EEL 4768
Computer Architecture

Single-Cycle Datapath

Outline

- Exercises

Exercise 1: Add sll

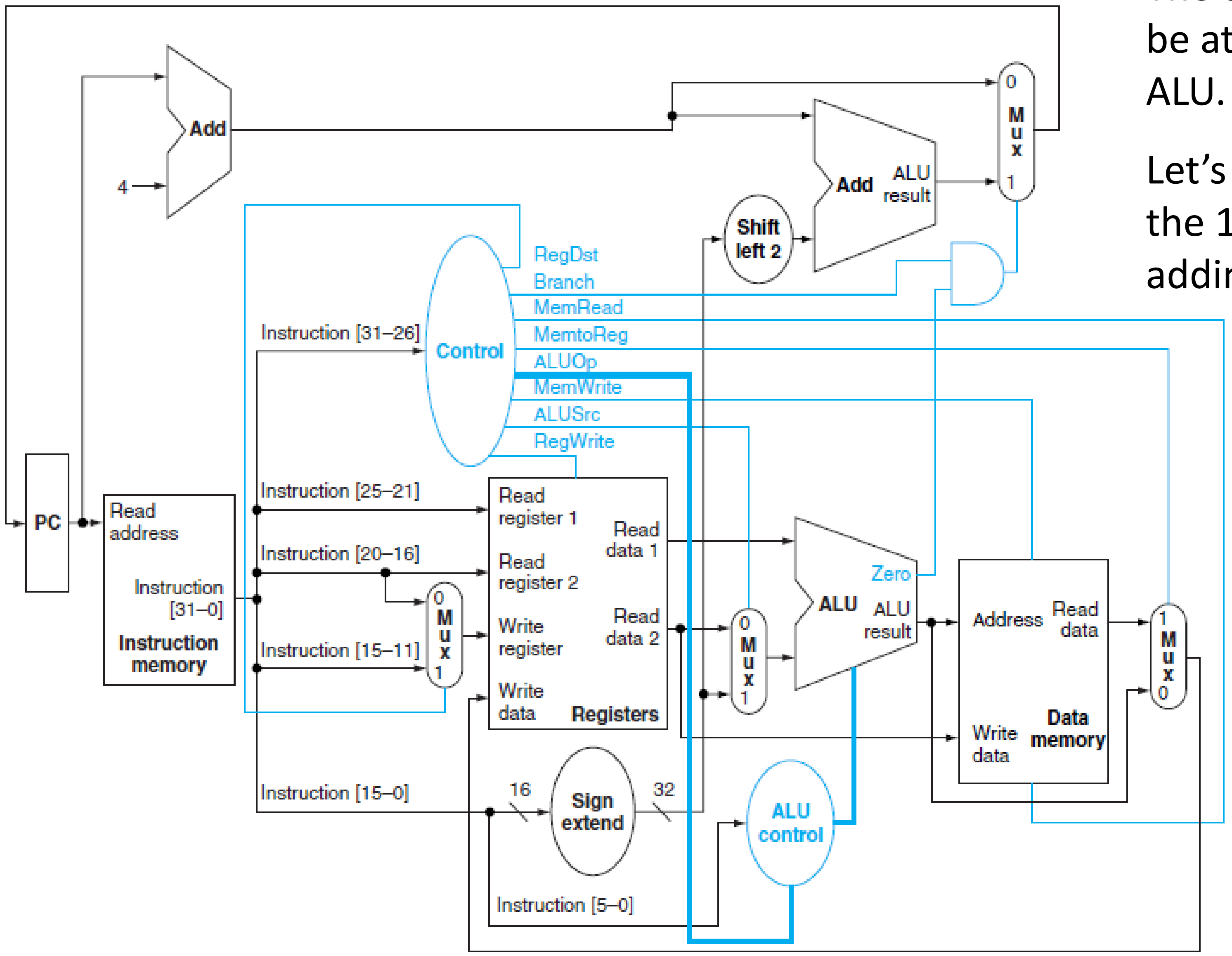
- Modify the single-cycle datapath to add the Shift Left Logical (sll) instruction

- The ‘sll’ instruction uses the R-type format
- Example: sll $t0, $t1, 2 # shift $t1 by 2 bits → $t0

000000	00000	$t1	$t0	2	000000
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Exercise 1: Add sll (cont'd)

- We could add a shifter circuit… but it's better to use the ALU

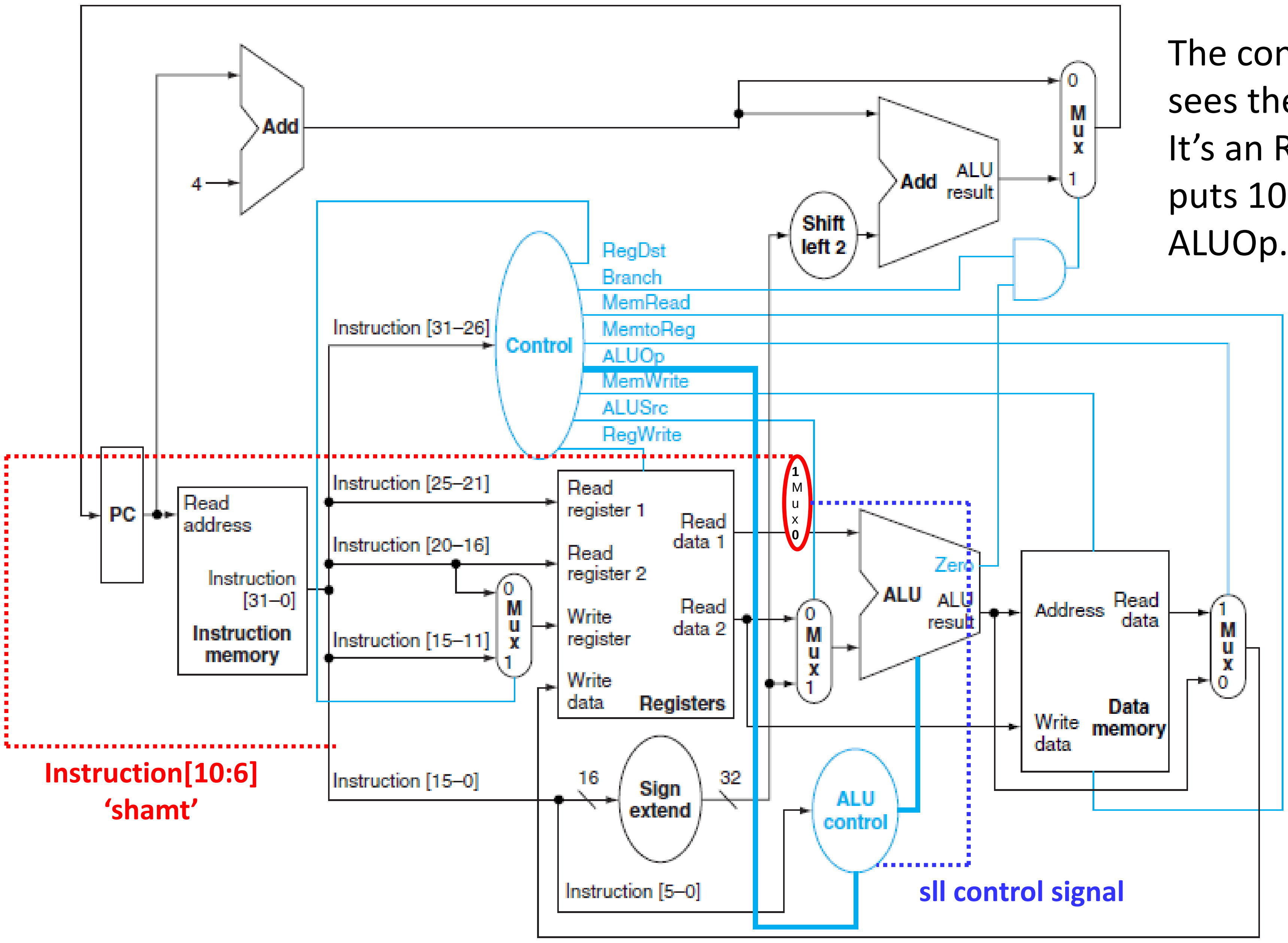


The data to shift, in 'rt' will be at the 2nd input of the ALU.

Let's give the shift amount at the 1st input of the ALU, by adding a multiplexer.

Exercise 1: Add sll

RegDst=1, Branch=0, MemRead=0, MemtoReg=0, ALUOp=10 (R-type), MemWrite=0, ALUSrc=0 (1st operand), RegWrite=1, sll=1



The control signal sees the opcode=0. It's an R-type and puts 10 (binary) on ALUOp.

The ALU control gets ALUOp=10, looks at FUNCT fields and sees 0. It recognizes the sll instruction. It sets the 4-bit field to the code of sll.

