EEL 4768: Computer Architecture

Single Cycle Datapath

Instructor: Zakhia (Zak) Abichar

Department of Electrical Engineering and Computer Science
University of Central Florida

Figures taken from: "Computer Organization and Design: The Hardware/Software Interface", Patterson & Hennessy, Third Edition

Single Cycle Datapath

- The single-cycle datapath represents a simple implementation
- Each instruction takes one clock cycle to execute
- Therefore, the Clock-Per-Instruction measure is: **CPI = 1**

- For the single-cycle datapath, the CPU performance equation that finds the execution time of a program is:

Time = Instruction Count * CPI * Clock Cycle Time

Time = Instruction Count * Clock Cycle Time

- The advantage of the single-cycle implementation is its simplicity
- The downside of this implementation is that it's not very fast

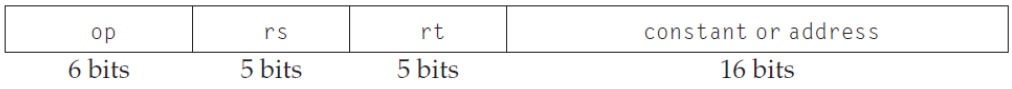
Single Cycle Implementation

- Our implementation will have a subset of the MIPS instructions that are shown in the table below

Type	Instruction	Syntax
Arithmetic	add	add $t0, $t1, $t2
	sub	sub $t0, $t1, $t2
	slt (set-on-less-than)	slt $t0, $t1, $t2 # t0=1 if t1<t2; else t0=0
Logic	and	and $t0, $t1, $t2
	or	or $t0, $t1, $t2
Data Transfer	lw (load word)	lw $t0, 12($t1)
	sw (store word)	sw $t0, 40($t1)
Decision	beq (branch-on-equal)	beq $t0, $t1, Label
	j (jump)	j Label

Review of Instruction Format

- Let's take another look at the instruction format, from the point of view of the CPU
- This format is used for Load Word (lw) and Store Word (sw)
  - 'lw' opcode=35 and 'sw' opcode=43



- What will the CPU do with this instruction?
- It computes the address as:

  `Address = rs+ (constant field sign-extended)`
- For 'lw' is reads from the address and saves into 'rt'
- For 'sw', is saves the data at 'rt' at the computed address

Review of Instruction Format

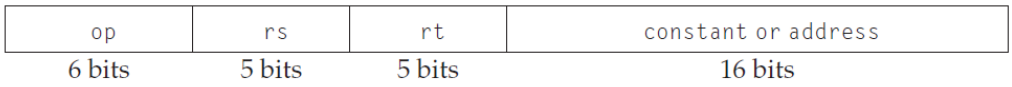
- This format is used by these instructions: add, sub, and, or, slt

- What will the CPU do?
- It reads ‘rs’ and ‘rt’
- It looks at ‘opcode’ and ‘funct’ to determine the operation (in table below)
- The operation is done and the result is saved in ‘rd’

	add	sub	and	or	slt
op field	0	0	0	0	0
funct field	32	34	36	37	42

Review of Instruction Format

- Branch-on-equal (beq) uses the I-Type format below



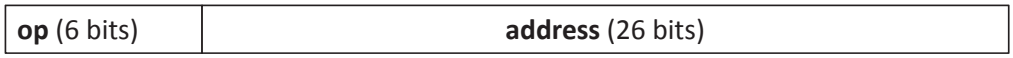
- What does the CPU do for ‘beq’?
- The registers ‘rs’ and ‘rt’ are read and compared
- The branch address is computed as:

```
Address = PC + 4 + shiftedLeft2[sign-extended(constant)]
```

- If the two registers are equal, the PC becomes the branch address

Review of Instruction Format

- This is the format of the jump ('j') instruction



j instruction

- What does the CPU do for 'j'?
- The jump is taken unconditionally

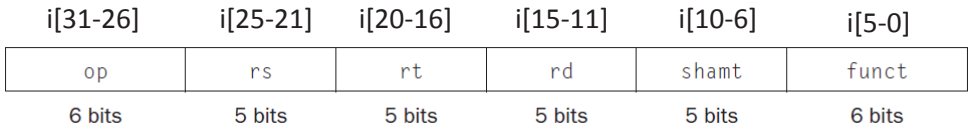
- The jump address is:

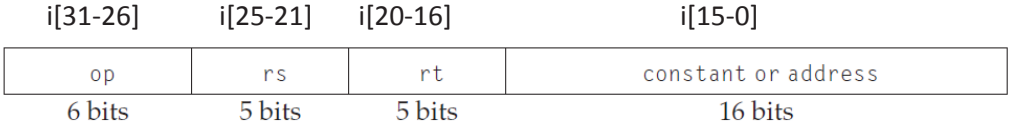
`[leftmost 4 bits of PC+4] [26 bits field in 'j' instruction] [00]`

Instruction Fields Index

- In the datapath, a field is designated by the bit positions it uses

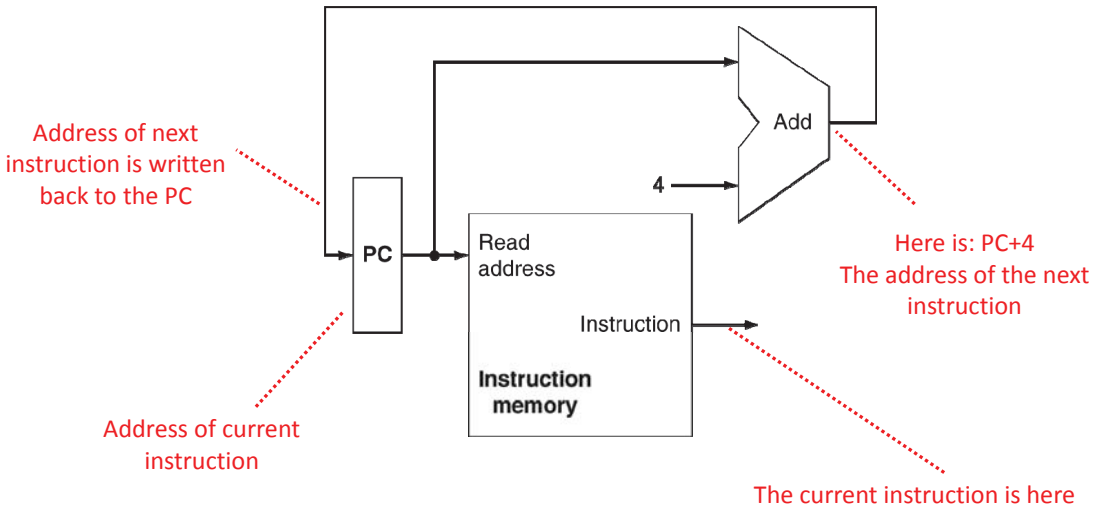
  Eg: i[31-26] designates the leftmost 6 bits of the instruction (the opcode)





Instruction Fetch

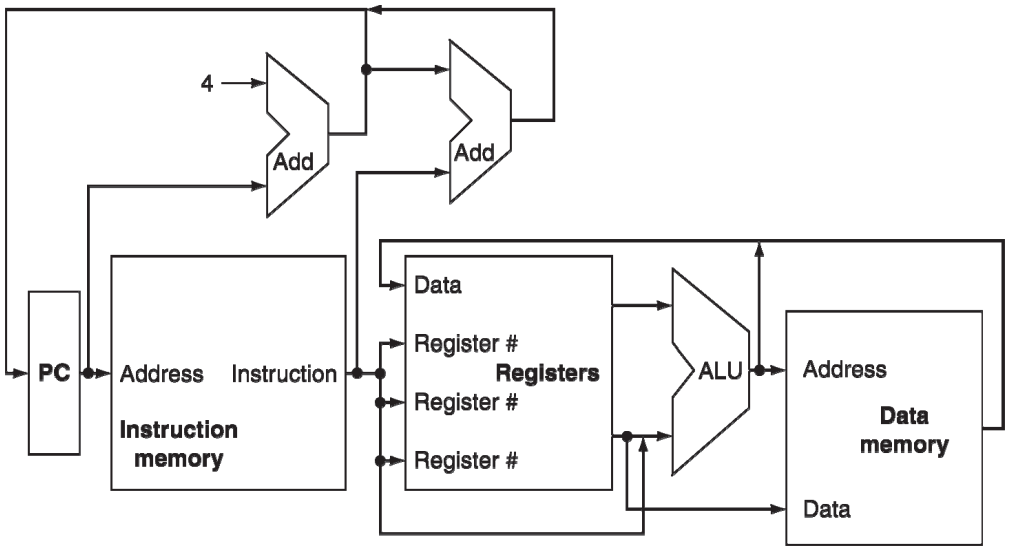
- Now, we will start to build the datapath



