

- Output of ALU Control will be 0010 when ALUOp is 00
- Output of ALU Control will be 0110 when ALUOp is 01
- Output of ALU control depends on `funct` field when ALUOp is 10
- For simplicity, we can use multiplexer as follows:



`funct` Control

- We can create the `funct` control circuit using truth table/karnaugh map
- Let's try at a very minimum. So, recall the `funct` values and its ALU Control output

funct	ALU Control
100000	0010
100010	0110
100100	0000
100101	0001
101010	0111

- For this design, we will only focus on that 5 instructions
- Let $f_5 f_4 f_3 f_2 f_1 f_0$ be a 6-bit value of `funct` field and $AC_3 AC_2 AC_1 AC_0$ be a 4-bit output value of ALU Control.

funct Control

- Let's make some observations:

funct	ALU Control
100000	0010
100010	0110
100100	0000
100101	0001
101010	0111

- From the above table:
  - f_5 is always 1 and f_4 is always 0 for all 5 instructions
  - AC_3 is always 0 for all 5 instructions
- We can ignore those inputs and output and focus on f_3 to f_0 and AC_2 to AC_0.

AC_2 Output

- Let's consider AC_2 output:

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	0	x	x	1
01	0	0	x	x
11	x	x	x	x
10	x	x	x	1

- An x indicates **don't care** which can be either 0 or 1

AC_2 Output

- Let's consider AC_2 output:

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	0	x	x	1
01	0	0	x	x
11	x	x	x	x
10	x	x	x	1

- An x indicates **don't care** which can be either 0 or 1

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	0	x	x	1
01	0	0	x	x
11	x	x	x	x
10	x	x	x	1

- If we consider xs in blue cells as 1s, we get $AC_2 = f_1$

AC_1 Output

- Let's consider AC_1 output:

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	1	x	x	1
01	0	0	x	x
11	x	x	x	x
10	x	x	x	1

- An x indicates **don't care** which can be either 0 or 1

AC_1 Output

- Let's consider AC_1 output:

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	1	x	x	1
01	0	0	x	x
11	x	x	x	x
10	x	x	x	1

- An x indicates **don't care** which can be either 0 or 1

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	1	x	x	1
01	0	0	x	x
11	x	x	x	x
10	x	x	x	1

- If we consider xs in blue cells as 1s, we get $AC_1 = f_2'$

AC_0 Output

- Let's consider AC_0 output:

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	0	x	x	0
01	0	1	x	x
11	x	x	x	x
10	x	x	x	1

- An x indicates **don't care** which can be either 0 or 1

AC_0 Output

- Let's consider AC_0 output:

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	0	x	x	0
01	0	1	x	x
11	x	x	x	x
10	x	x	x	1

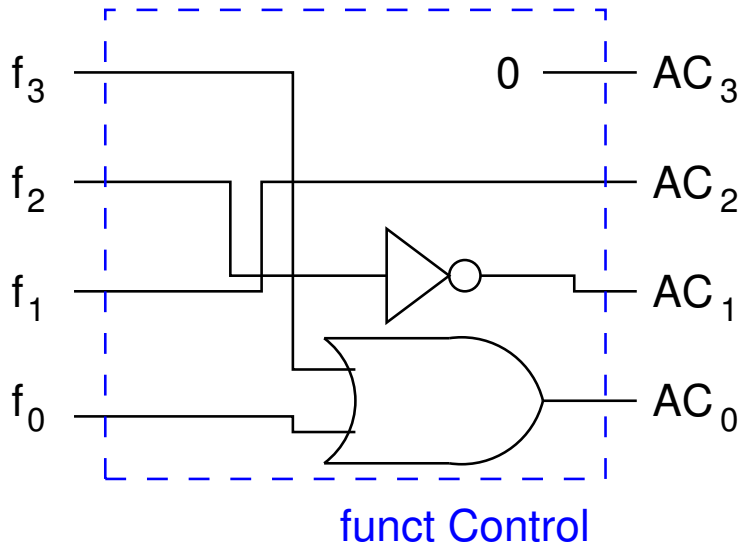
- An x indicates **don't care** which can be either 0 or 1

$f_3 f_2$ \ $f_1 f_0$	00	01	11	10
00	0	x	x	0
01	0	1	x	x
11	x	x	x	x
10	x	x	x	1

- If we consider xs in blue cells as 1s, we get $AC_0 = f_3 + f_0$

`funct` Control

- If we ignore inputs f_5 and f_4, we have
  - $AC_3 = 0$,
  - $AC_2 = f_1$,
  - $AC_1 = f_2'$, and
  - $AC_0 = f_3 + f_0$
- Thus, the `funct` control circuit can be