Exercise 1: Add sll

- We add a ‘shift left’ operation in the ALU and we give it a unique 4-bit code
- Remember, in the ALU, the 1st input is the shift amount, the second input is the data

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR
????	**Shift Left**

Exercise 2: Add bne

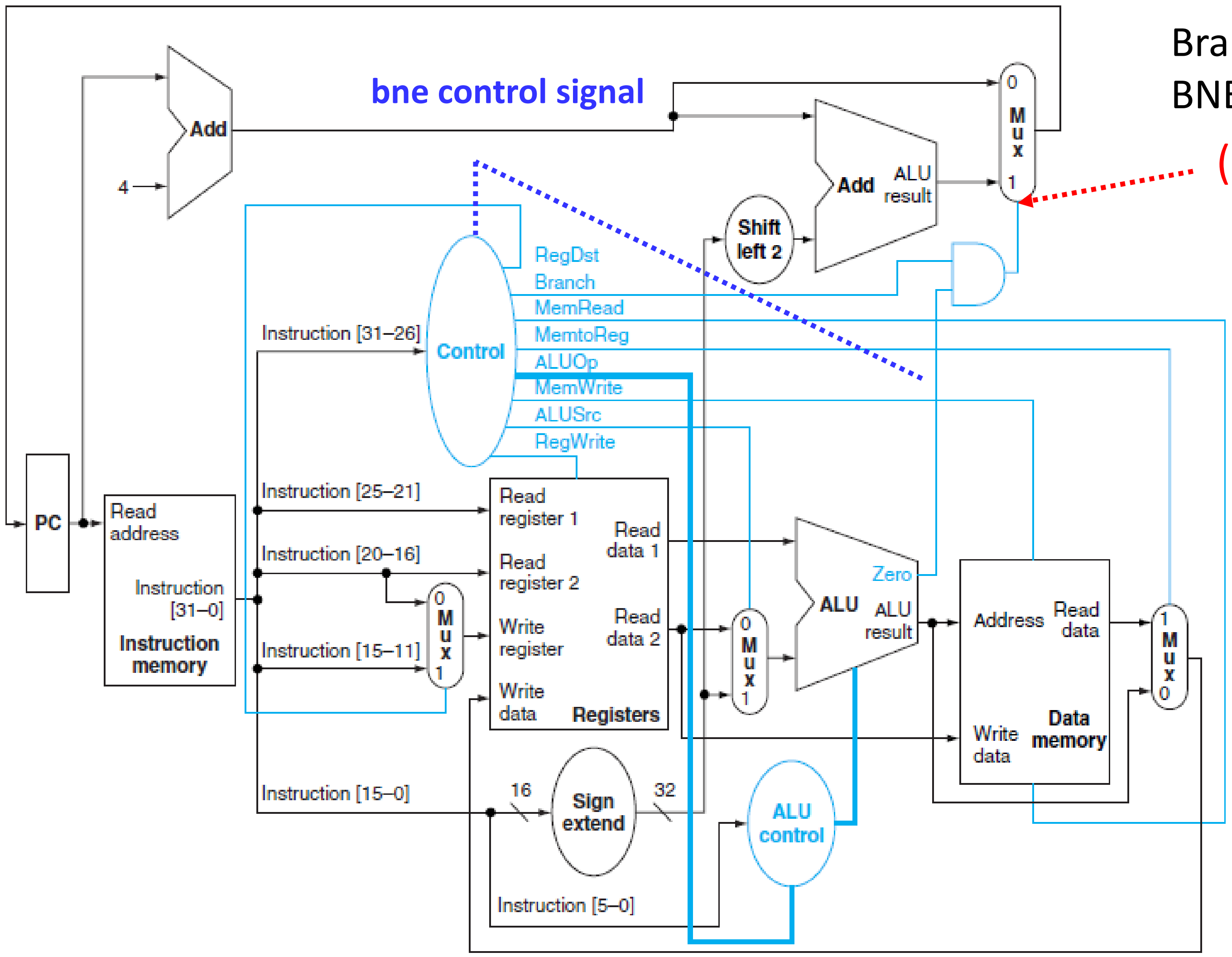
- Modify the single-cycle datapath to support the branch-on-not-equal (BNE) instruction

- BNE instruction has the same format as the BEQ instruction except that it has a different opcode

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Exercise 2: Add bne

* We add a control signal, ‘bne’, that’s equal to 1 only for the BNE instruction.
* The control unit recognizes the opcode of ‘BNE’ and generates bne=1
* We need to add more logic at the selector of the multiplexer that chooses the next PC



Branch if BEQ and Zero signal is 1 or BNE instruction and Zero signal is 0

(Branch&Zero) | (bne & ~zero)

Control Signals

(RegDst=X), (Branch=0), (MemRead=0), (MemtoReg=X), (ALUOp=01 for subtraction), (MemWrite=0), (ALUSrc=0 for 2nd register), (RegWrite=0), (bne=1 to indicate BNE instruction)

Exercise 3: Add jal

Modify the single-cycle datapath so it implements the "jump-and-link" (jal) instruction. The "jal" instruction saves the PC+4 value in the return address ($ra) register.

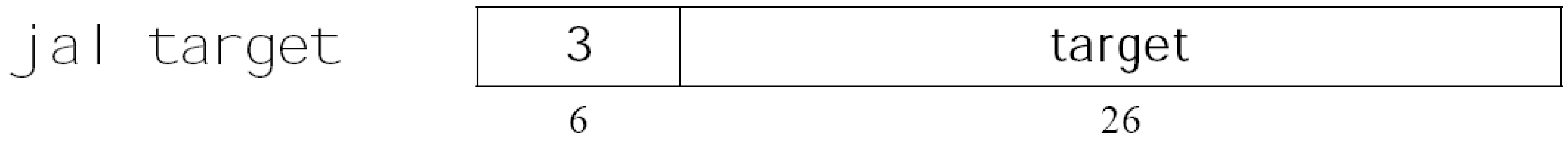
Example of using "jal"

```
96:     addi    $a0, $s1, $s2       # This is an argument to the procedure
100:    addi    $a1, $s3, $s4       # This is an argument to the procedure
104:    jal     FindMax             # Jump to FindMax. Saves 108 in $ra
108:    sw      $v0, 0($t7)         # Print the returned answer
…
        FindMax:
400:    slt     $t0, $a0, $a1       # (if a0<a1, t0=1), (if a0>=a1, t0=0)
404:    beq     $t0, $zero, Returna0
408:    add     $v0, $zero, $a1
412:    j       End
416:    Returna0:       add $v0, $zero, $a0
        End:
420:    jr      $ra                 # Go to address 108, at the "sw" instruction
```

Exercise 3: Add jal (cont'd)

It acts as a jump. The jump address is computed similar to a "j" instruction.

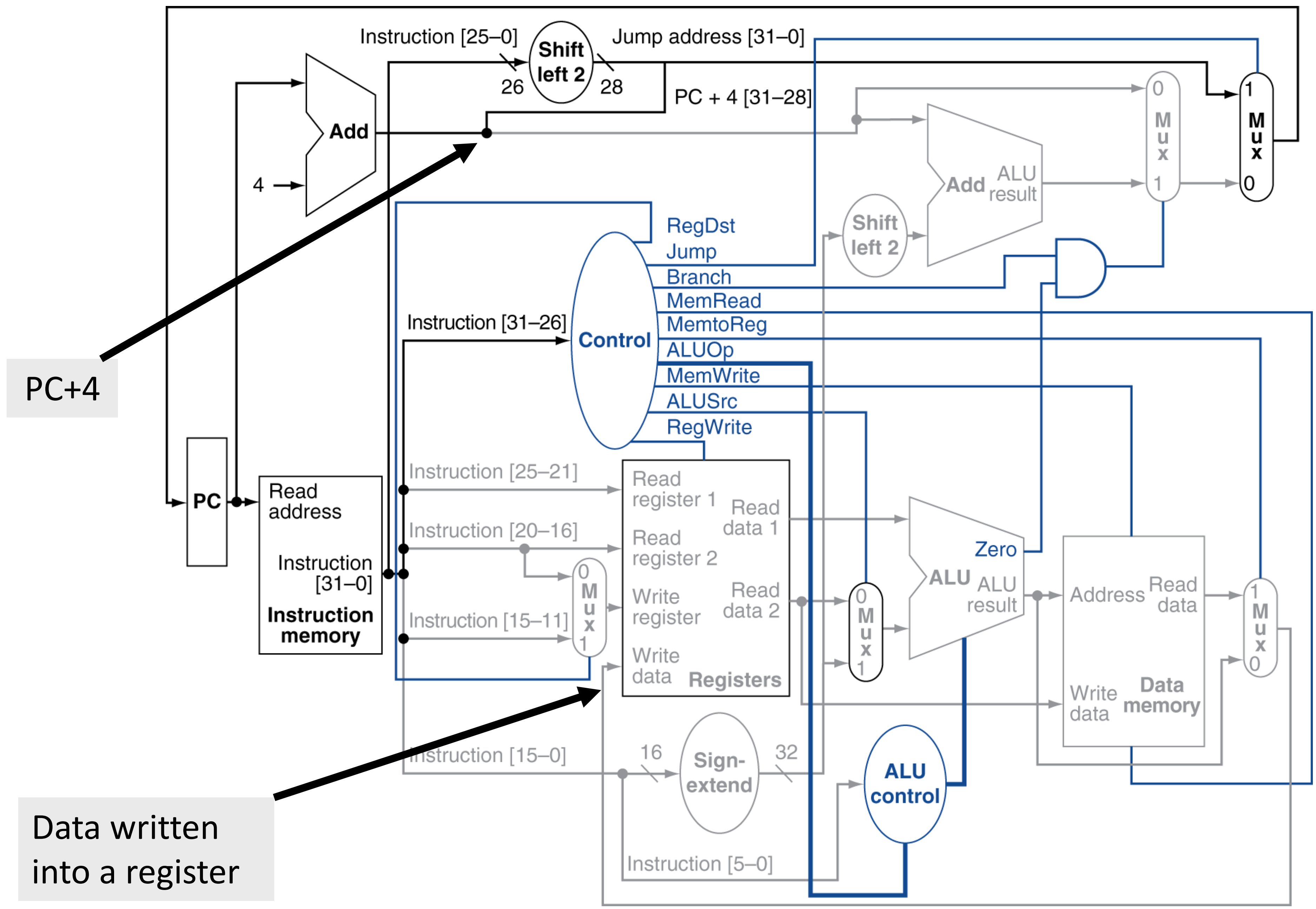
On top of that, it saves PC+4 in the return address register ($ra).



```
…
104:    jal    FindMax        # Jump to FindMax. Saves 108 in $ra
108:    sw     $v0, 0($t7)    # Print the returned answer
…
```