Instruction Fetch Stage

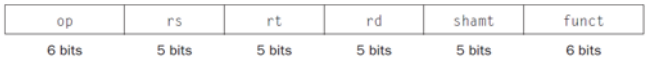
Building the Datapath

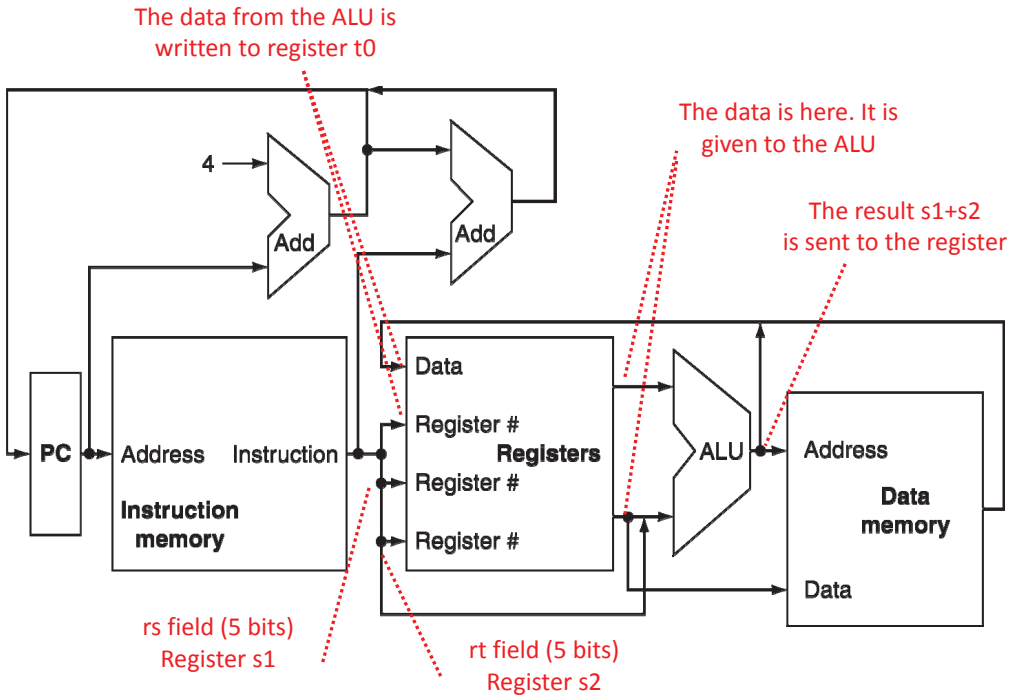
- Now, we add more components to the datapath



R-Type Instruction

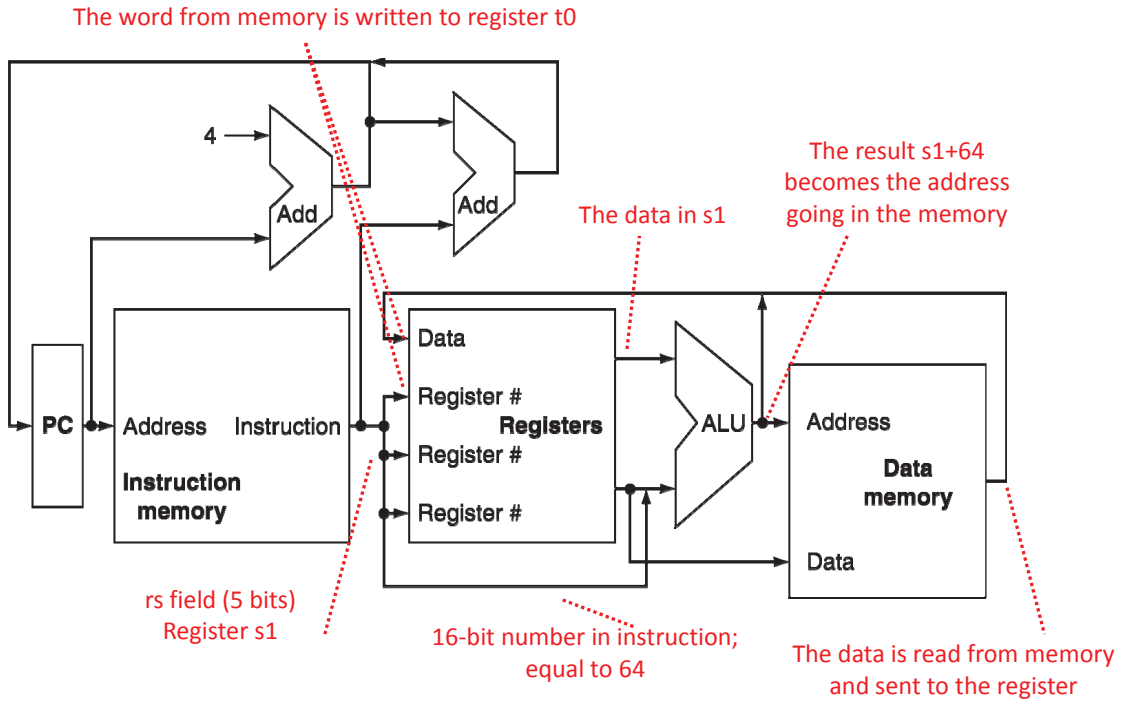
add t0, s1, s2 s1 s2 t0





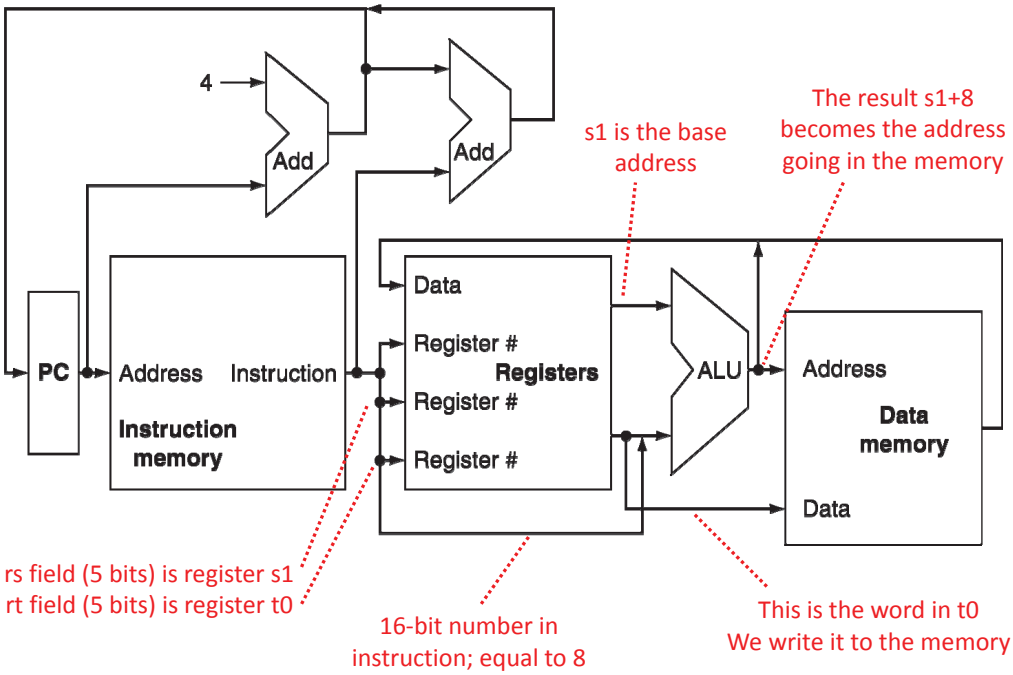
Load Word Instruction

lw t0, (64)s1 s1 t0 0100 0000 (64)



Store Word Instruction

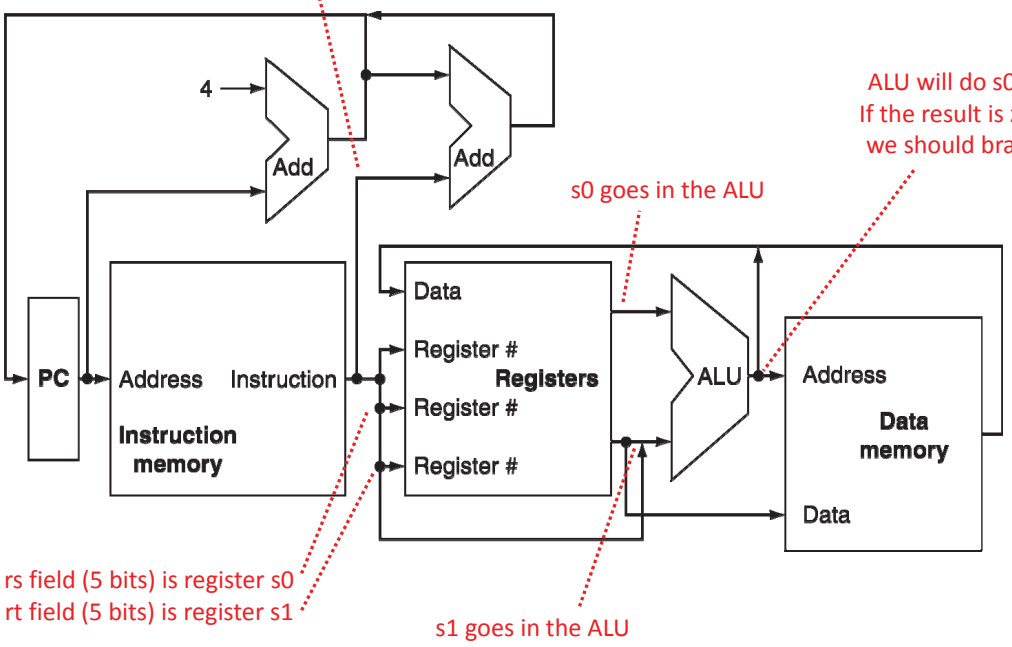
sw t0, (8)s1 s1 t0 0000 1000 (8)