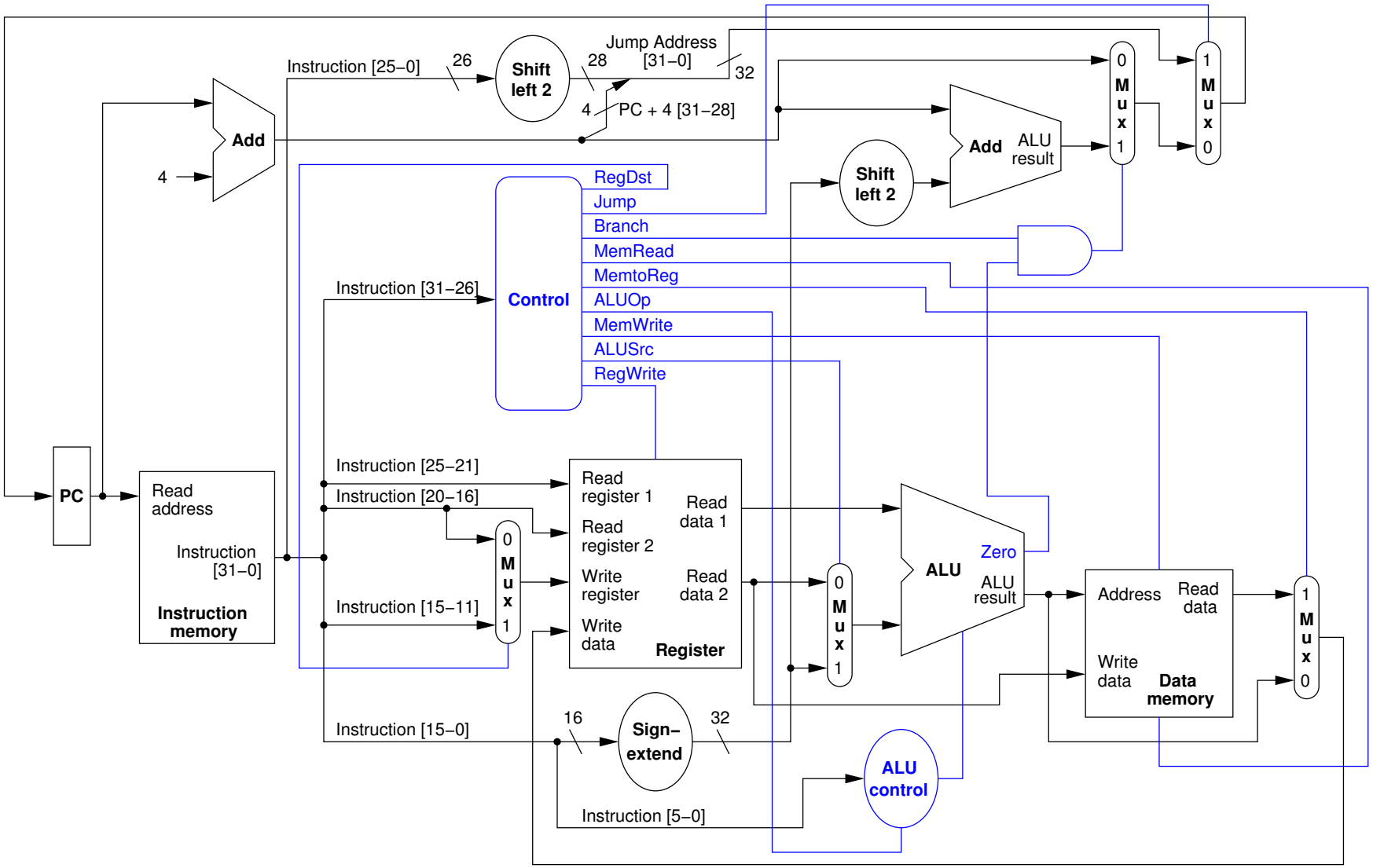


Main Control Unit

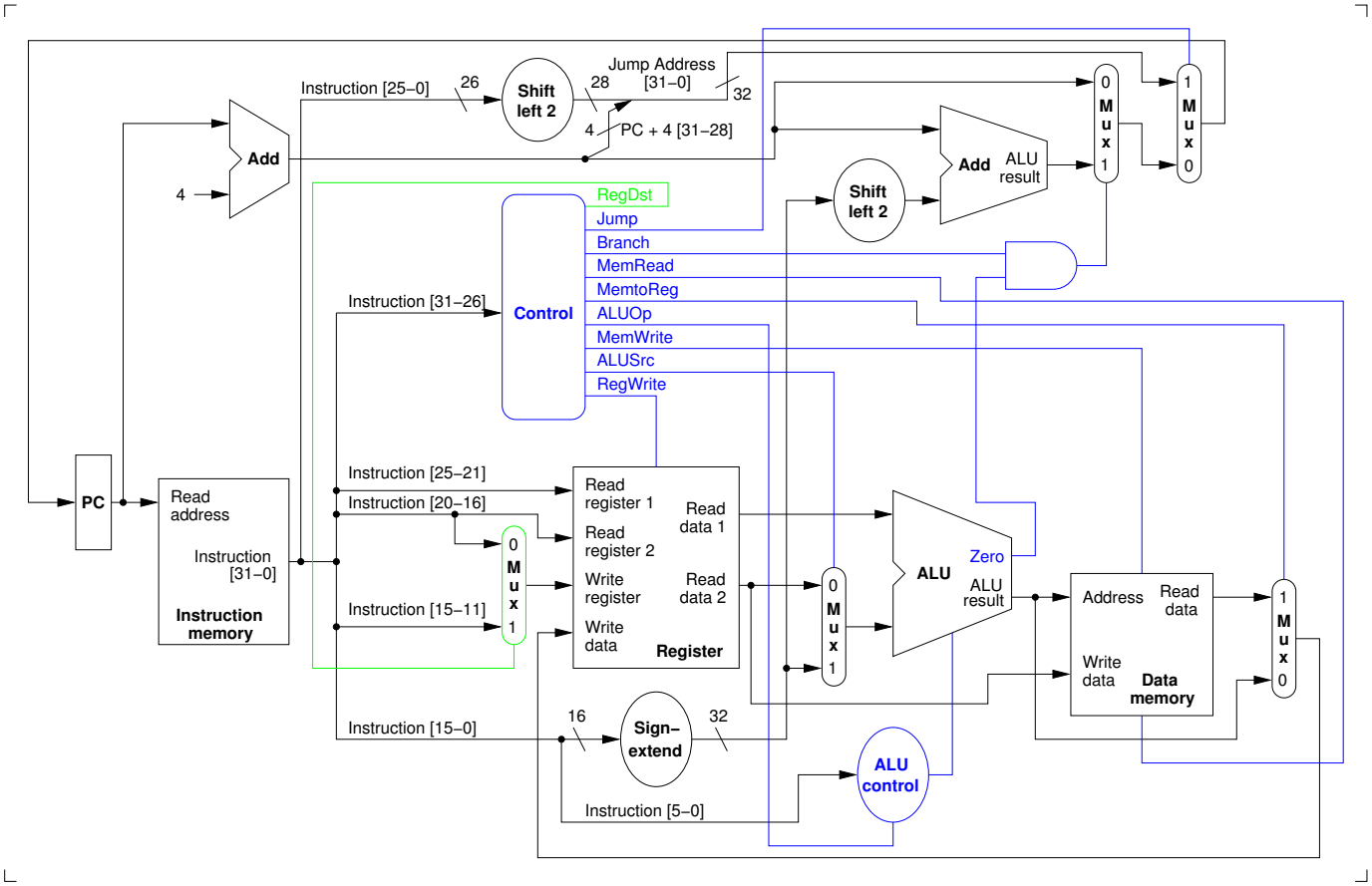
Signal Name	Effect with deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the `rt` field ([20-16])	The register destination number for the Write register comes from the `rd` field ([15-11])
RegWrite	None.	The register on the write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc (Branch)	The PC is replaced by the output of the adder that computes the value of PC + 4 or jump address	The PC is replaced by the output of the adder that computes the branch target or jump address
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.
Jump	The PC is replaced by the output of either PC + 4 or adder that computes branch target	The PC is replaced by the jump address

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp	jump
R-type	1	0	0	1	0	0	0	10	0
`lw`	0	1	1	1	1	0	0	00	0
`sw`	N/A	1	N/A	0	0	1	0	00	0
`beq`	N/A	0	N/A	0	0	0	1	01	0
`j`	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1