Exercise 3: Add jal (cont'd)

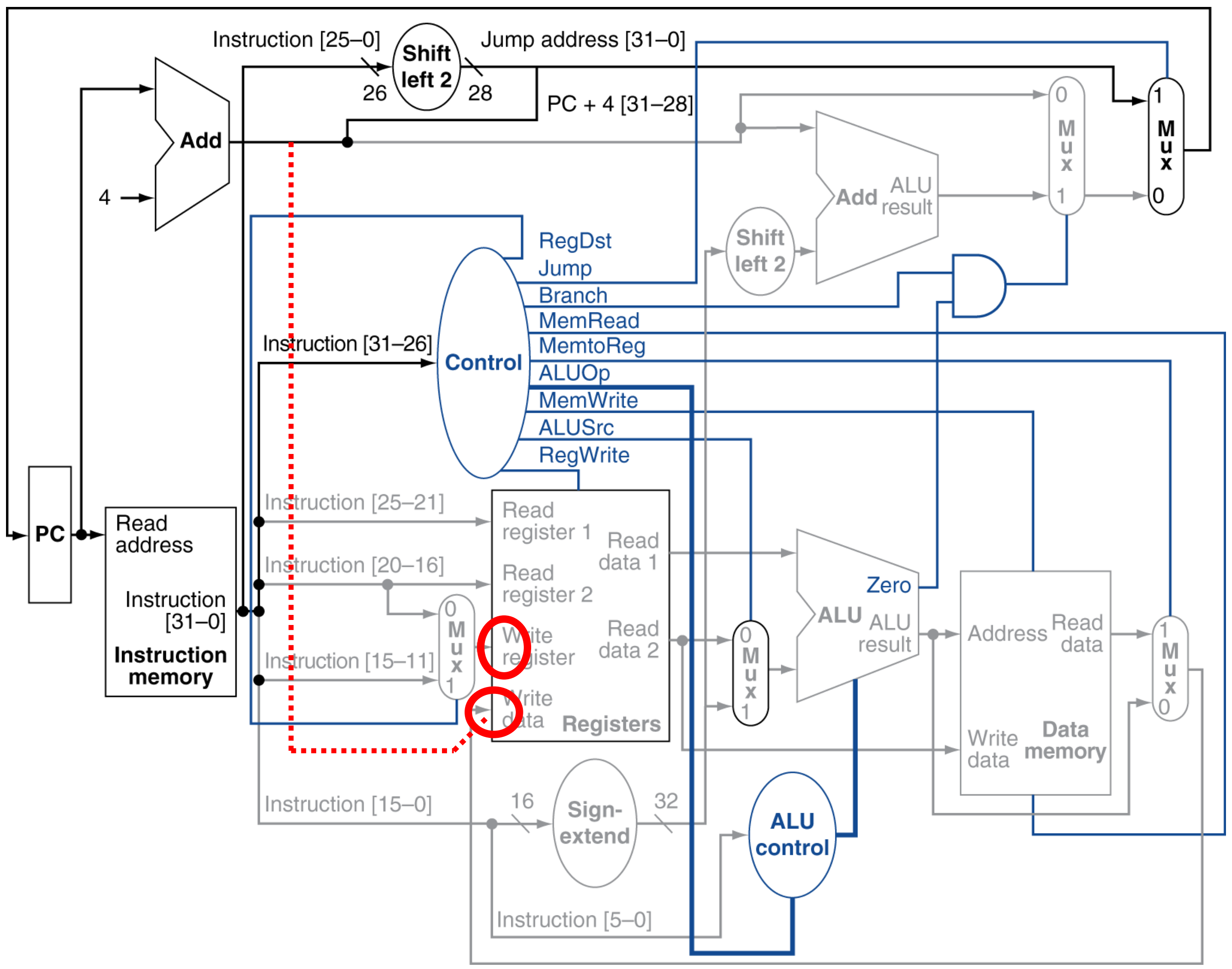
We need the PC+4 value to be saved inside a register.



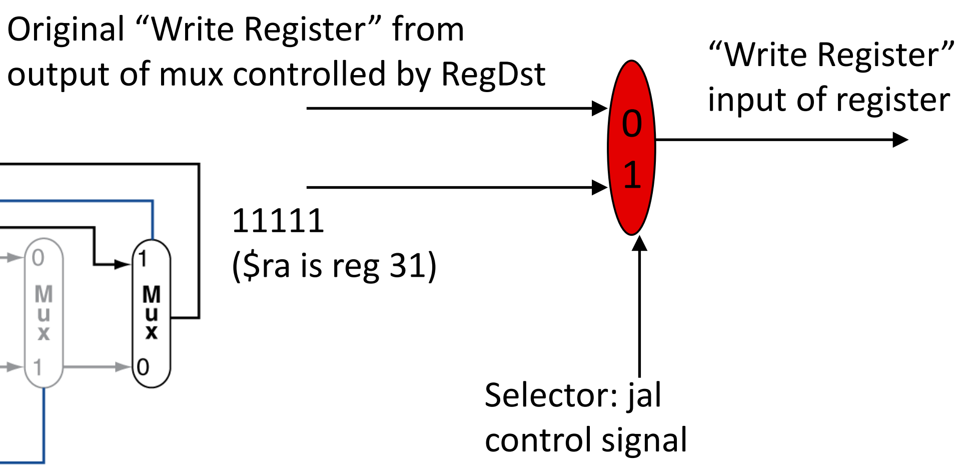
Exercise 3: Add jal (cont'd)

A new line that wires PC+4 to the "Write Data" input of the register file.

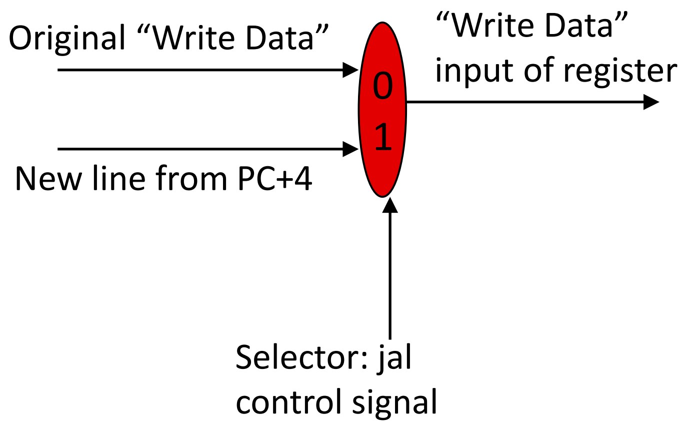
Make sure "Write Register" field is set to the address of $ra. Register $ra is register number 31.



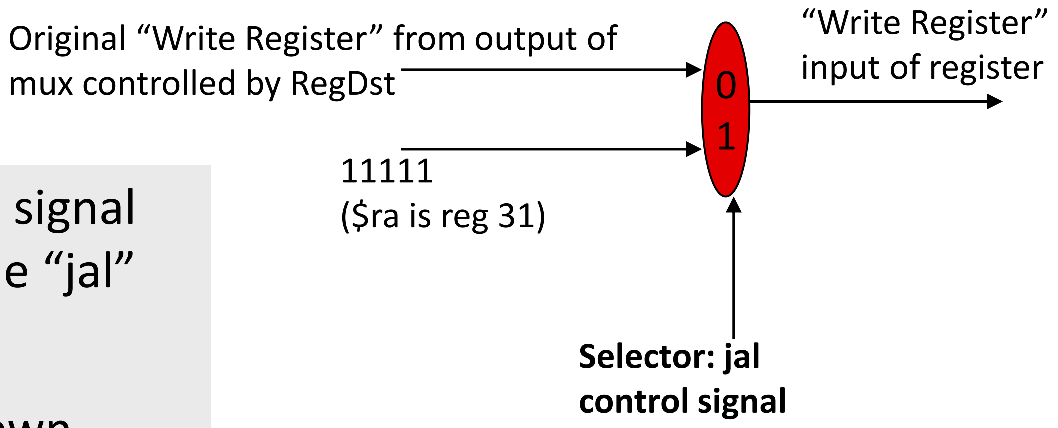
We add two MUXes:



Original "Write Register" from output of mux controlled by RegDst
"Write Register" input of register
0
1
11111 ($ra is reg 31)
Selector: jal control signal

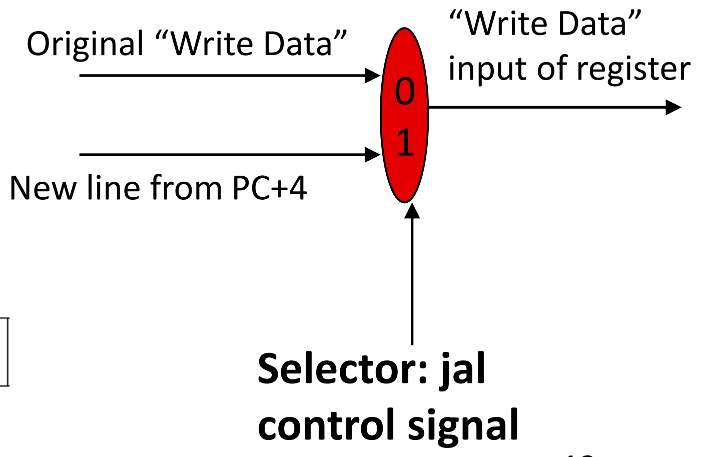


Exercise 3: Add jal (cont'd)


Original "Write Register" from output of mux controlled by RegDst
11111
($ra is reg 31)
0
1
"Write Register" input of register
Selector: jal control signal

The main control unit generates an additional signal called "jal". This signal is equal to 1 only for the "jal" instruction.