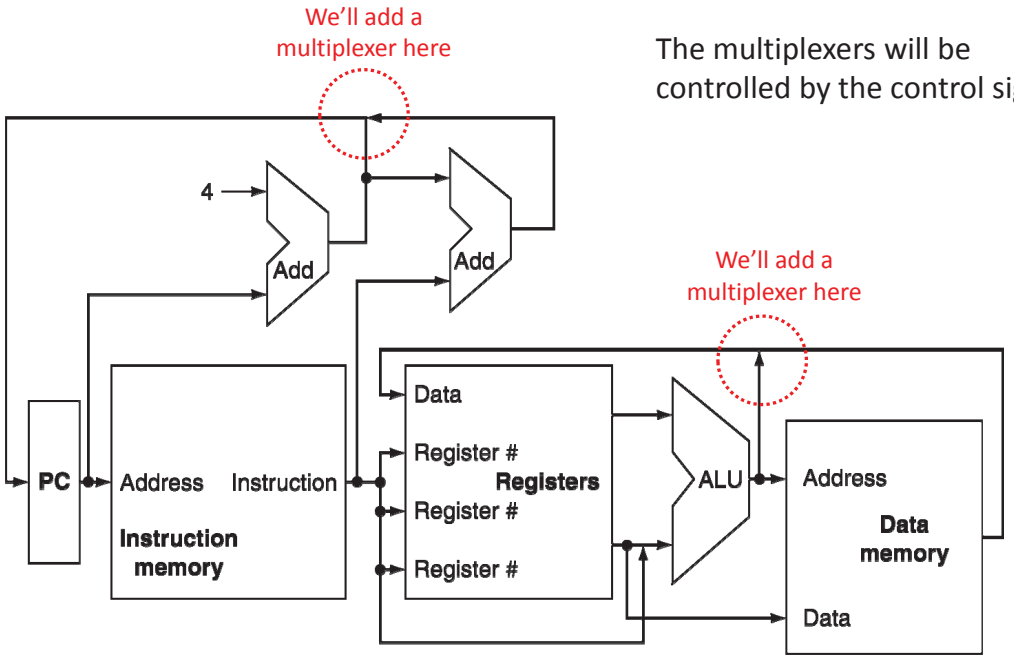
Branch Instruction

beq s0, s1, Loop s0 s1 address offset

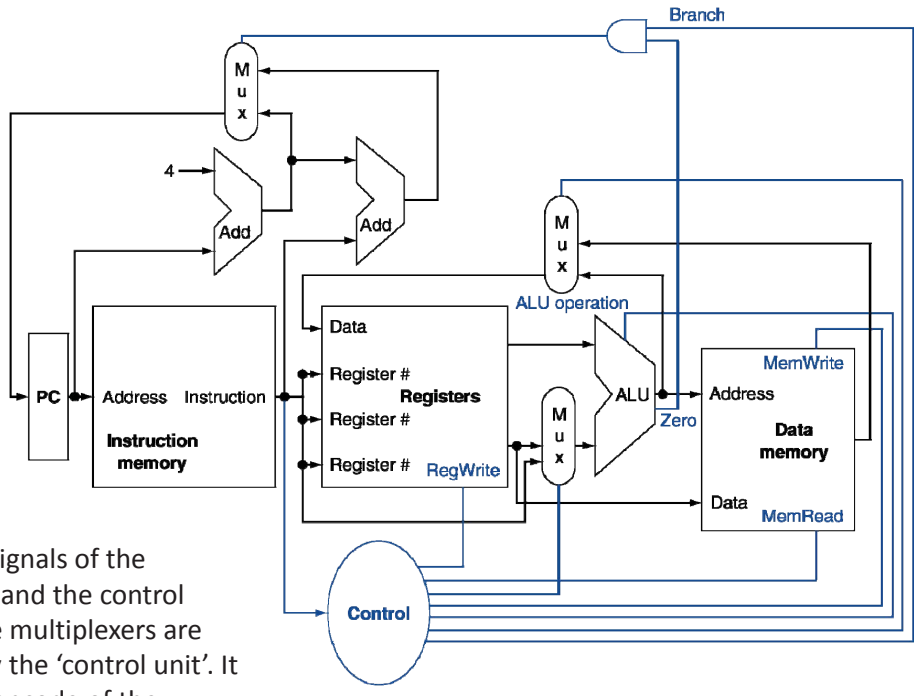
We add 16-bit address offset to PC+4. We branch to this address



We Need to Add Control Signals & Multiplexers



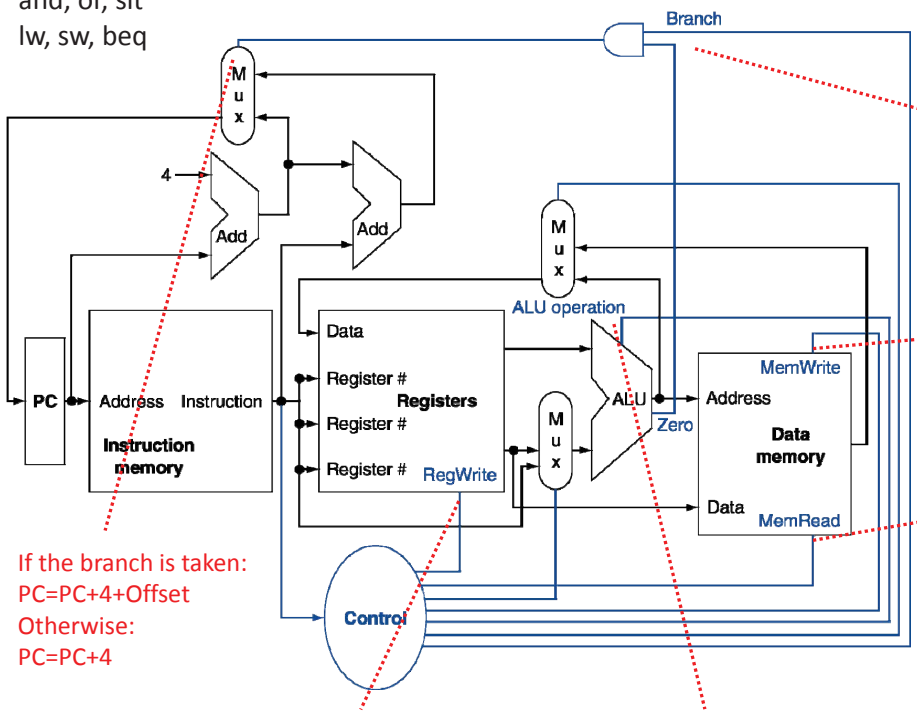
We add the Multiplexers (MUXes) and Control Signals



The control signals of the components and the control signals of the multiplexers are generated by the 'control unit'. It looks at the opcode of the instruction.

Instructions: add, sub, and, or, slt
lw, sw, beq

Datapath with Multiplexers (MUXes) and Control Signals



Branch signal is 1 when it's the beq operation.

zero signal is 1 bit. It is 1 if s0-s1 is 0

Is equal to 1 for sw only

Is equal to 1 for lw only

If the branch is taken:
PC=PC+4+Offset
Otherwise:
PC=PC+4

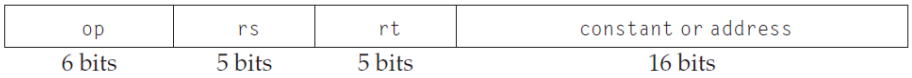
It is 1 for: add, sub, and, or, slt, lw

Asks the ALU to do: add, sub, and, or, slt. Do add for sw, lw. Do sub for beq.

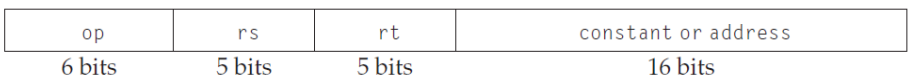
We use MemRead because the memory throws an error if we try to read from an invalid address.

Sign Extender and Shifter

- For 'lw' and 'sw' the memory address is:
- Address = rs + (16-bit field sign extended to 32 bits)

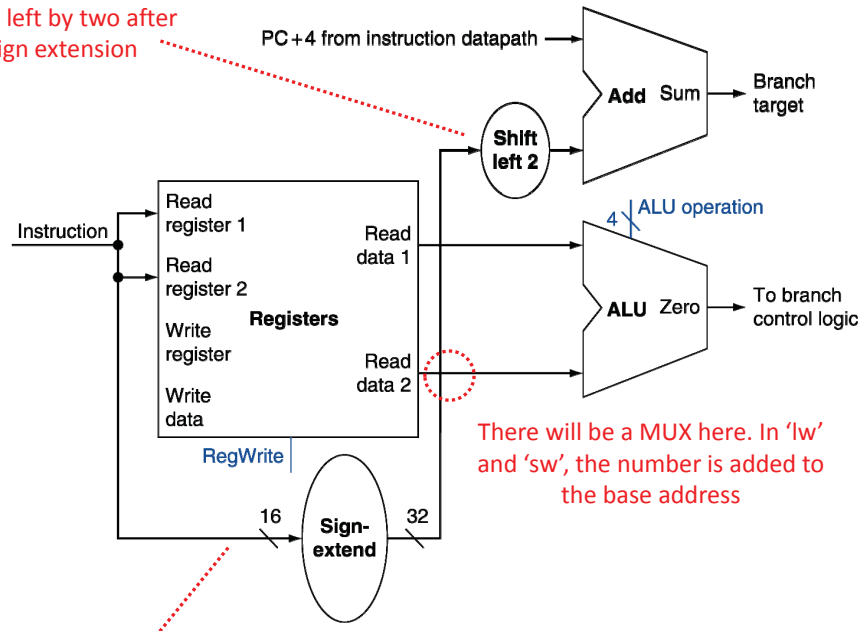


- For 'beq', the branch address is:
- Address = PC + 4 + (16-bit field sign extended to 32 bits then shifted left by 2)