A Simple Implementation

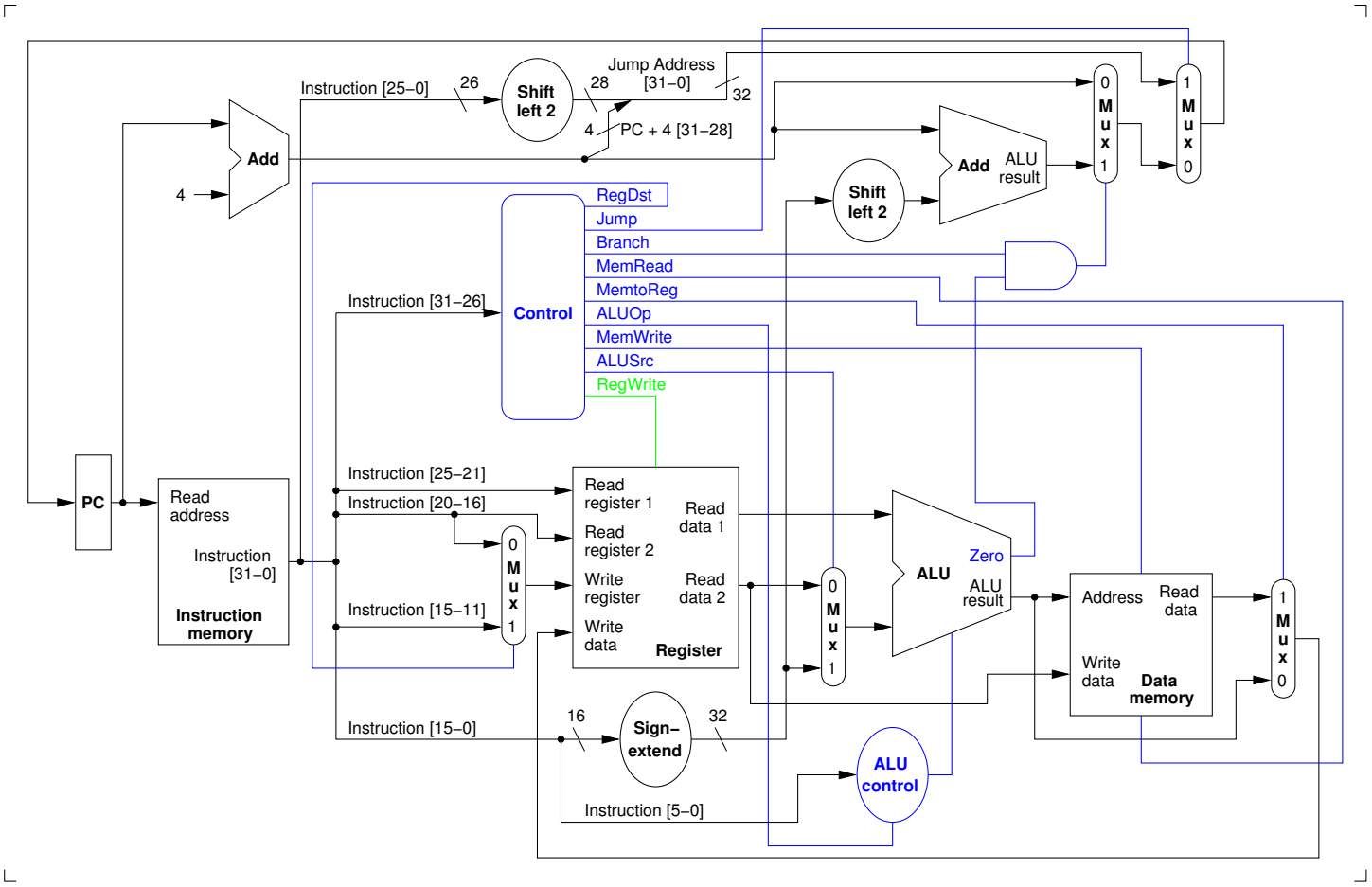


RegDst



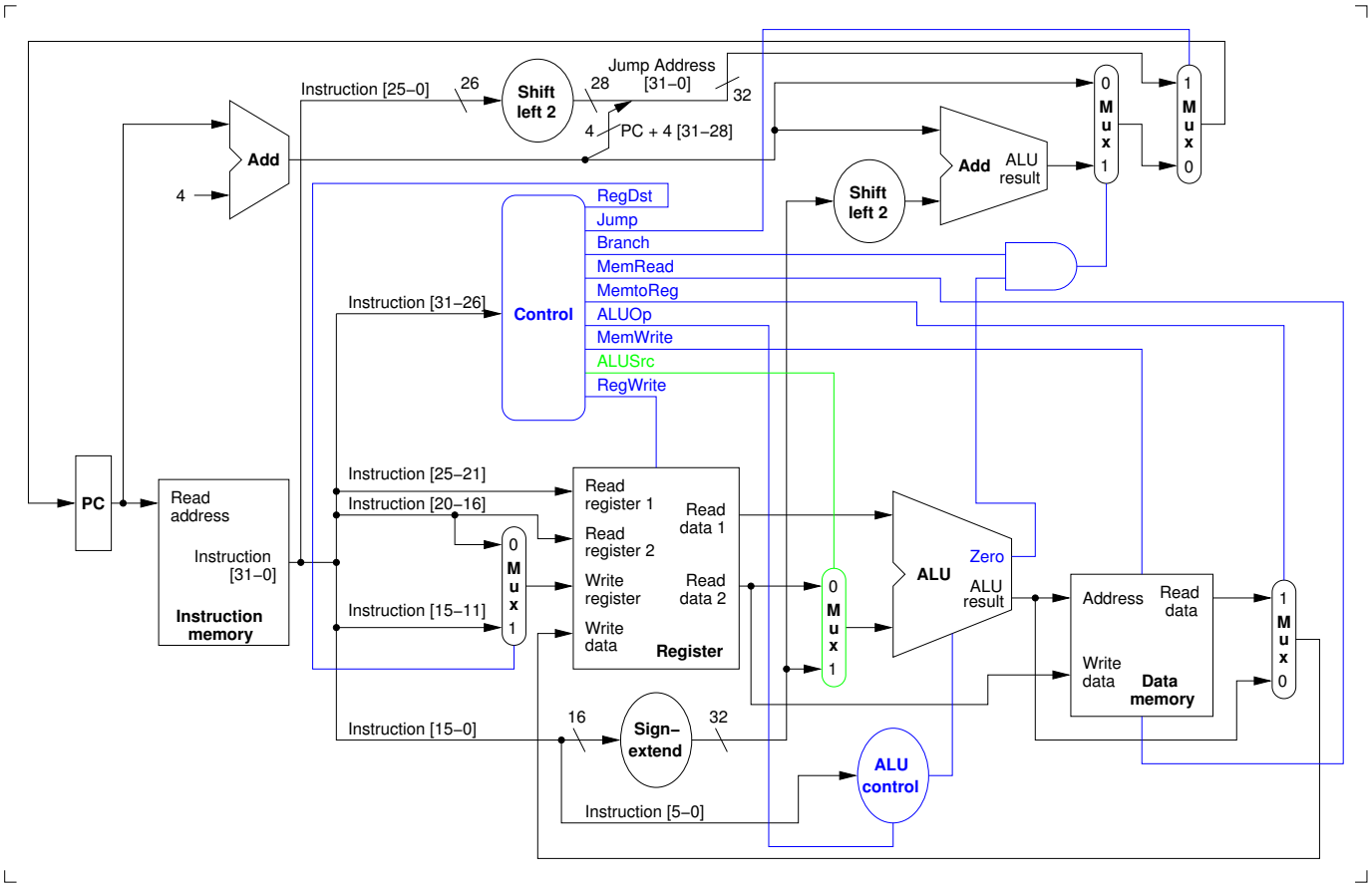
- 0: The register destination number for the Write register comes from the `rt` field ([20-16])
- 1: The register destination number for the Write register comes from the `rd` field ([15-11])

RegWrite



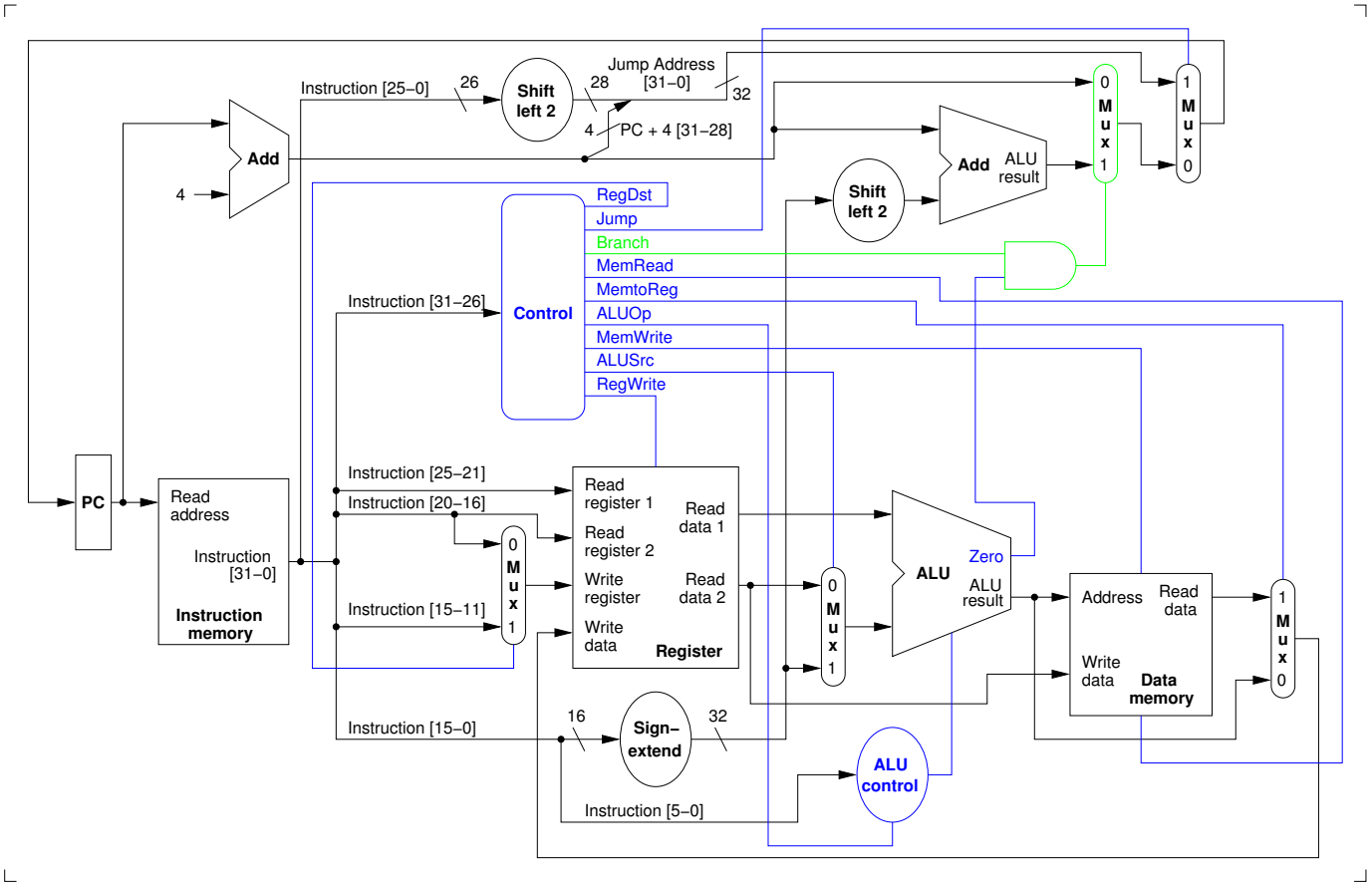
- 0: None
- 1: The register on the write register input is written with the value on the Write data input