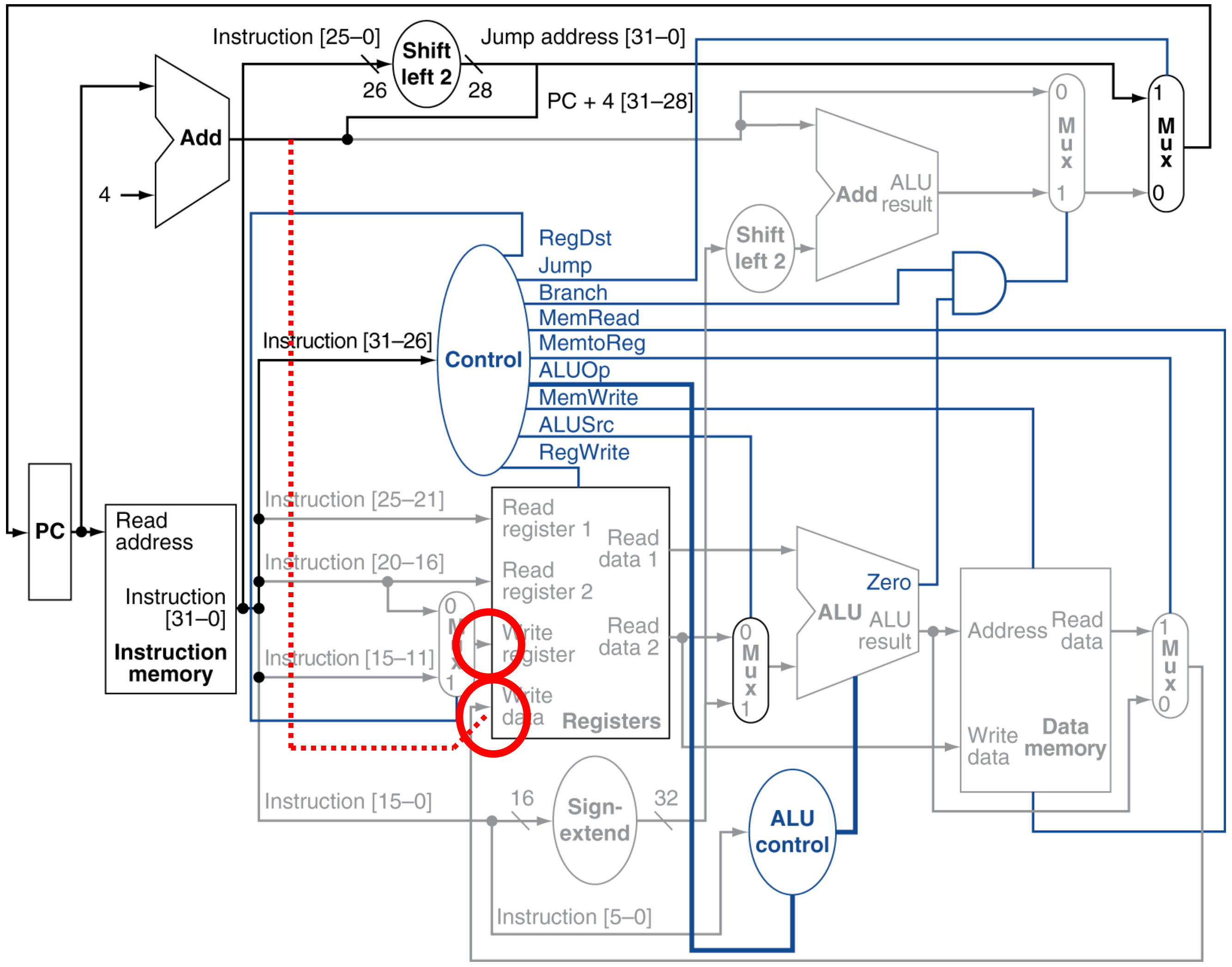
The MIPS format of the "jal" instruction is shown below. It uses opcode=3, so the control unit recognizes it through this opcode.

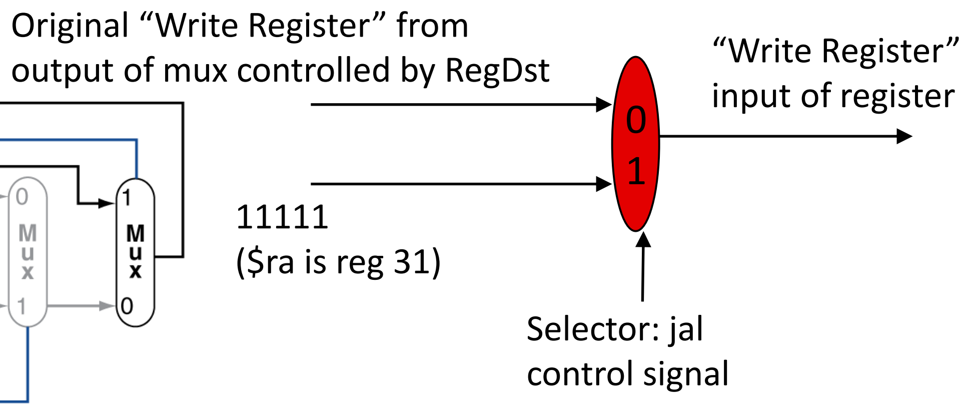


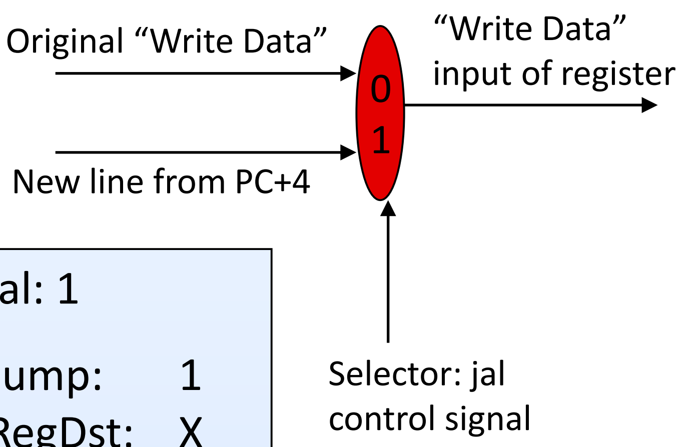
`jal target`

3	target
6	26

Exercise 3: Add jal (cont'd)




Original "Write Register" from output of mux controlled by RegDst
0
1
"Write Register" input of register
11111 ($ra is reg 31)
Selector: jal control signal



jal:	1
Jump:	1
RegDst:	X
Branch:	0
MemRead:	0
MemtoReg:	X
ALUOp:	XX
MemWrite:	0
ALUSrc:	X
RegWrite:	1

Exercise 4: Add jr

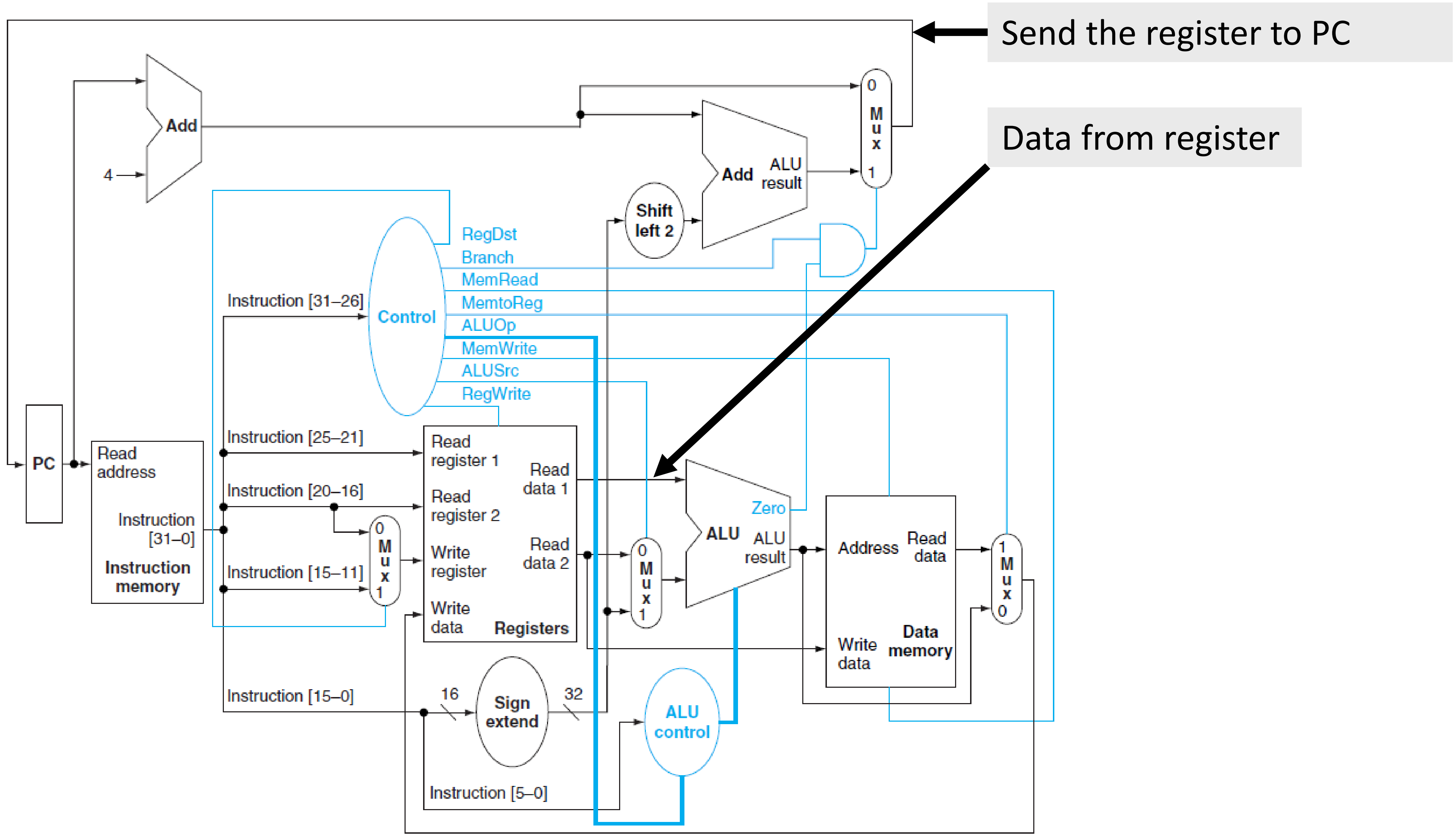
Modify the single-cycle datapath so it implements the “jump-to-register” (jr) instruction. The “jr” jumps to the 32-bit address that’s stored inside a register. Typically, it’s used as “jr $ra” to jump to the return address after finishing a procedure.