Adding the Sign Extender and the Shifter

In beq, the address offset is shifted left by two after the sign extension



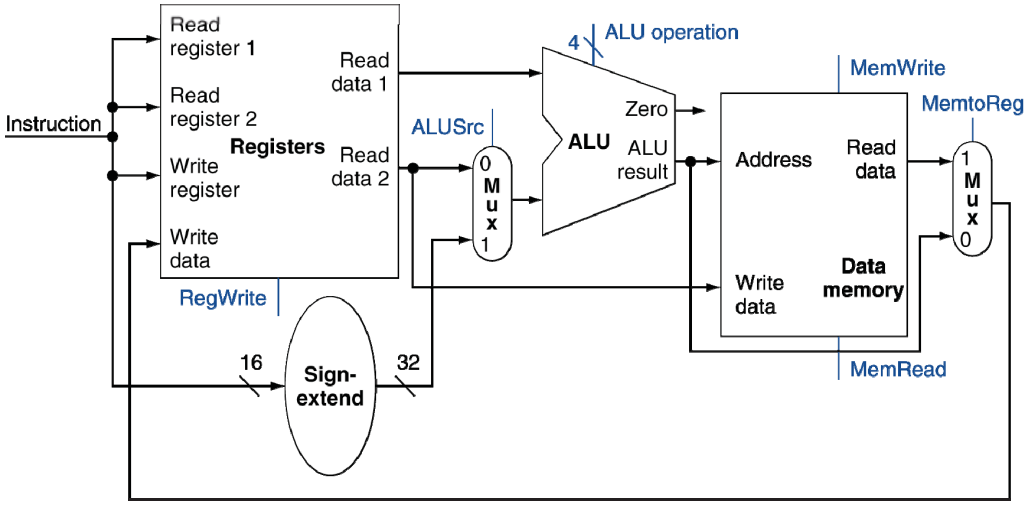
There will be a MUX here. In 'lw' and 'sw', the number is added to the base address

16-bit number from the instruction

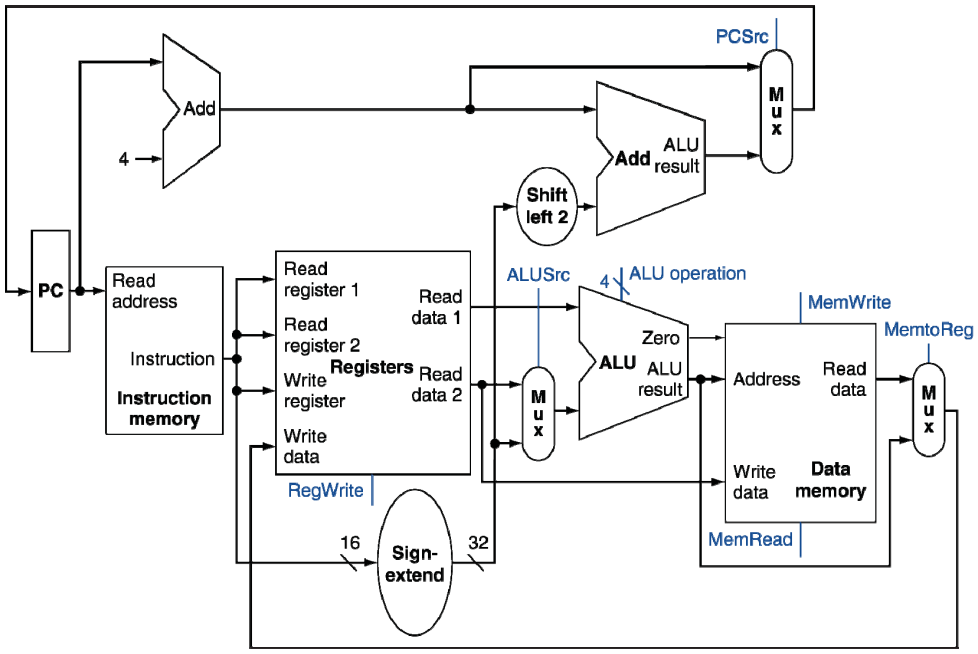
Control Signals

These are the control signals:

RegWrite:	Register Write
ALUSrc:	ALU Source
ALU operation	
MemWrite:	Memory Write
MemRead:	Memory Read
MemtoReg:	Memory to Register



Datapath with the Control Signals



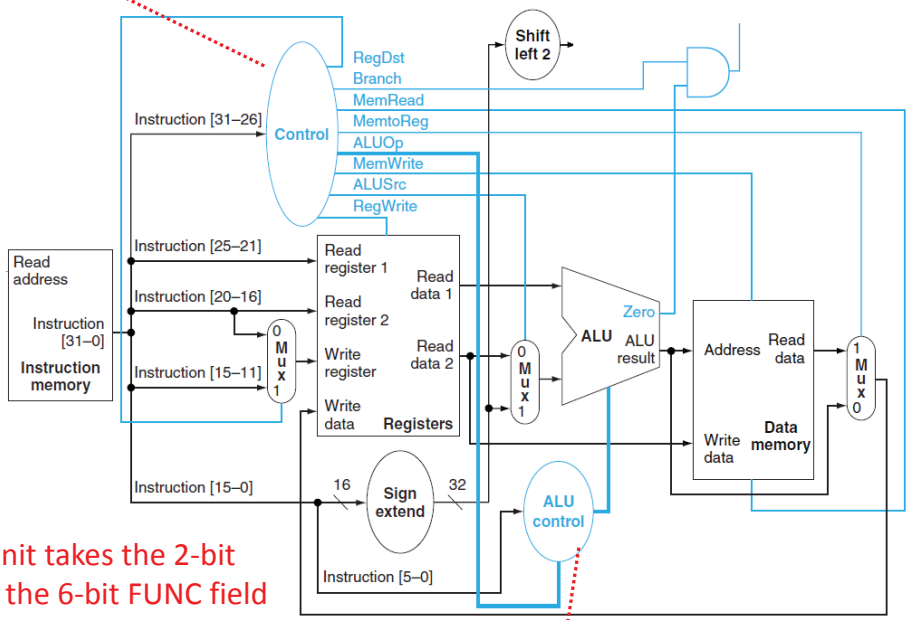
ALU Operations

- The ALU takes a 4-bit 'ALU Operation' field
- This 4-bit signal can designate 16 unique operations in the ALU
- The table below shows values of the 4-bit signal that we'll use and the corresponding ALU operation

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

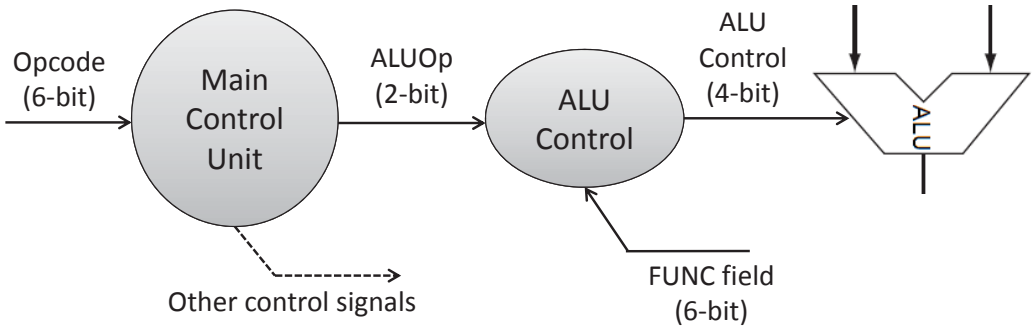
The main control unit takes the opcode and generates a 2-bit signal called 'ALUOp'

Two Levels of Control



The ALU control unit takes the 2-bit ALUOp signal and the 6-bit FUNC field and generates the 4-bit signal to the ALU

ALU Control

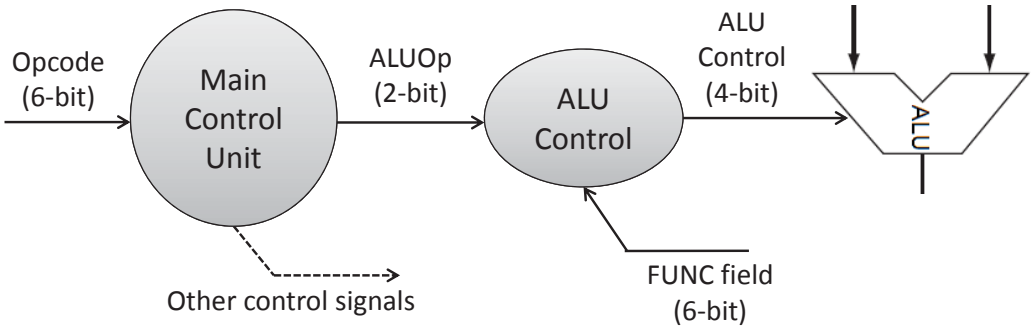


Main Control Unit
- Takes the opcode
- ALUOp=00 is for Addition (used for "lw", "sw")
- ALUOp=01 is for Subtraction (used for "beq")
- ALUOp=10 is for R-type (we need the FUNC field to decide the operation)

ALU Control Unit
- Takes ALUOp and FUNC field
- ALUOp=00, do addition (output code: 0010)
- ALUOp=01, do subtraction (output code: 0110)
- ALUOp=10, look at FUNC field to decide