ALUSrc



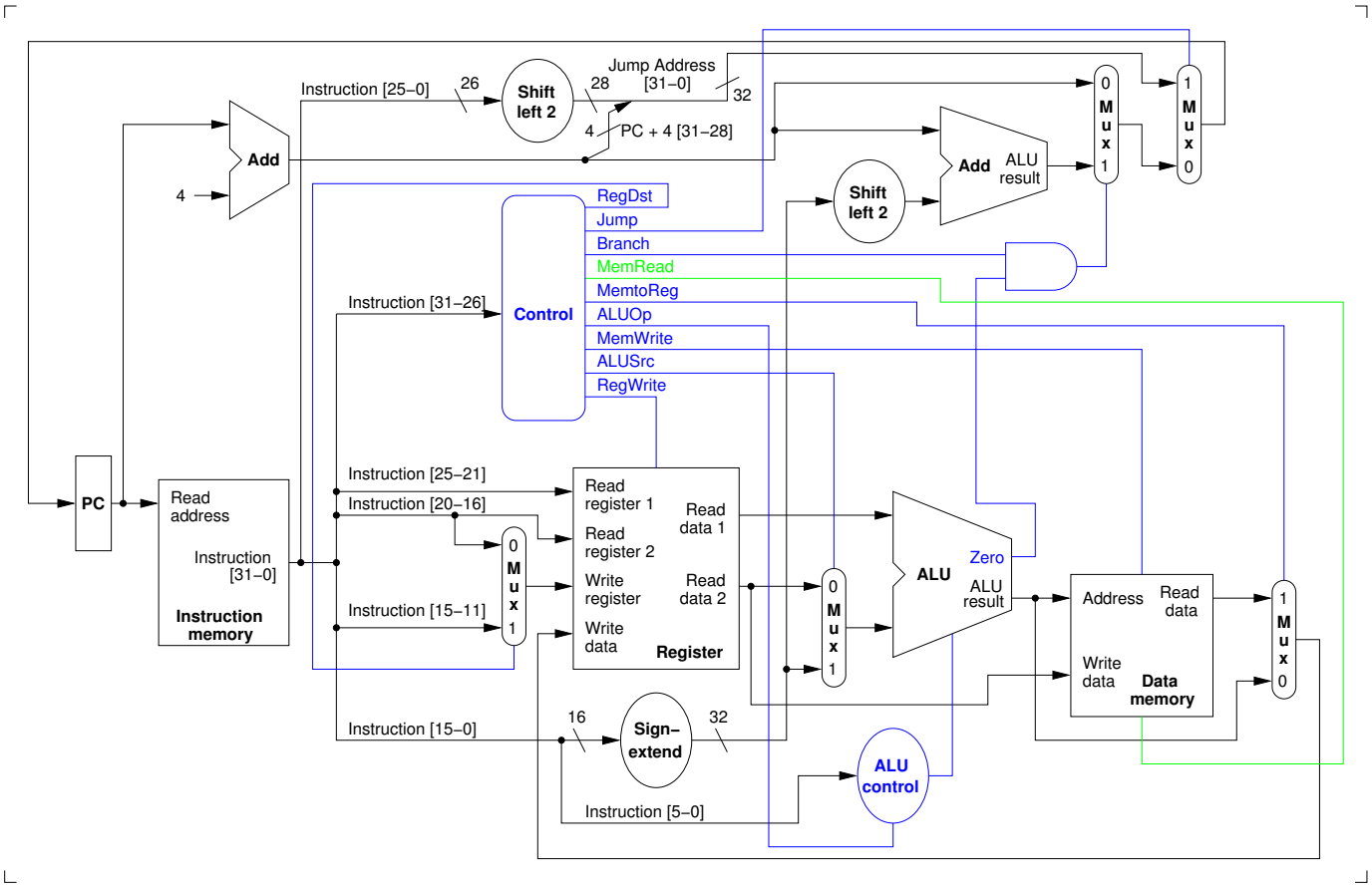
- 0: The second ALU operand comes from the second register file output (Read data 2)
- 1: The second ALU operand is the sign-extended, lower 16 bits of the instruction.

Branch



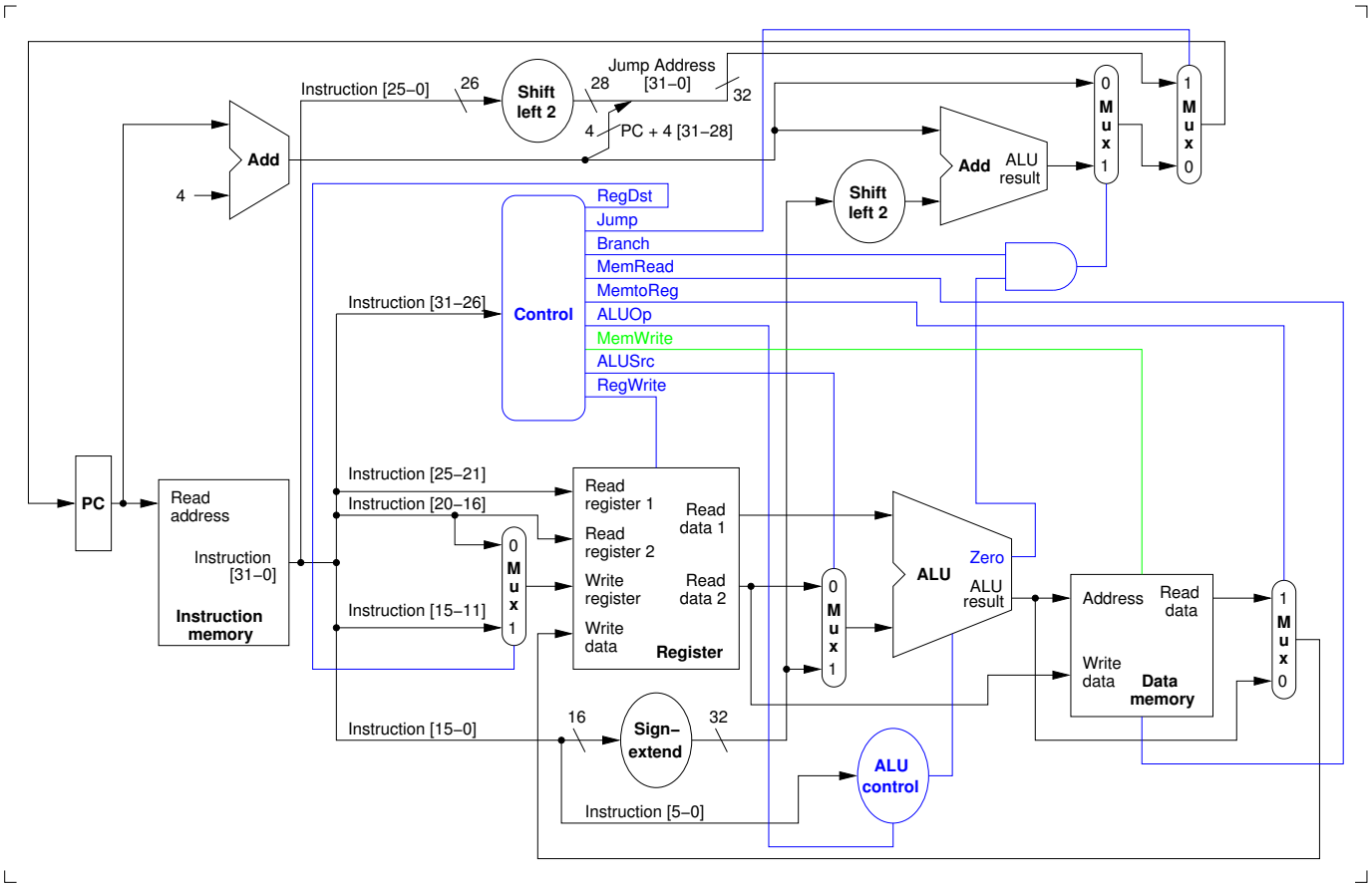
- 0: The PC is replaced by the output of the adder that computes the value of PC + 4
- 1: The PC is replaced by the output of the adder that computes the branch target

MemRead



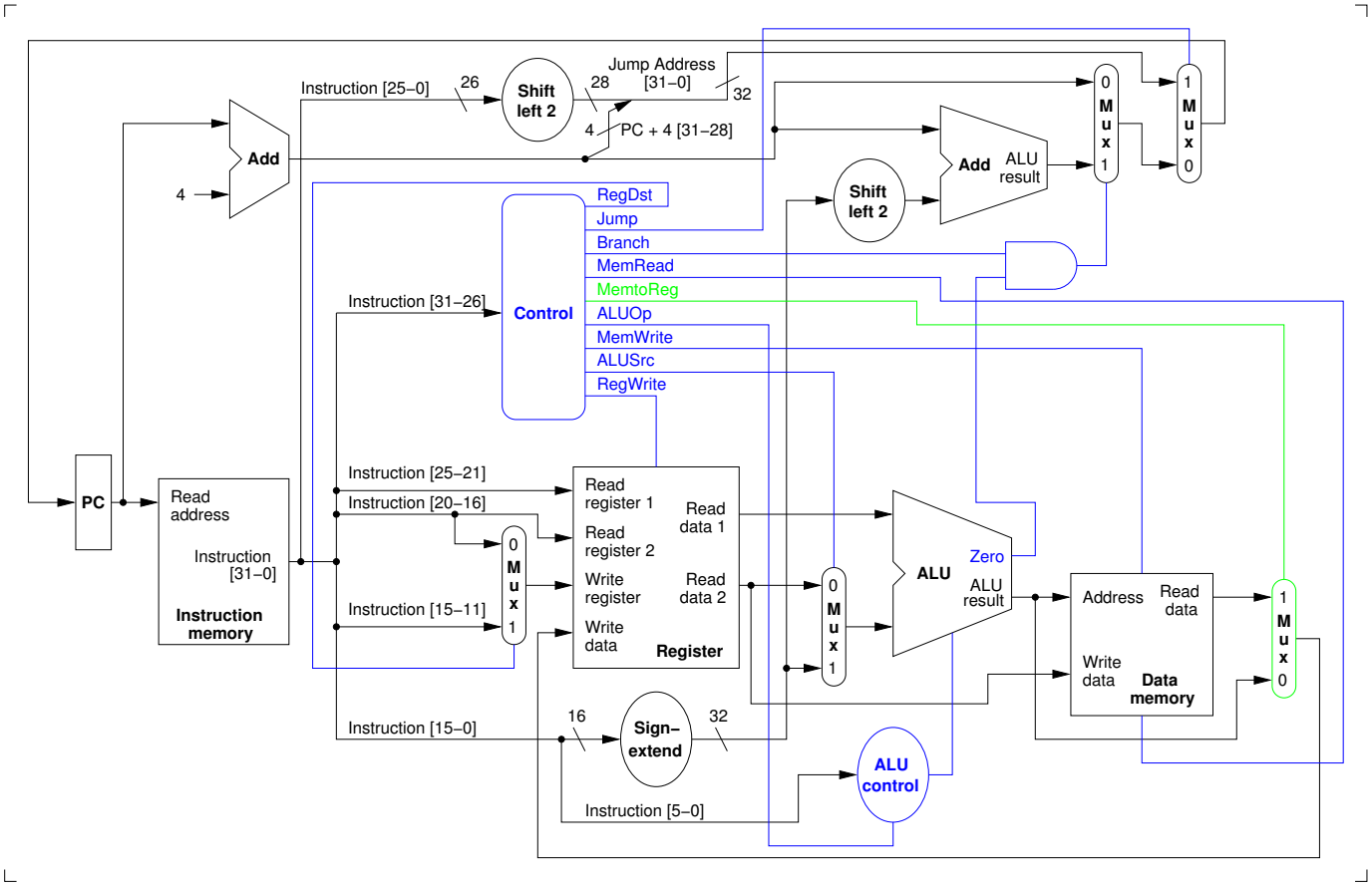
- 0: None
- 1: Data memory contents designated by the address input are put on the Read data output