- This is the format of the ‘jr’ instruction.
- Note, it uses the opcode=0 (same as R-type)
- So the main control can’t distinguish it from R-type
- The ALU control unit can identify ‘jr’ by looking for Opcode=0 and FUNC=8

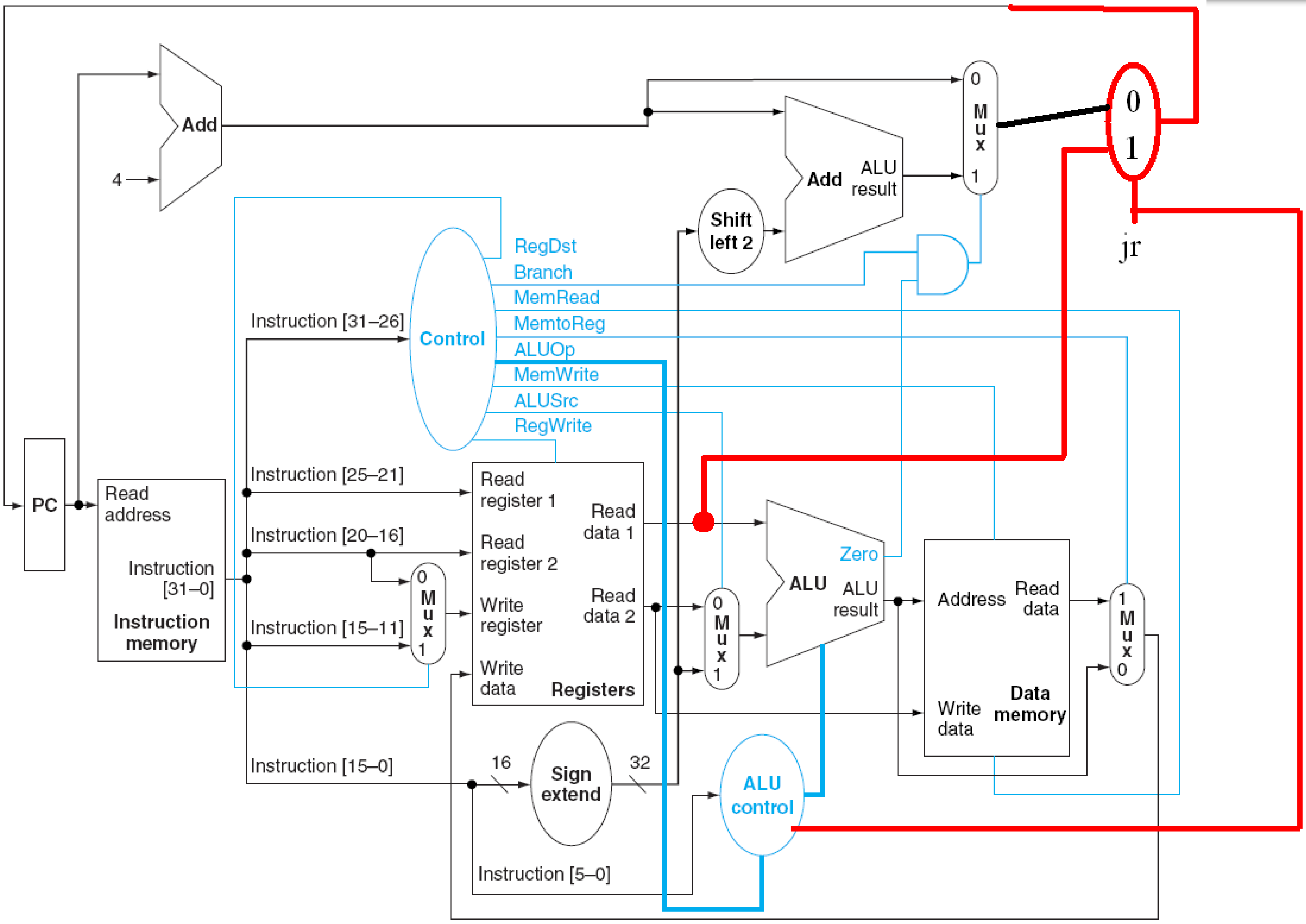
jr rs	0x0	rs	0	0x8
	6	5	16	5

Exercise 4: Add jr (cont'd)

We need to send the 32-bit data from the register to the PC. There's no path to do this, so we'll add a new line.



Exercise 4: Add jr (cont'd)



Exercise 4: Add jr (cont'd)

The main control unit sees Opcode=0 when there's a 'jr' instruction, and cannot distinguish between the 'jr' instruction and other R-type instructions. Accordingly, it sets the control signals as follows:

RegWrite=1 # This can be a problem because 'jr' is not supposed to modify a register. However, the 'rd' field is '00000' (due to the 16-bit zero field), therefore, attempting to write to the $zero register will not result in writing.

RegDst=1 # This sends 'rd' field to 'Write data' in the register, but no writing will happen since rd is the $zero register.

Branch=0 # That's ok for 'jr'

MemRead=0 # That's ok for 'jr'

MemWrite=0 # That's ok for 'jr'

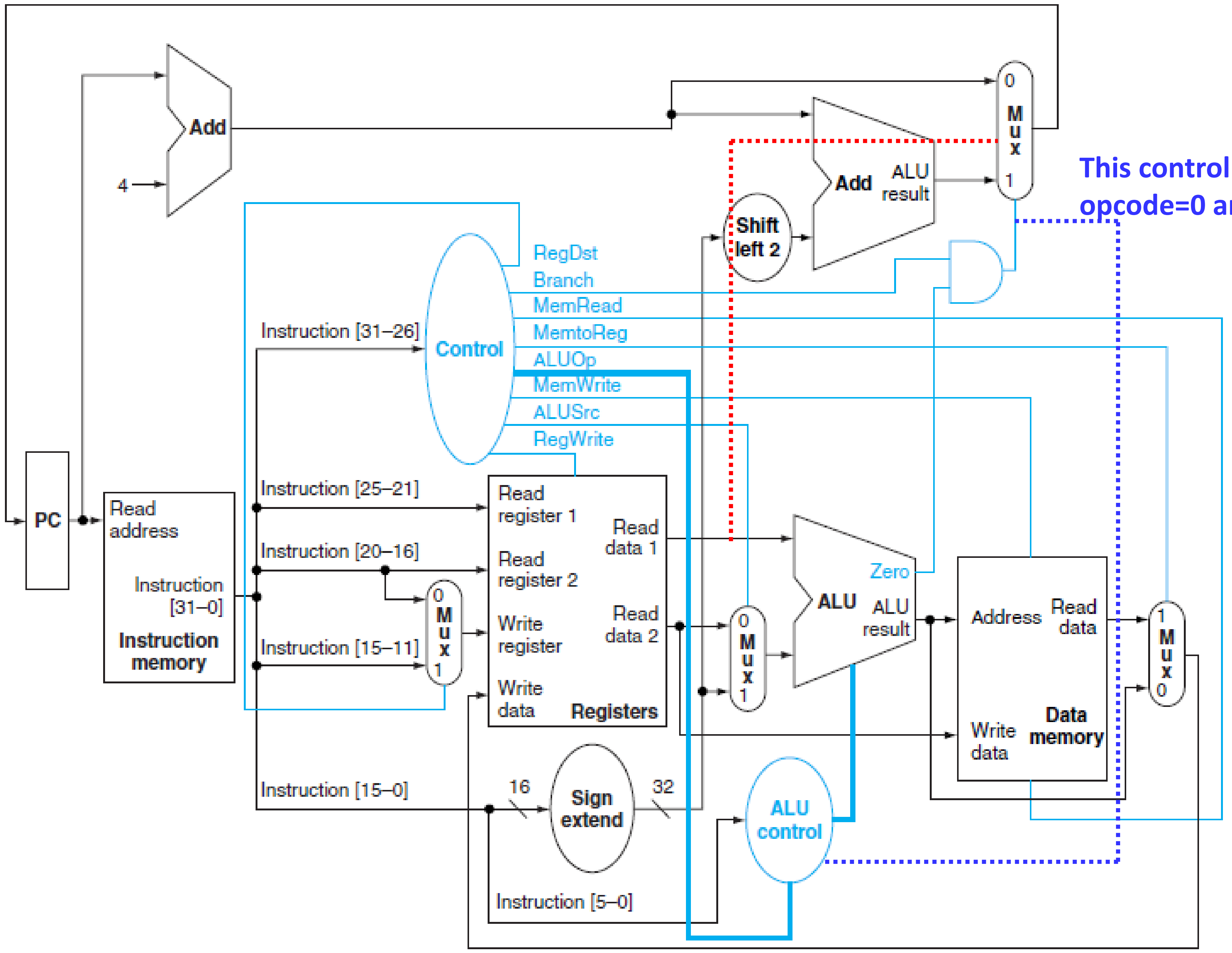
ALUOp=10 # That's ok for 'jr'