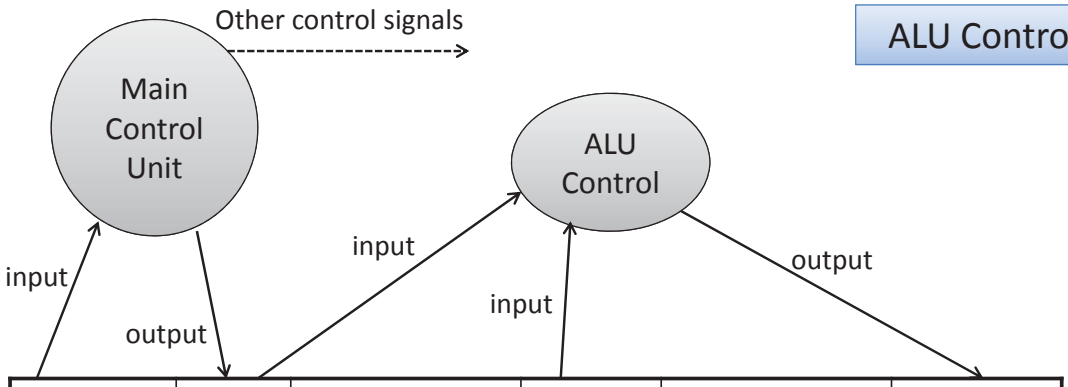
Our instructions: add, sub, and, or, slt, lw, sw, beq, j

Benefit of Two Levels of Control



- If we add a new R-type instruction to the datapath
- No change is needed in the main control
- The main control unit will give out '10' (this corresponds for R-type)
- In this case the ALU control looks at the FUNC field
- The new R-type instruction is assigned a unique FUNC value

ALU Control



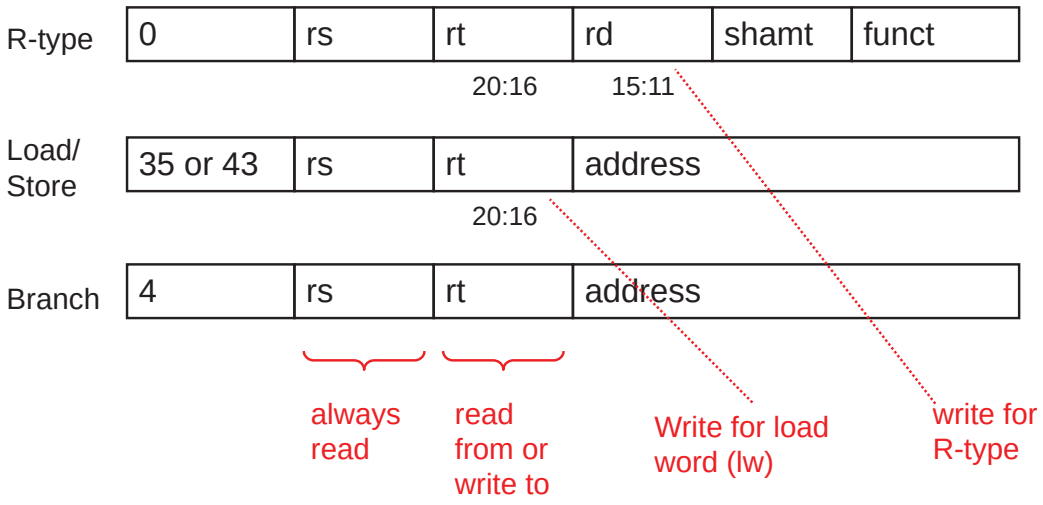
opcode	ALUOp	Instruction	funct	ALU function	ALU control
100011 (35)	00	load word	XXXXXX	add	0010
101011 (43)	00	store word	XXXXXX	add	0010
000100 (4)	01	branch equal	XXXXXX	subtract	0110
000000 (0)	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Main Control Unit

Truth table

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

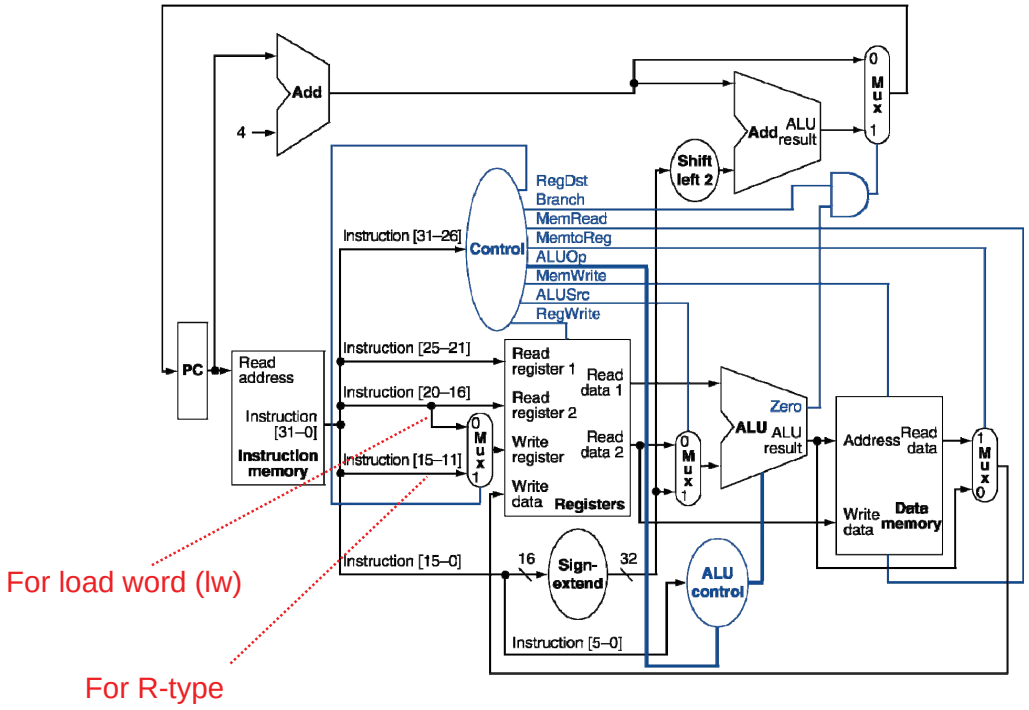
To which register should we write data?



- In R-type, we read 'rs' and 'rt' and write to 'rd'
- In load word (lw), we read 'rs' and write to 'rt'
- In store word (sw), we read 'rs' and 'rt'

New control signals
RegDst: Register Destination
Branch: is equal to 1 only for 'beq'

Datapath with Control

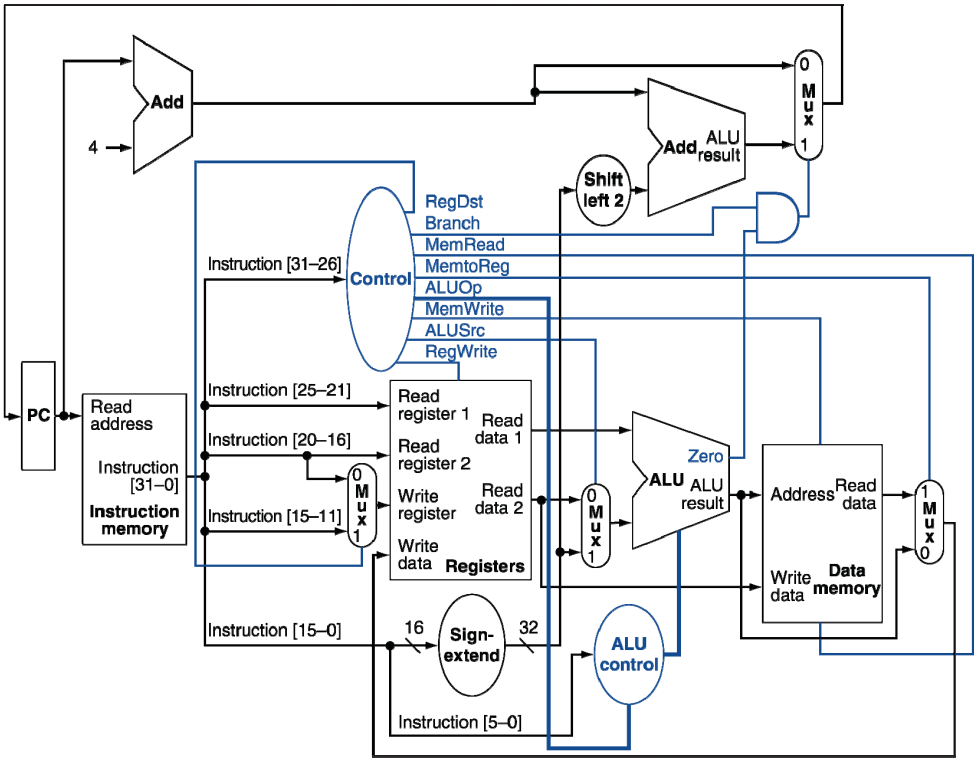


For load word (lw)