MemWrite



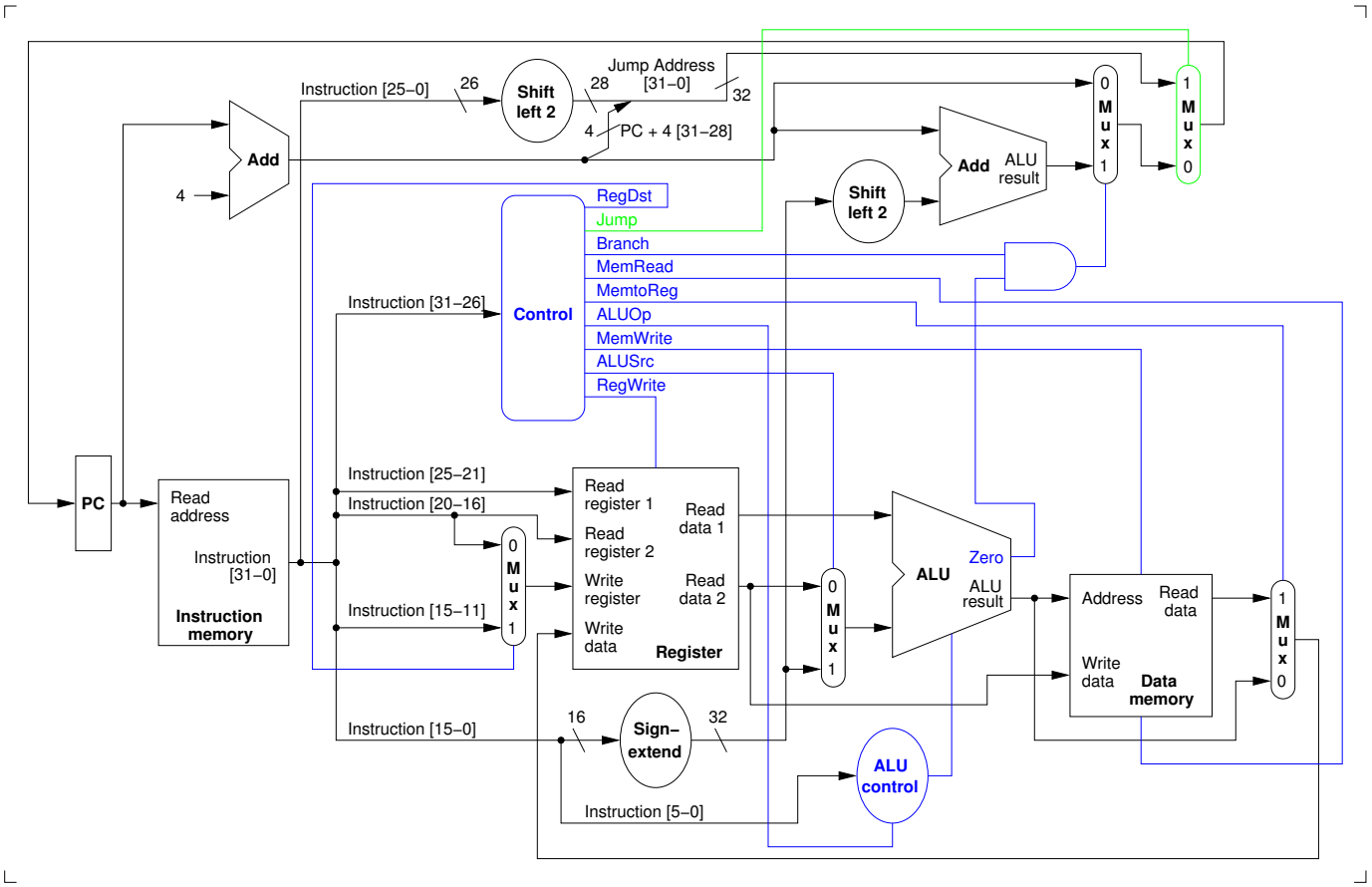
- 0: None
- 1: Data memory contents designated by the address input are replaced by the value on the Write data input

MemtoReg



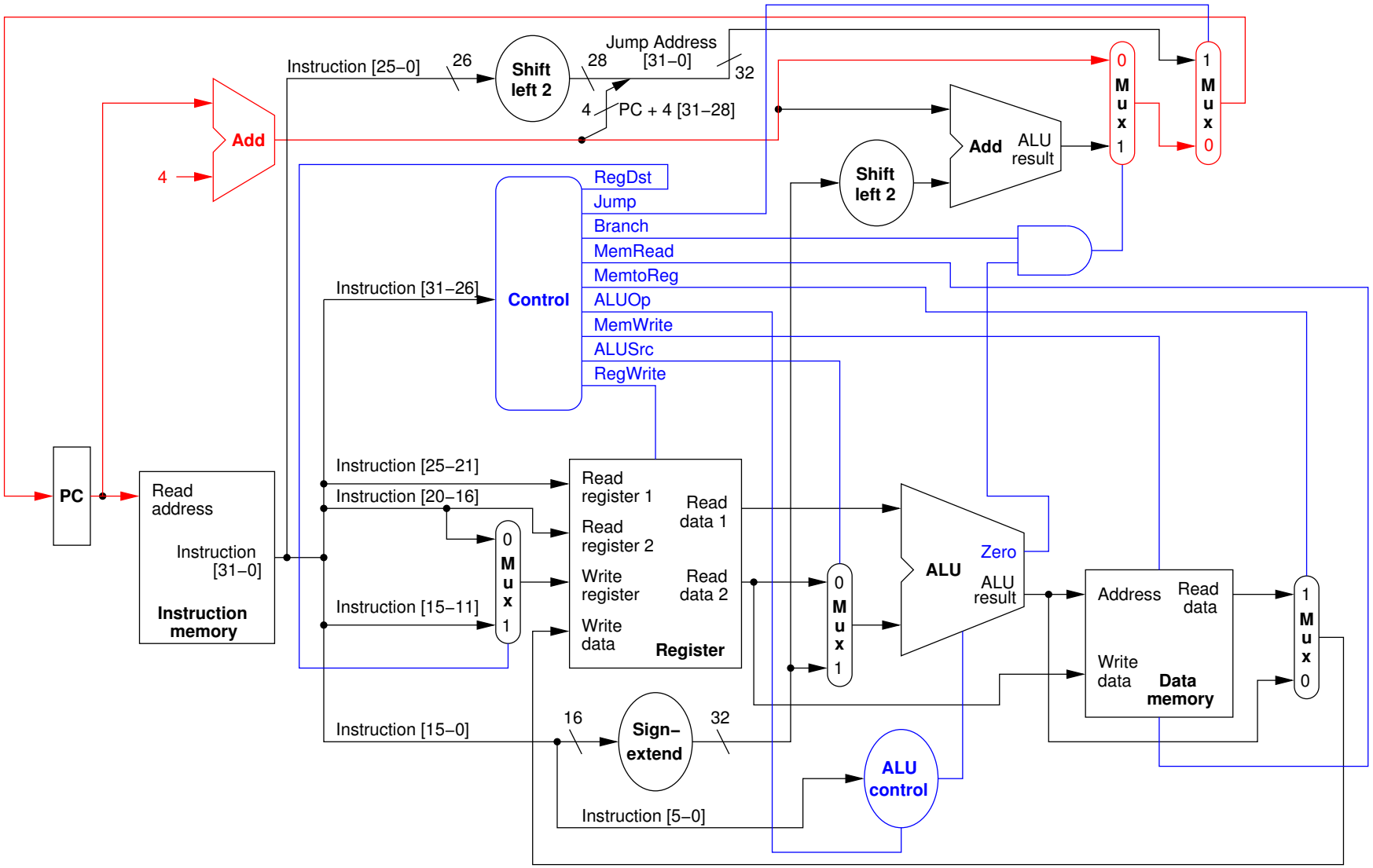
- 0: The value fed to the register Write data input comes from the ALU
- 1: The value fed to the register Write data input comes from the data memory