ALUSrc=0 # That's ok for 'jr'

So, even if the control signals are set for an R-type, they're still okay for the 'jr' instruction.

jr rs	0x0	rs	0	0x8
	6	5	16	5

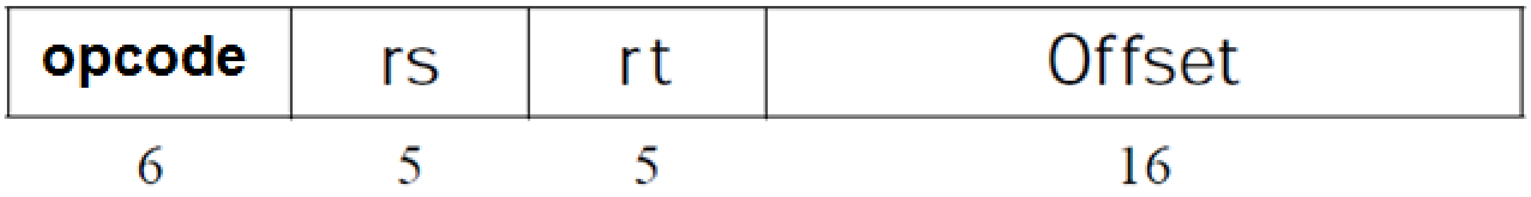
Exercise 4: Add jr (cont'd)



This control signal is 1 when opcode=0 and funct=8

Exercise 5: Add swi

Modify the single-cycle datapath to implement a new instruction called '*store-word-and-increment*' (swi). It has the same format as 'sw' but uses a different opcode. It stores a register in the memory and it increments the base register by 4.



opcode	rs	rt	Offset
6	5	5	16

Example of use:

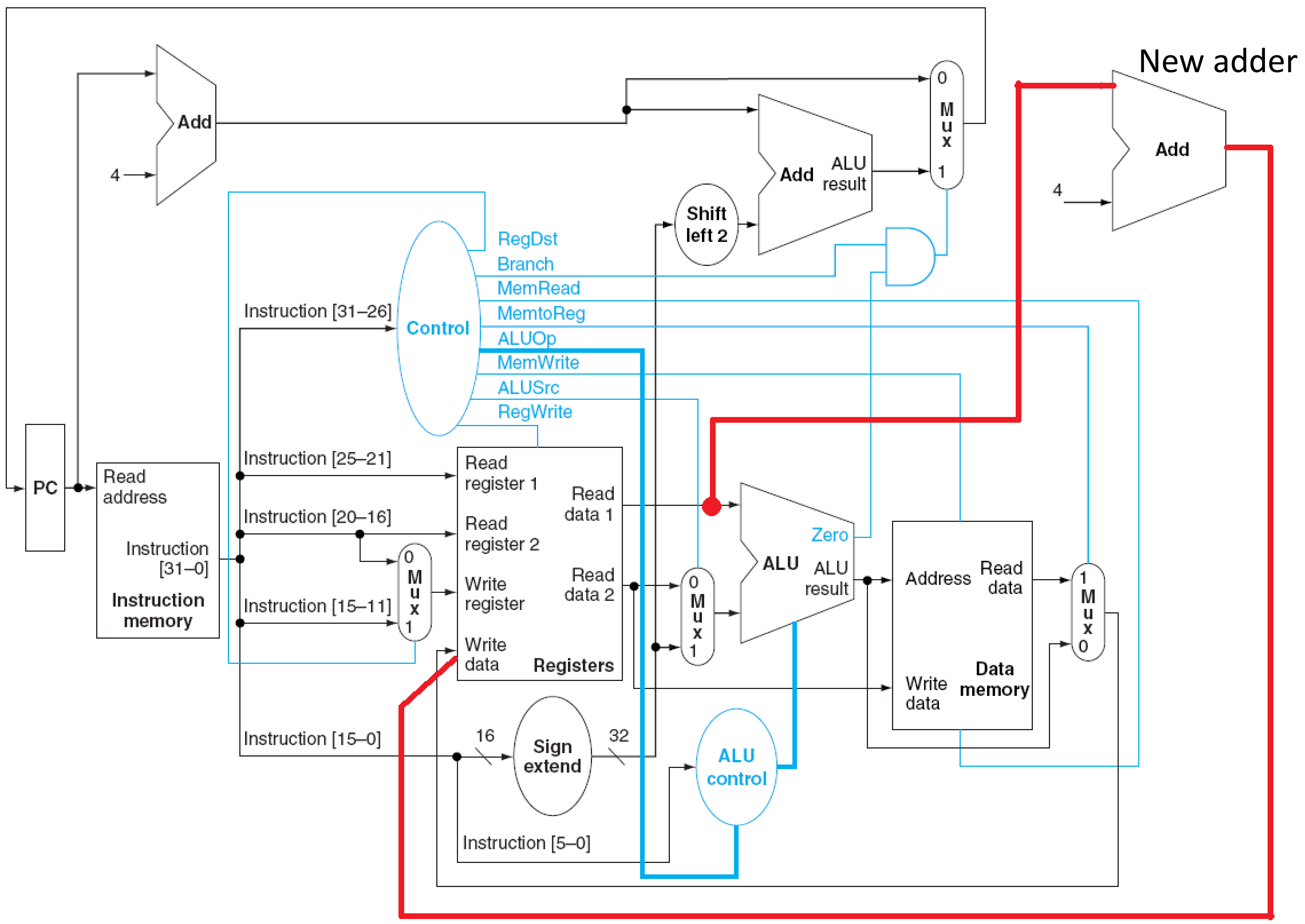
```
swi     $t0, 0($s0)        # Store register $t0 at address ($s0+0)
                           # Also increments $s0 by 4
```

To store $t0, $t1, $t2 in an array with the base in $s0.

```
add     $t7, $s0, $zero    # Copy $s0 into $t7
swi     $t0, 0($t7)        # $t7 is incremented by 4 automatically
swi     $t1, 0($t7)
swi     $t2, 0($t7)
```

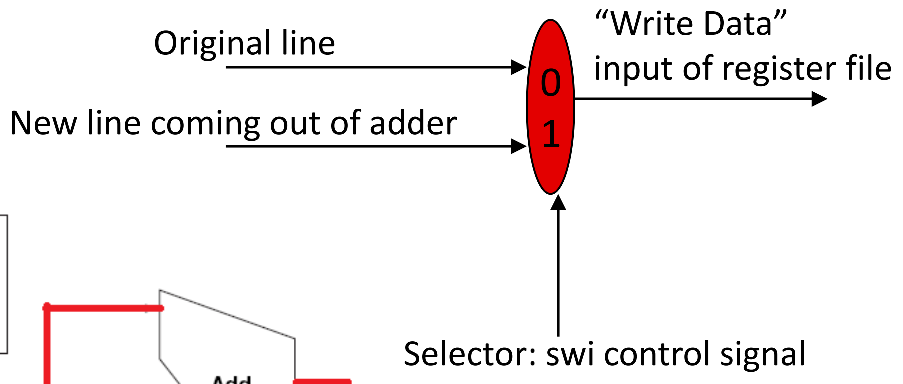
Exercise 5: Add swi (cont'd)

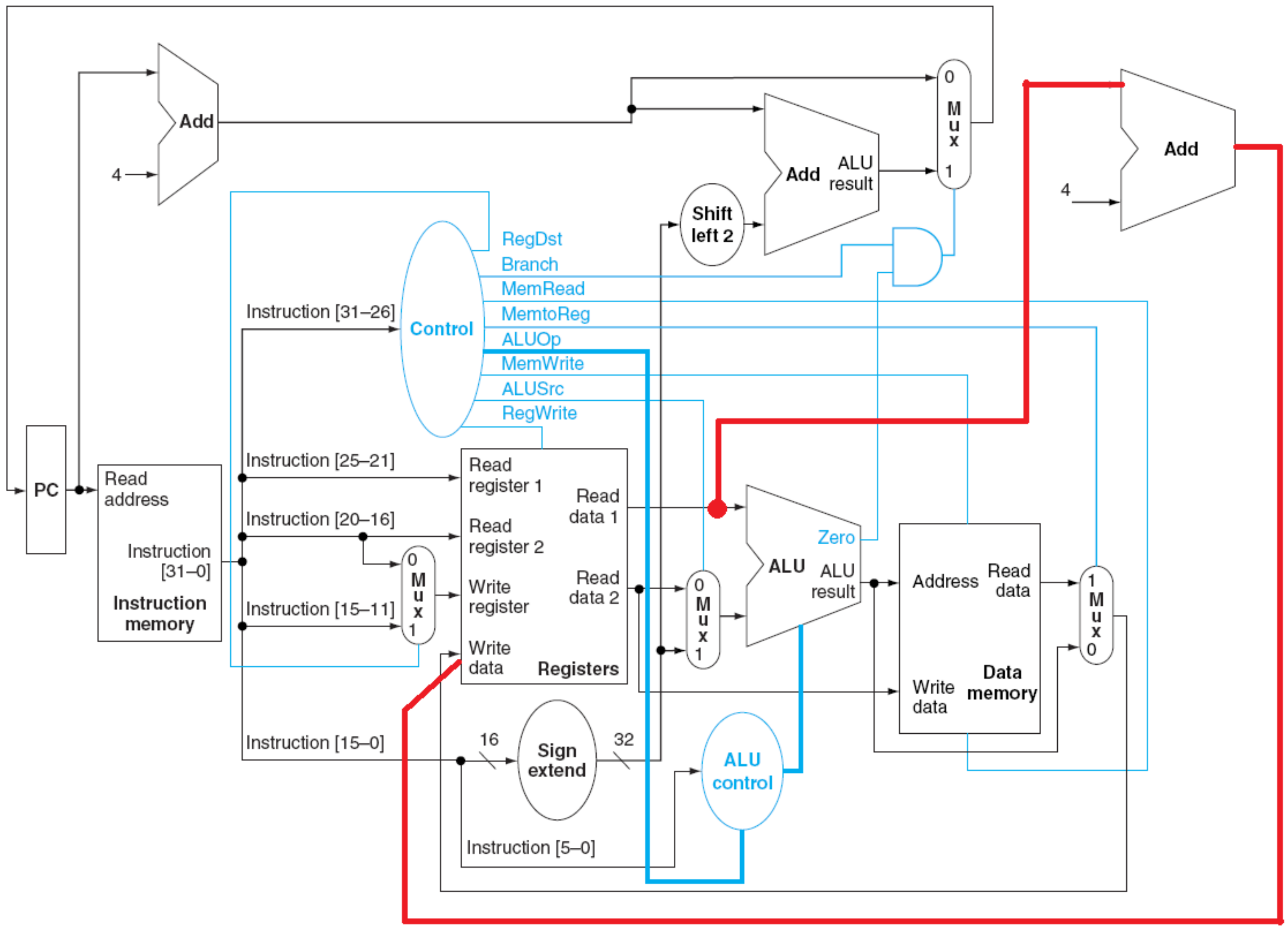
We need a new adder.