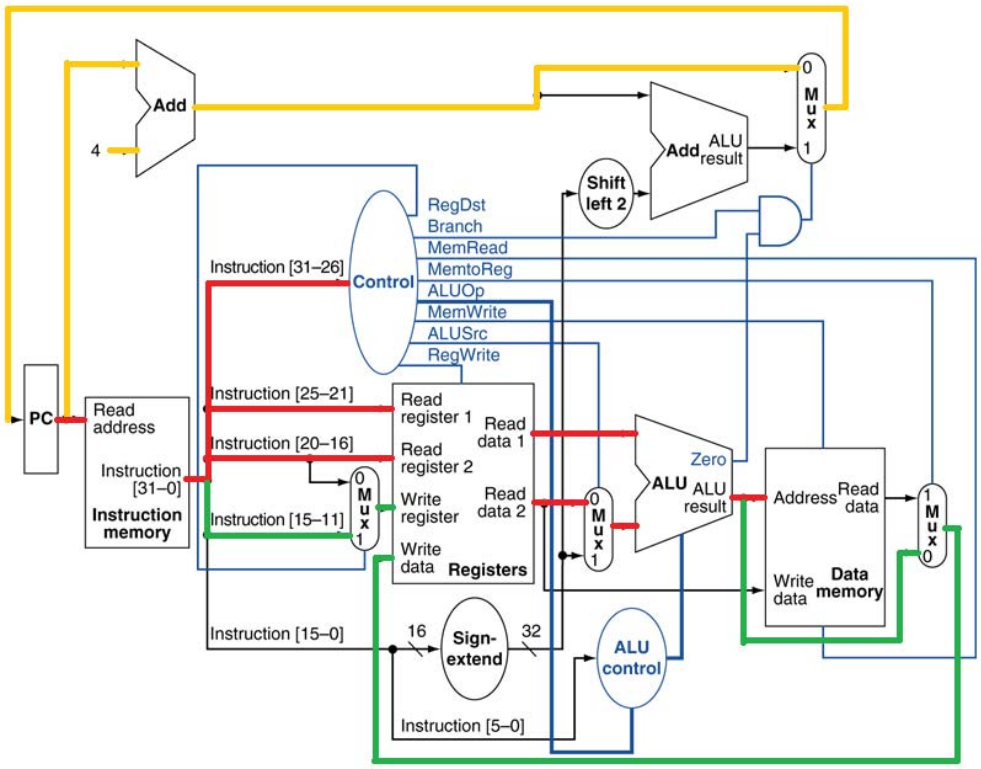
For R-type

For each instruction, identify the path taken

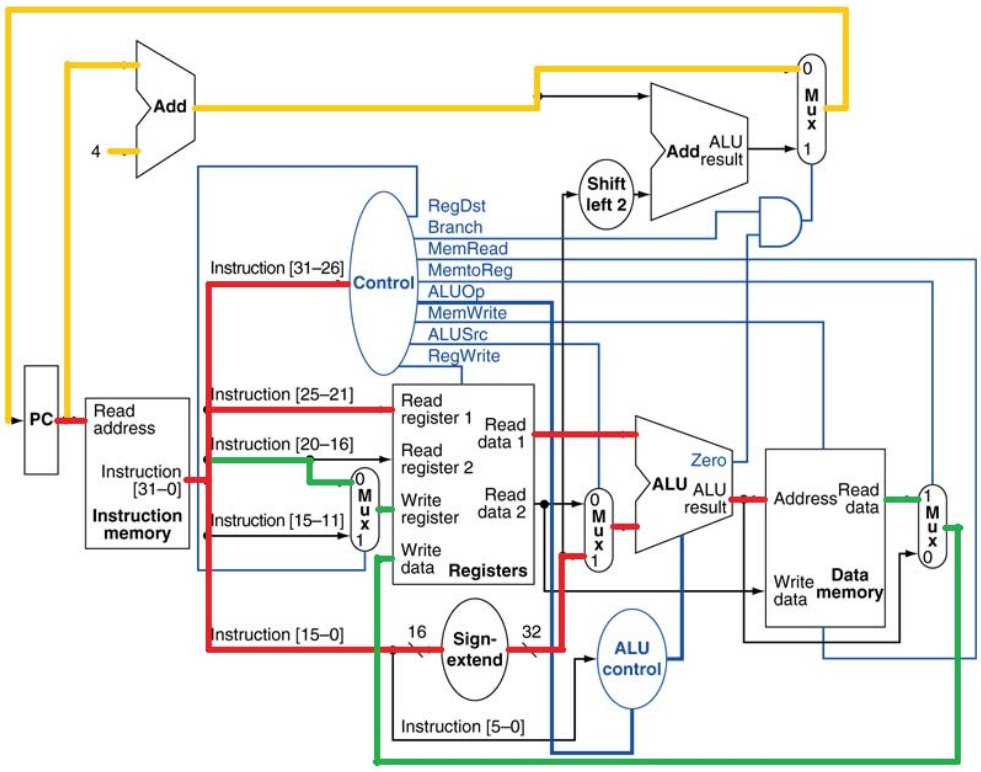
Datapath with Control



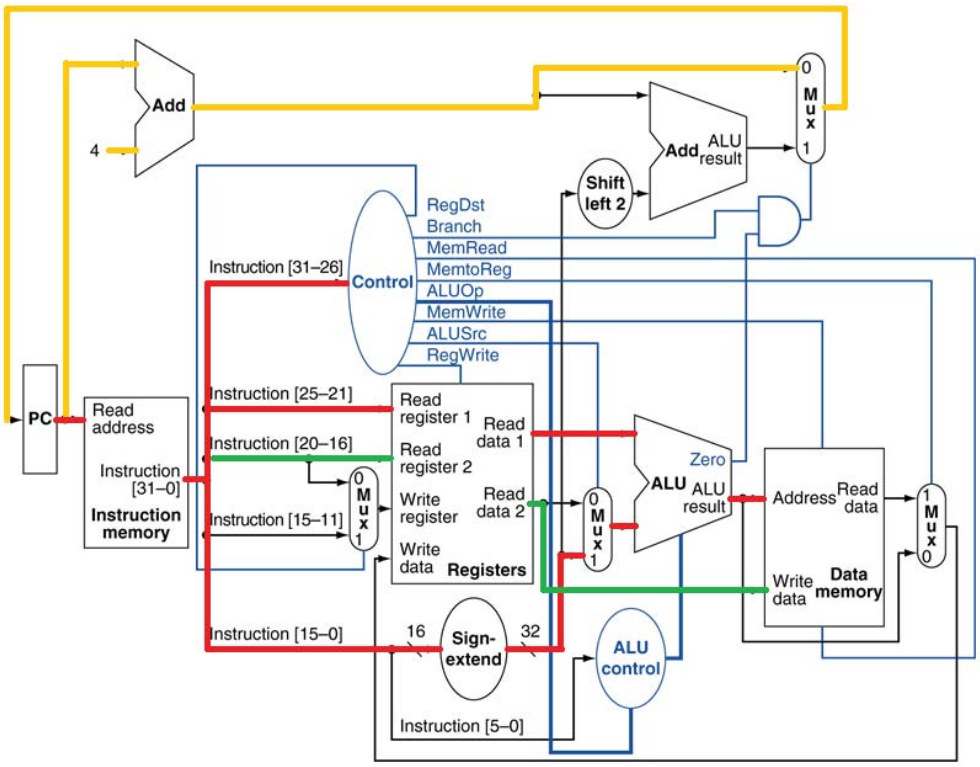
Path for R-Type



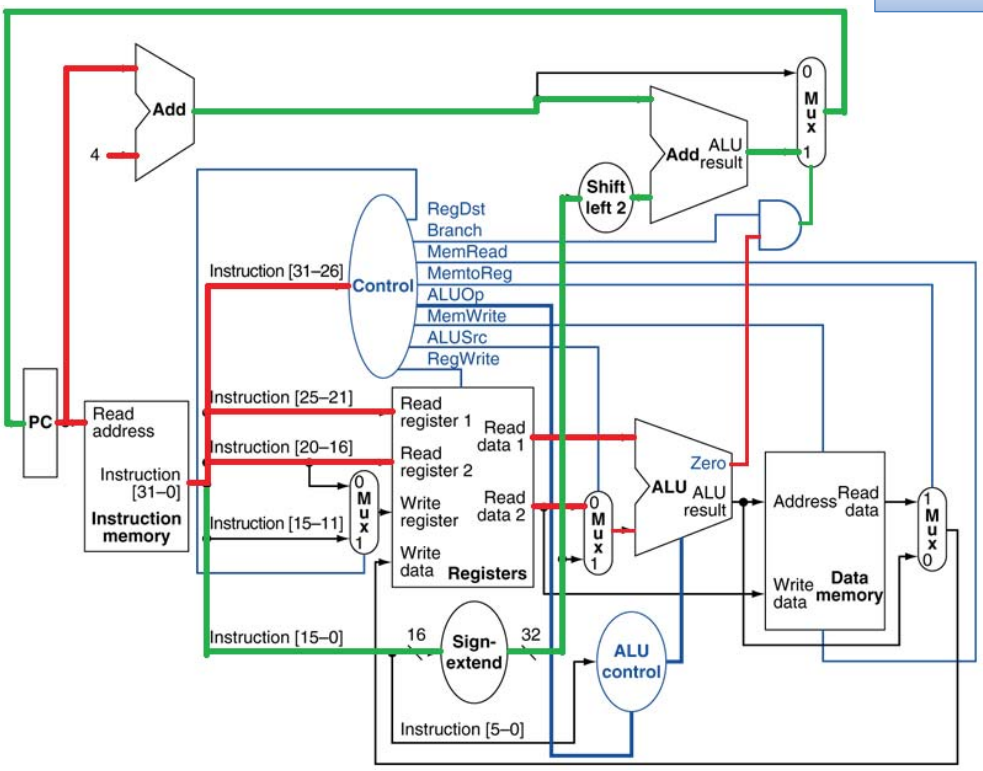
Path for Load Word



Path for Store Word

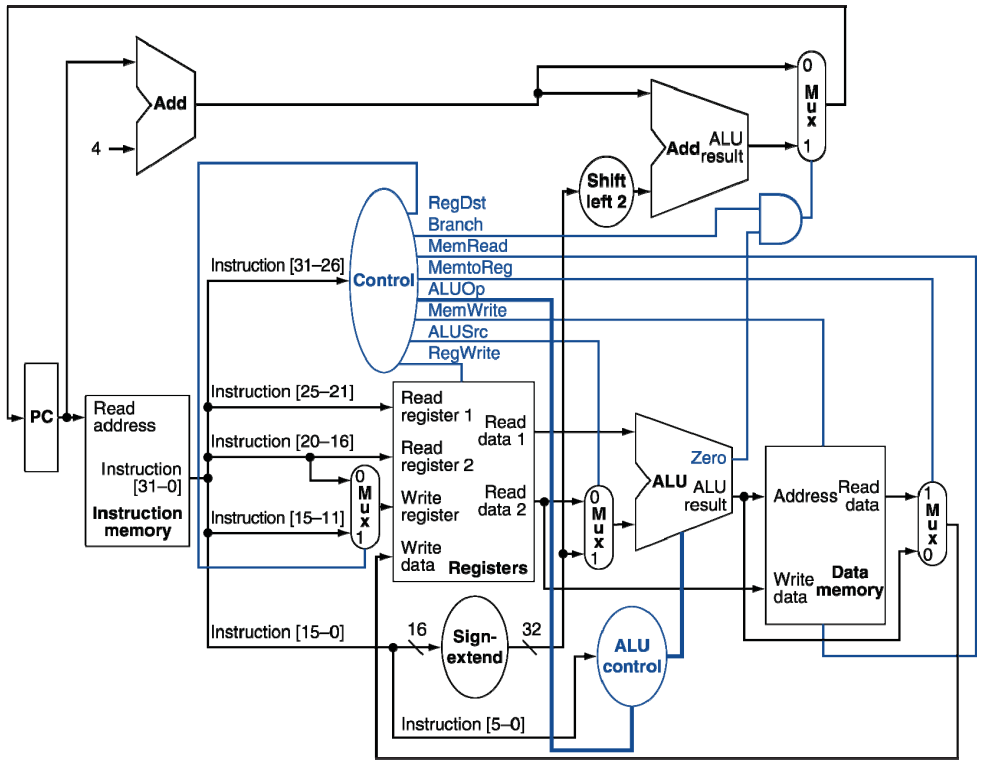


Branch (beq)

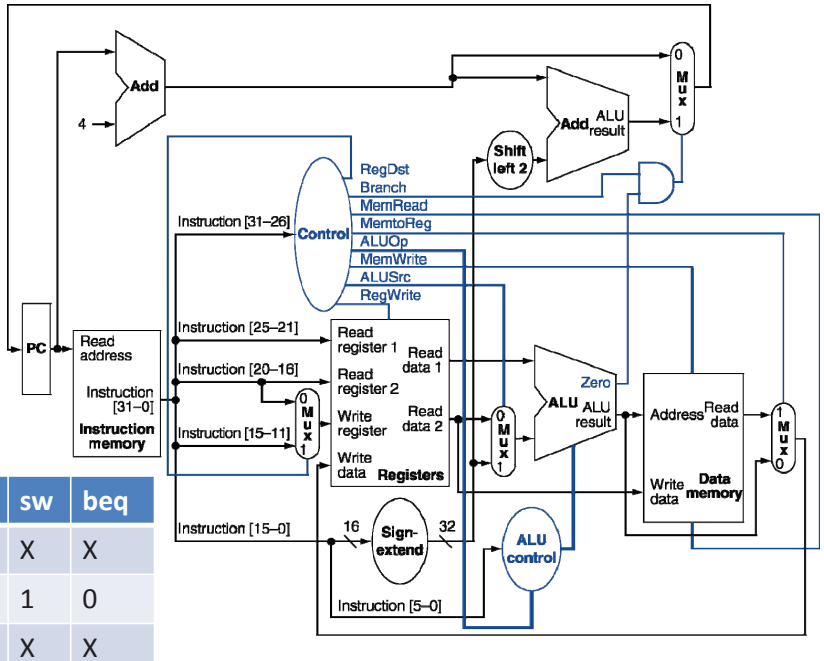


What is the value of the control signals for every instruction?

Datapath with Control



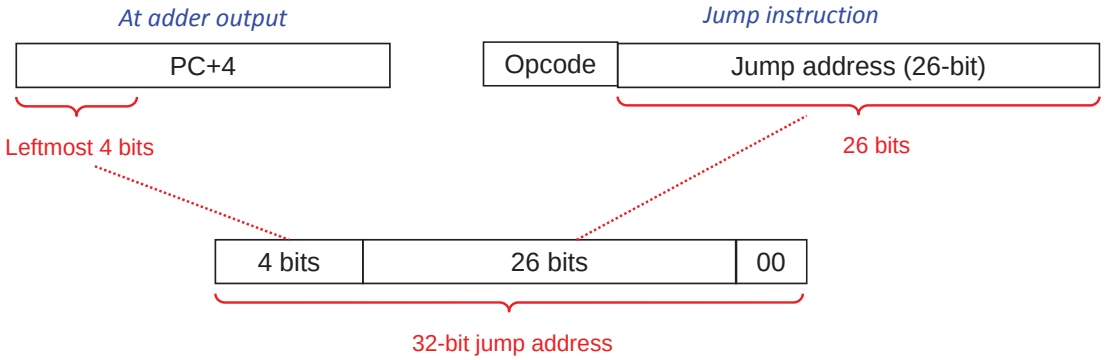
These are the values of the control for every instruction



	R-type	lw	sw	beq
RegDst	1	0	X	X
ALUSrc	0	1	1	0
MemtoReg	0	1	X	X
RegWrite	1	1	0	0
MemRead	0	1	0	0
MemWrite	0	0	1	0
Branch	0	0	0	1

Supporting the Jump instruction