Jump

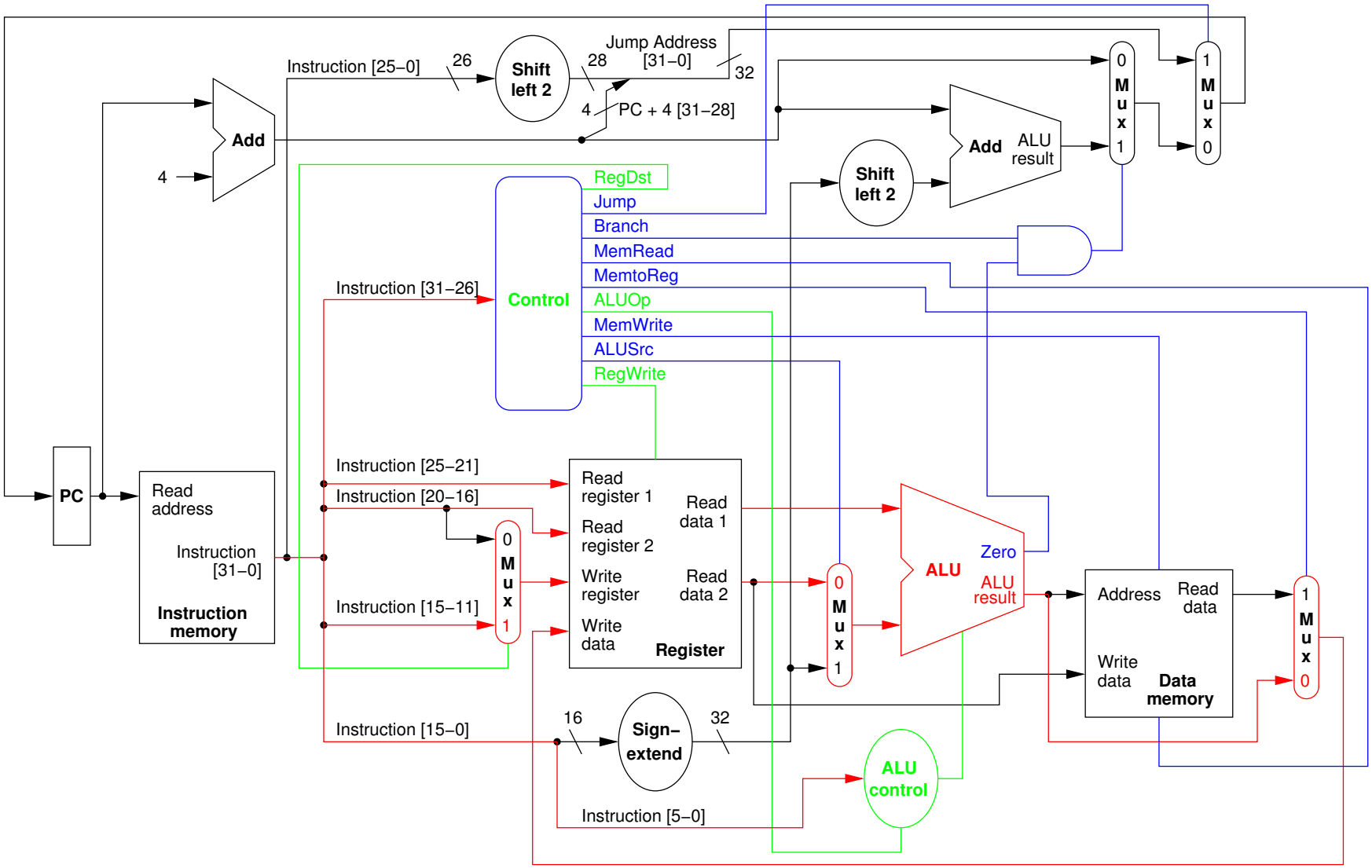


- 0: The PC is replaced by the output of either PC + 4 or adder that computes branch target
- 1: The PC is replaced by the jump address