Exercise 5: Add swi (cont'd)

We need a new MUX.

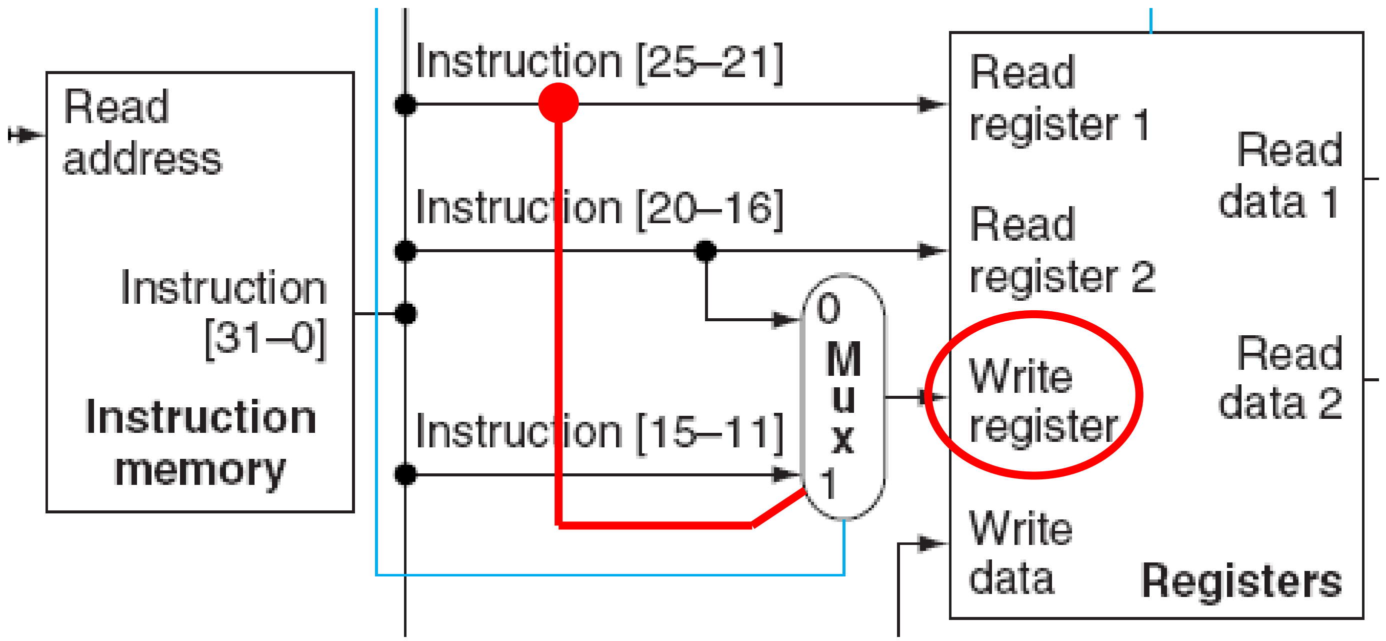

Original line
New line coming out of adder
0
1
"Write Data" input of register file
Selector: swi control signal



Exercise 5: Add swi (cont’d)

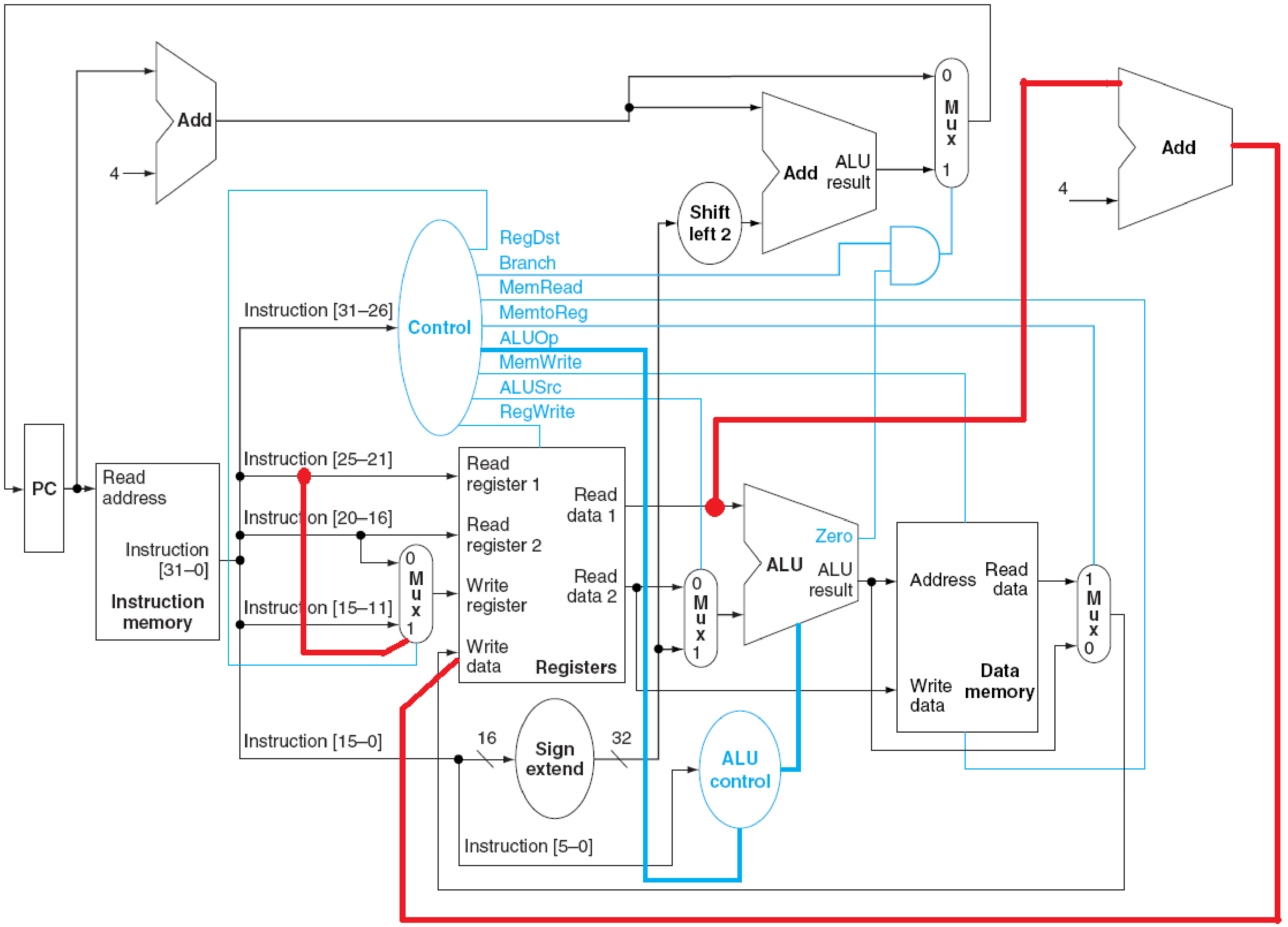
- The datapath is set up to write to fields [20-16] or [15-11]
- However, with ‘swi’ we want to write to field [25-21]
- So we’ll add a new line to do this
- Now we make ‘RegDst’ a 2-bit selector. It’s equal to 2 for ‘swi’

	Base	Data	
opcode	rs	rt	Offset
[31-26]	[25-21]	[20-16]	[15-0]



Exercise 5: Add swi (cont'd)

We added a new line from [25-21] to the register 'Write Data' input



Control signals for 'swi':

(swi=1 new signal)

(RegDst=10),
(Branch=0),
(MemtoReg=X),
(ALUOp=00 add),
(MemWrite=1),
(ALUSrc=1),
(RegWrite=1),

More Exercises

- Modify the datapath to implement ‘load word and increment’ (lwi)