- The datapath we have doesn't support the 'j' instruction yet
- The jump address is computed as shown below
- We'll add this mechanism in the datapath

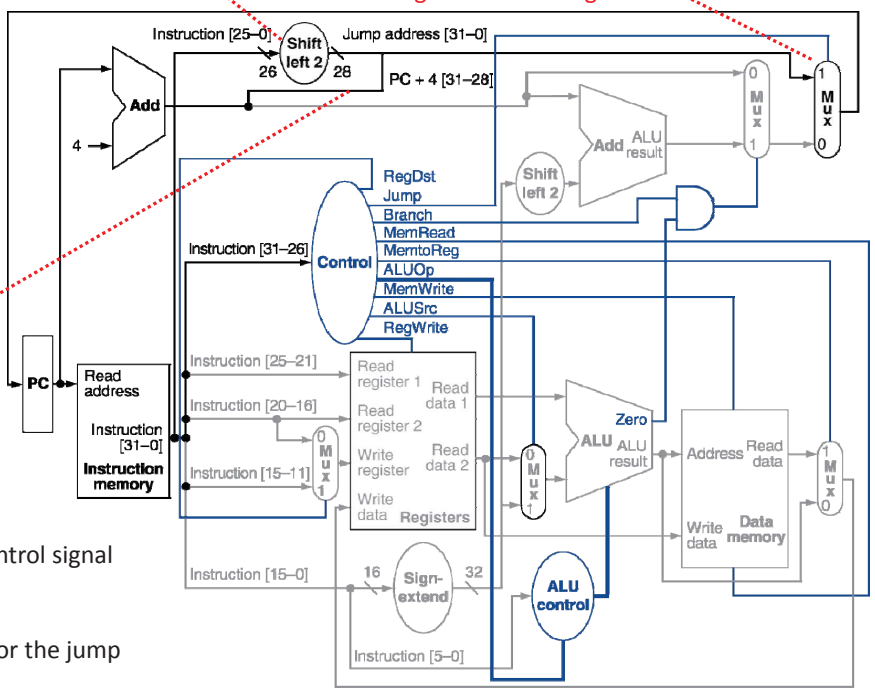


Supporting the jump (j) instruction

The 26-bit field from the jump instruction is shifted left by 2 bits

We add a new multiplexer. We could have made the other mux a 4-to-1, but we would have to change the control signals.

Take leftmost 4 bits from PC+4 at adder output

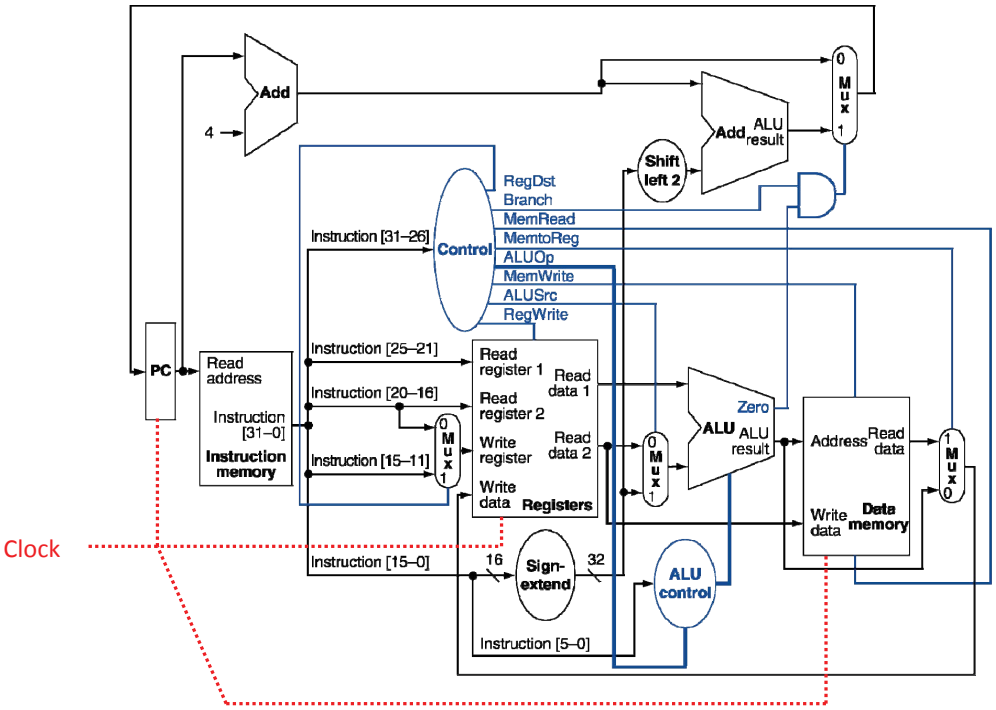


We add a Jump control signal

Jump:
is equal to 1 only for the jump instruction

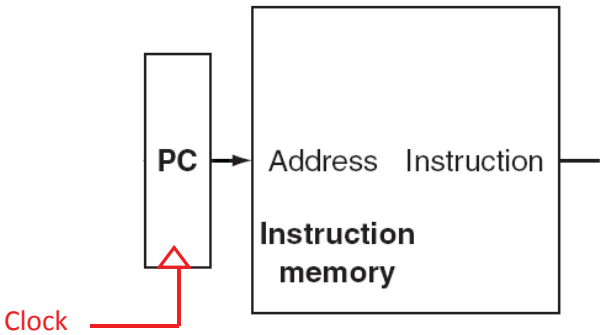
Clock in the Datapath

The clock signal is connected to the PC register, the register file and the data memory.