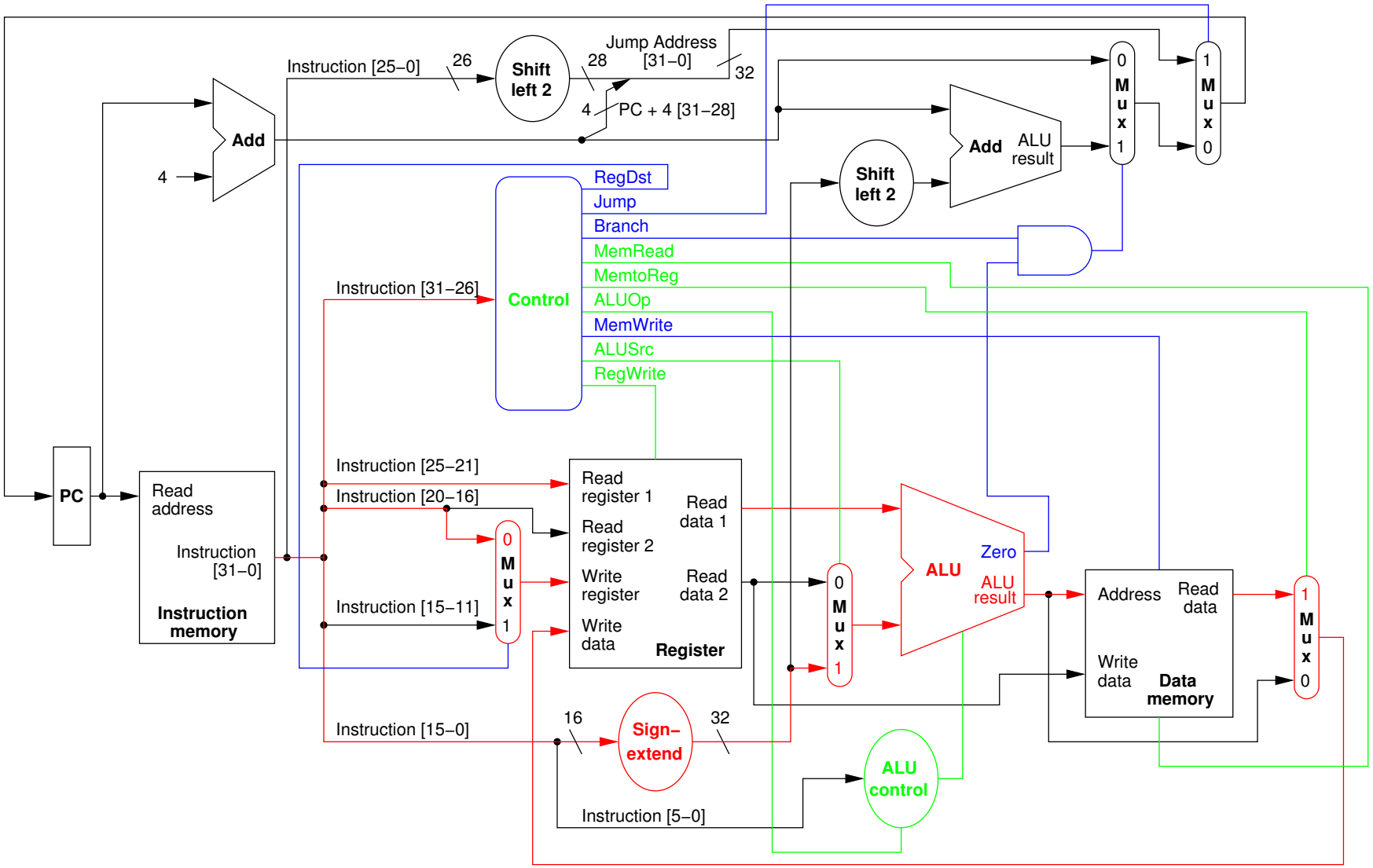
PC for Non-Branch and Jump



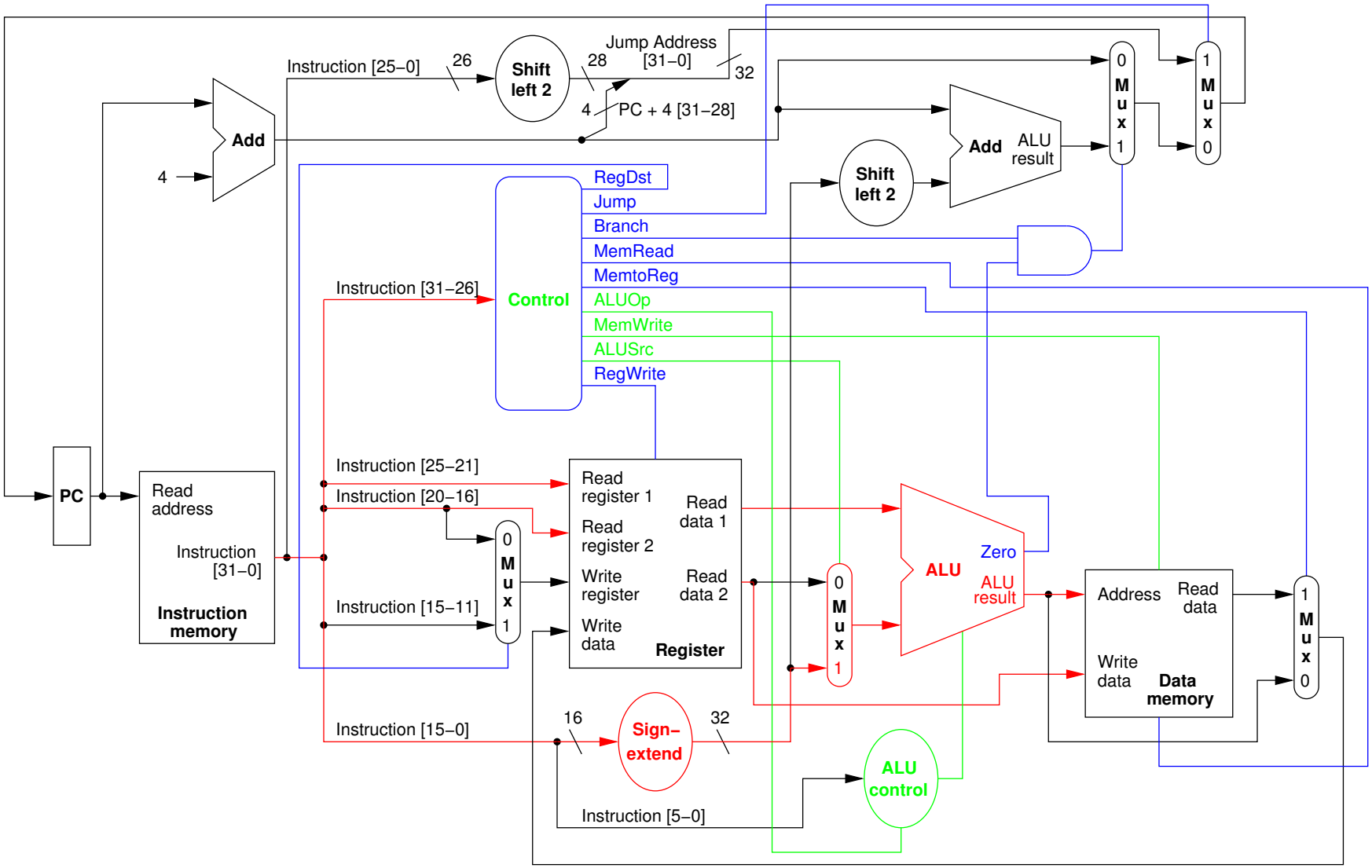
R-type: add, sub, and, or, and slt



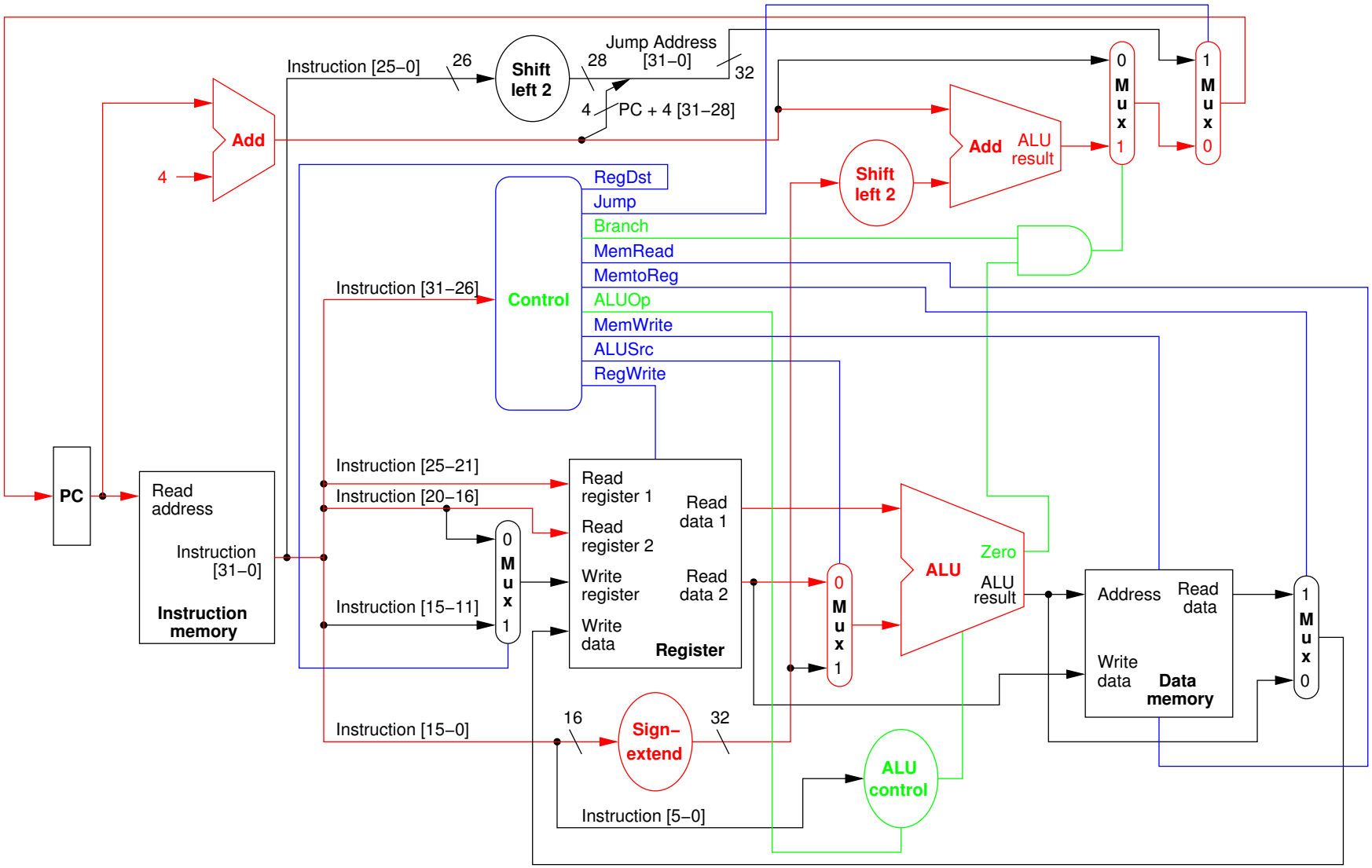
I-type: lw


Add
4
Instruction [25–0]
26
Shift left 2
28
Jump Address [31–0]
32
4
PC + 4 [31–28]
0 M u x 1
1 M u x 0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
0 M u x 1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Register
0 M u x 1
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
1 M u x 0
Instruction [15–0]
16
Sign–extend
32
ALU control
Instruction [5–0]