- Modify the datapath to implement an instruction that increments two registers by 4

  inc $t0, $t1

- Modify the datapath to implement the ‘load upper immediate’ (lui) instruction

- This instruction stores ($t0+4) into the memory at address $t1

  swr4 $t0, $t1

More Exercises

- Move-if-zero instruction; if t2=0, then t0=t1; otherwise, no change

  movz $t0, $t1, $t2

- Set the bit #20 in register $t0

  sbit $t0, 20

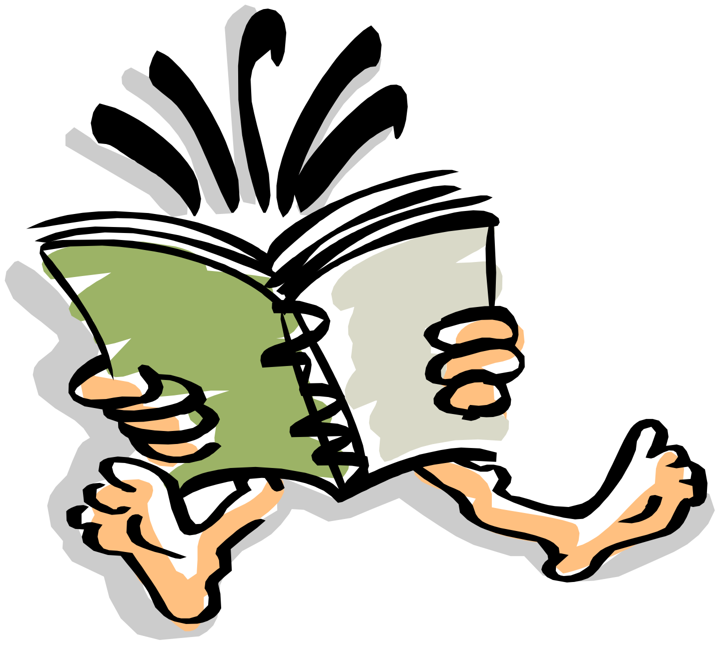
- Clear the bit #20 in register $t0

  cbit $t0, 20

- The absolute value of $t0 is stored in $t1

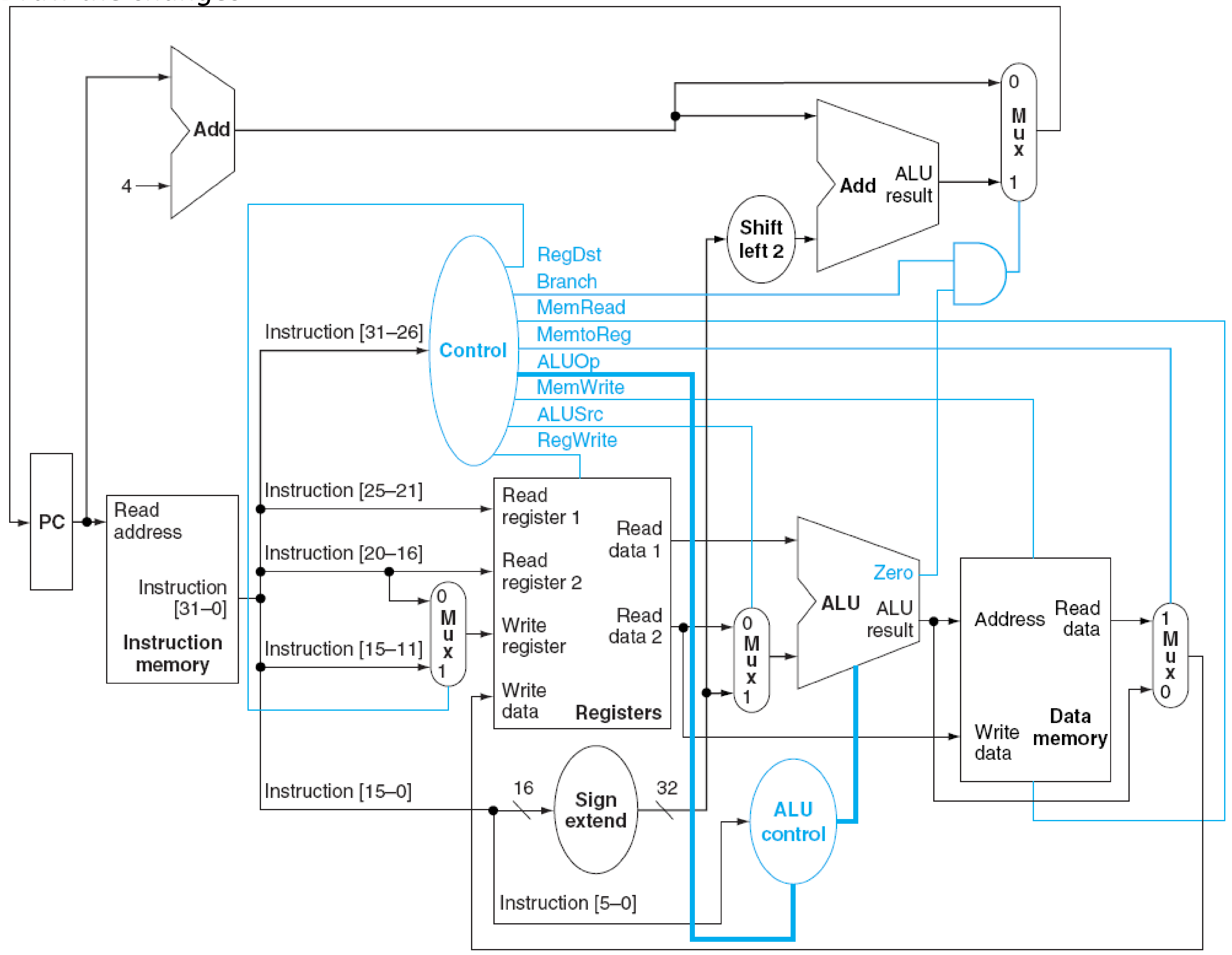
  abs $t1, $t0

Readings

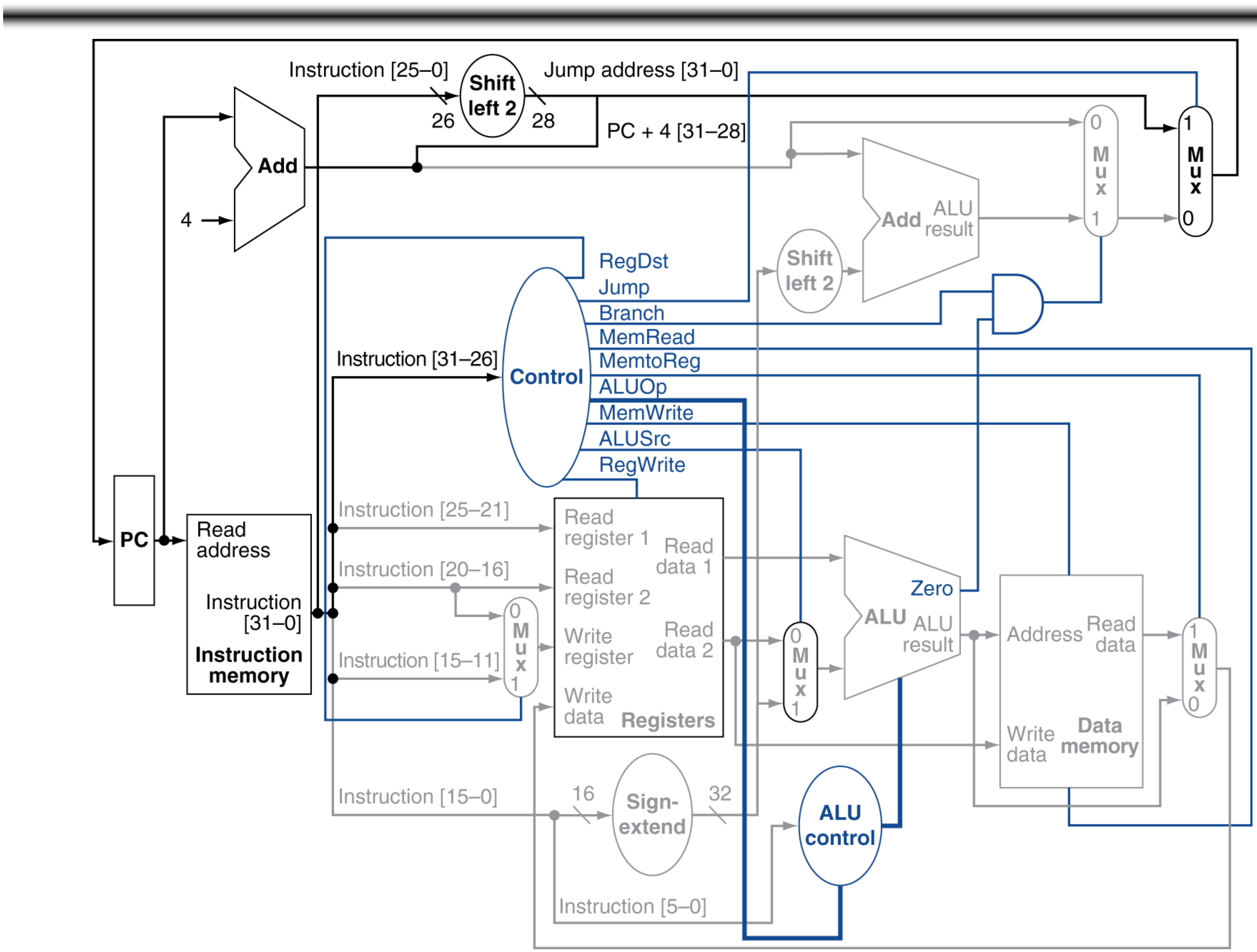


- H&P COD
  - Chapter 4

Draw the changes



Draw the changes


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