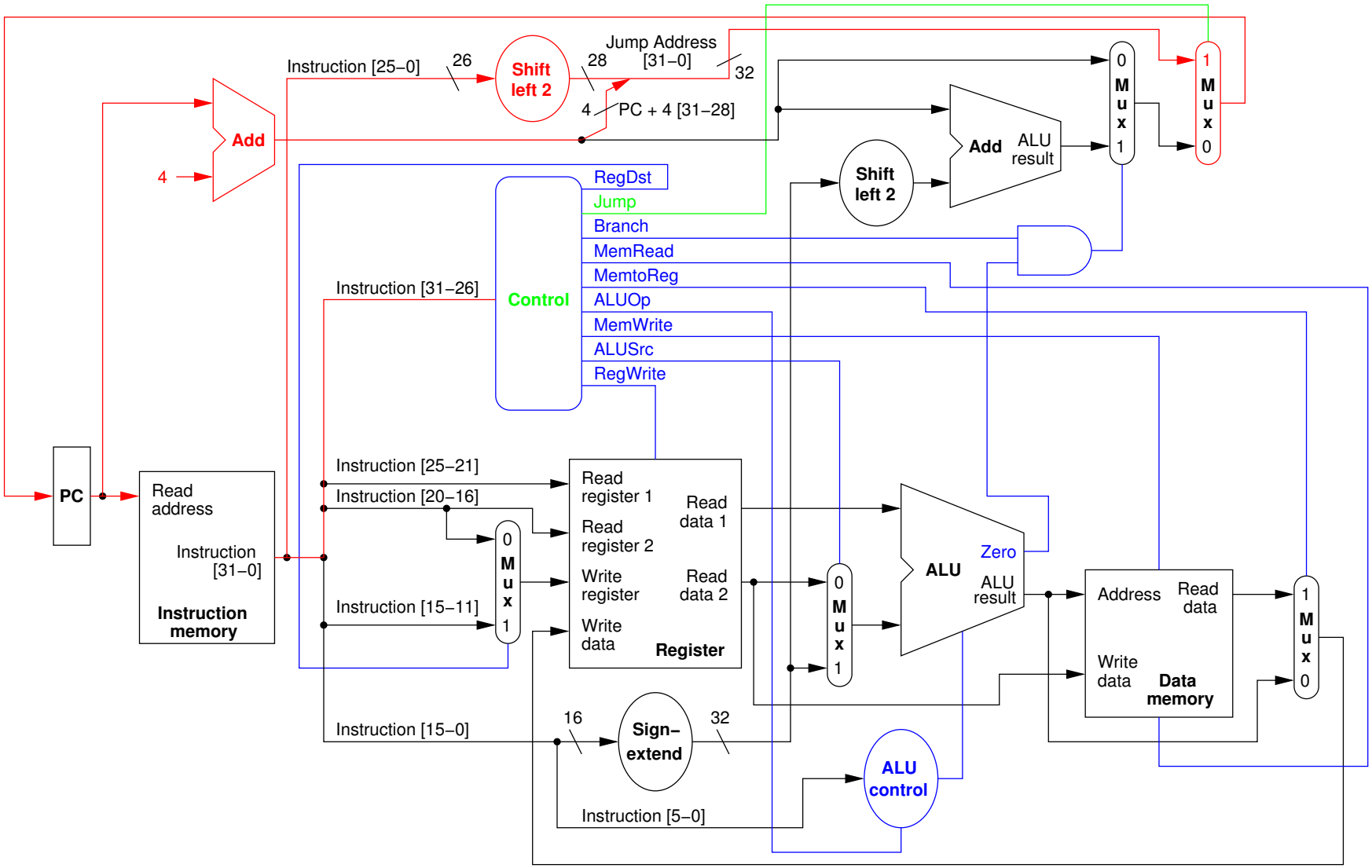
I-type: sw


Instruction [25–0]
26
Shift left 2
28
Jump Address [31–0]
32
4
PC + 4 [31–28]
Add
4
Add
ALU result
Shift left 2
0 M u x 1
1 M u x 0
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
0 M u x 1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Register
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
16
Sign–extend
32
ALU control
Instruction [5–0]

PC Branch


Add
4
Instruction [25–0]
26
Shift left 2
28
Jump Address [31–0]
32
4
PC + 4 [31–28]
0 M u x 1
1 M u x 0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
0 M u x 1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Register
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
16
Sign–extend
32
ALU control
Instruction [5–0]

PC Jump


Add
4
Instruction [25–0]
26
Shift left 2
28
Jump Address [31–0]
32
4
PC + 4 [31–28]
0 M u x 1
1 M u x 0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Register
16
Sign–extend
32
ALU
Zero
ALU result
ALU control
Address
Read data
Write data
Data memory

Implementing the Main Control Unit

- Suppose we want to implement the main control unit to support our small set of instructions.
- Let's consider their Op code:

Instruction	Op
`add`	000000
`sub`	000000
`and`	000000
`or`	000000
`slt`	000000
`beq`	000100
`lw`	100011
`sw`	101011
`j`	000010

- Let $Op_5\,Op_4\,Op_3\,Op_2\,Op_1\,Op_0$ be a 6-bit value of Op field

RegDst

Instruction	Op
add	000000
sub	000000
and	000000
or	000000
slt	000000
beq	000100
lw	100011
sw	101011
j	000010

- Set RegDst to 1 if we want rd field to go to Write Register
  - add, sub, and, or, and slt
- Set RegDst to 0 if we want rt field to go to Write Register
  - lw
- beq, sw, and j do not need to write to register file (ignore)
- Note that among those 6 instructions that need RegDst, only lw has Op_5 equals to 1
- Thus, RegDst $= Op_5'$
  - RegDst $= Op_1'$ and RegDst $= Op_0'$ are OK

Jump

Instruction	Op
`add`	000000
`sub`	000000
`and`	000000
`or`	000000
`slt`	000000
`beq`	000100
`lw`	100011
`sw`	101011
`j`	000010

- Set `Jump` to 1 if the instruction is `j`
- Simply

$$\texttt{Jump} = Op_5' Op_4' Op_3' Op_2' Op_1 Op_0'$$

- Note that Op_4 can be ignored since all instructions have Op_4 equals to 0

Branch

Instruction	Op
`add`	000000
`sub`	000000
`and`	000000
`or`	000000
`slt`	000000
`beq`	000100
`lw`	100011
`sw`	101011
`j`	000010

- Set `Branch` to 1 if the instruction is `beq`
- Simply

$$\texttt{Branch} = Op_5' Op_4' Op_3' Op_2 Op_1' Op_0'$$

- Note that Op_4 can be ignored since all instructions have Op_4 equals to 0

MemRead

Instruction	Op
`add`	000000
`sub`	000000
`and`	000000
`or`	000000
`slt`	000000
`beq`	000100
`lw`	100011
`sw`	101011
`j`	000010

- Set `MemRead` to 1 if we want to read data from data memory
- Only the instruction `lw` requires CPU to read data from data memory
- Simply

$$\texttt{MemRead} = Op_5\, Op_4'\, Op_3'\, Op_2'\, Op_1\, Op_0$$

- Note that Op_4 can be ignored since all instructions have Op_4 equals to 0

MemtoReg

Instruction	Op
`add`	000000
`sub`	000000
`and`	000000
`or`	000000
`slt`	000000
`beq`	000100
`lw`	100011
`sw`	101011
`j`	000010

- Set `MemtoReg` to 1 if we want data from data memory to be written into register file
- Only the instruction `lw` requires data from data memory
- Simply

$$\texttt{MemtoReg} = Op_5\, Op_4'\, Op_3'\, Op_2'\, Op_1\, Op_0$$

- Note that Op_4 can be ignored since all instructions have Op_4 equals to 0

MemWrite

Instruction	Op
`add`	000000
`sub`	000000
`and`	000000
`or`	000000
`slt`	000000
`beq`	000100
`lw`	100011
`sw`	101011
`j`	000010

- Set `MemWrite` to 1 if we want to write data to data memory
- Only the instruction `sw` requires to write data to data memory
- Simply

$$\texttt{MemWrite} = Op_5 Op_4' Op_3 Op_2' Op_1 Op_0$$

- Note that Op_4 can be ignored since all instructions have Op_4 equals to 0

ALUSrc

Instruction	Op
add	000000
sub	000000
and	000000
or	000000
slt	000000
beq	000100
lw	100011
sw	101011
j	000010

- Set `ALUSrc` to 0 if we want the second operand of the ALU to come from the `read data 2` of the register file
  - `add`, `sub`, `and`, `or`, `slt`, `beq`
- Set `ALUSrc` to 1 if we want the second operand of the ALU to come from sign-extended immediate field of an instruction
  - `lw` and `sw`
- Note that among those 7 instructions that need `ALUSrc`, only `lw` and `sw` has Op_5 equals to 1
- Thus, `ALUSrc` $= Op_5$
  - `ALUSrc` $= Op_1$ and `ALUSrc` $= Op_0$ are OK

RegWrite

Instruction	Op
`add`	000000
`sub`	000000
`and`	000000
`or`	000000
`slt`	000000
`beq`	000100
`lw`	100011
`sw`	101011
`j`	000010

- Set `RegWrite` to 1 if we want write data to the register file
  - `add`, `sub`, `and`, `or`, `slt`, and `lw`
- Set `RegWrite` to 0 if we do not want to write data to register file
  - `beq`, `sw`, and `j`
- Thus

$$\texttt{RegWrite} = Op_5' Op_4' Op_3' Op_2' Op_1' Op_0' + Op_6 Op_5' Op_4' Op_3' Op_1 Op_0$$