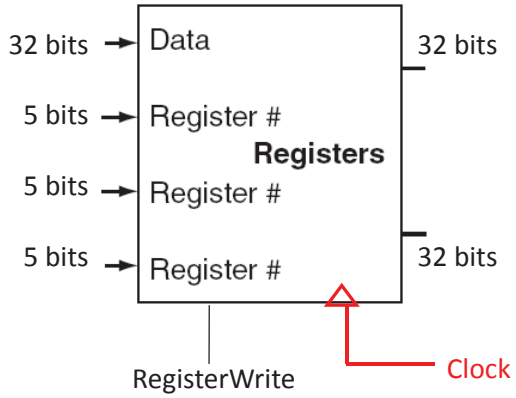
PC Register

- In our implementation, we'll use the positive edge of the clock
- The PC register is updated at the positive edge of the clock
- When the PC is updated a new instruction is read from the memory



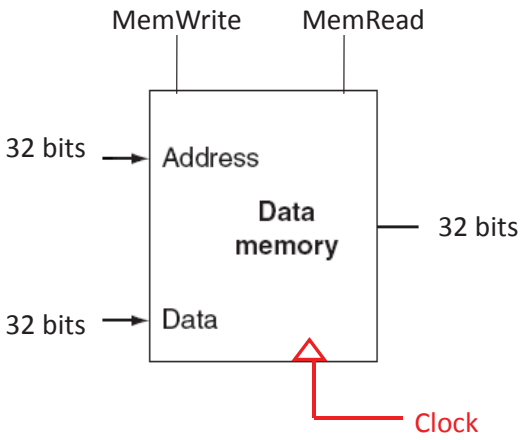
Register File

- The ‘read’ operation is not synchronized with the clock
  - When we set the two 5-bit read addresses, the two register values appear at the output immediately (not sync’d with the clock)
- The ‘write’ operation is synchronized with the clock
  - We set the 5-bit write address and the 32-bit data
  - The data is written in the register at the positive edge of the clock (on condition that RegWrite=1)



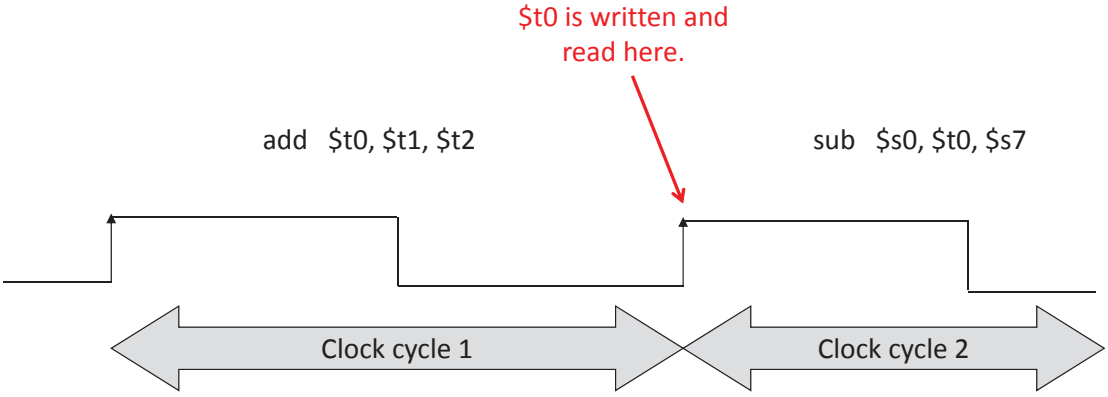
Data Memory

- The ‘read’ operation is not synchronized with the clock
  - Once we set the address, the data shows at the output, after the memory’s delay (provided MemRead=1)
- The ‘write’ operation is synchronized with the clock
  - We set the 32-bit address and the data, the data is written in the memory at the positive edge of the clock (provided MemWrite=1)



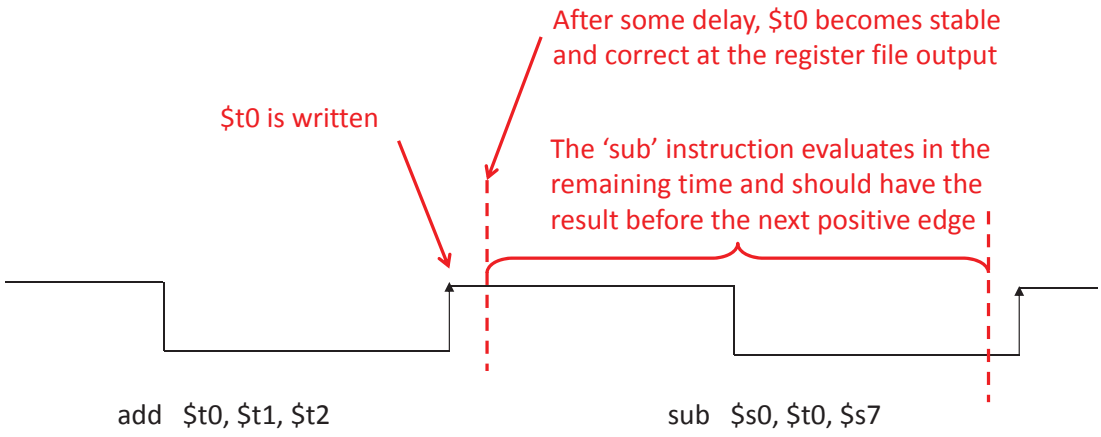
Writing and Reading the Same Register

- The 'add' instruction writes to $t0 at the positive edge between cycle 1 and cycle 2
- The 'sub' instruction starts reading $t0 at the same positive edge
- Does this lead to a racing condition?
- Usually in logic circuit, we shouldn't read and write a data at the same time



Writing and Reading the Same Register

- Register $t0 is written at the end of clock cycle 1
- When clock cycle 2 starts, the value of $t0 takes some time to become stable and correct (but after some delay, it will be correct)
- From that point, there should be enough time for the 'sub' instruction to evaluate successfully before the next positive edge arrives



Clock Cycle Time

- At any point in time, there is one instruction in the CPU
- The instruction will finish execution in 1 clock cycle
- The instructions we're supporting are:
  - add, sub, and, or, slt, lw, sw, beq, j

- The clock cycle is equal to the duration of the longest instruction
- The "load word" (lw) takes the longest time:
  - Read instruction
  - Read register
  - Add 16-bit address offset to register
  - Read from memory
  - Write back to memory

Clock Cycle Time

- What is the time required by every instruction type?
- The table gives the time of every component and the instruction's total time

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100		50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100			350 ps
Jump	200					200 ps

- The clock cycle time is equal to the maximum time among all the instructions
- In this example, it's 600 ps ➔ F = 1.66 GHz

1 ps (picosecond) = 10^{-12} second

Clock Cycle Time

- Using the table below, the clock cycle duration is 600 ps
- How much time are we actually wasting?

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100		50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100			350 ps
Jump	200					200 ps

- This depends on the instruction mix
- If we're doing 'load' instructions only, we're not wasting any time since it needs the 600 ps
- However, if we're doing and R-type, we're wasting 200 ps since the instruction needs 400ps but we're giving 600 ps

Clock Cycle Time

Load:	25%
Store:	10%
R-type:	45%
Branch:	15%
Jump:	5%

- Let's consider this instruction mix
- Hypothetically, if we're able to give each instruction only the duration of time it needs, the average time of an instruction is:

$$\text{Time}_{\text{needed}} = (0.25*600)+(0.1*550)+(0.45*400)+(0.15*350)+(0.05*200)$$
$$= 447.5 \text{ ps}$$

- However, with the single-cycle implementation that we have, each instruction takes: 600 ps
- Therefore, we can potentially speed the CPU by a factor of:

$$600 / 447.5 = 1.34 \text{ times}$$

Clock Cycle Time

- In the single-cycle implementation example, we gave 600 ps to each instruction
- A typical practice in computer design is to have a constant clock duration
- That's why we don't vary the clock cycle duration for each instruction
  - This will make the design complex since we have to decode the instruction before allowing it in the datapath and determine its corresponding clock duration
- The downside of using the same clock cycle duration for all the instructions is that we might end up wasting some CPU time
- The next datapath designs that we'll look at will take care of this problem

Single-Cycle Datapath Exercises

Exercise 1

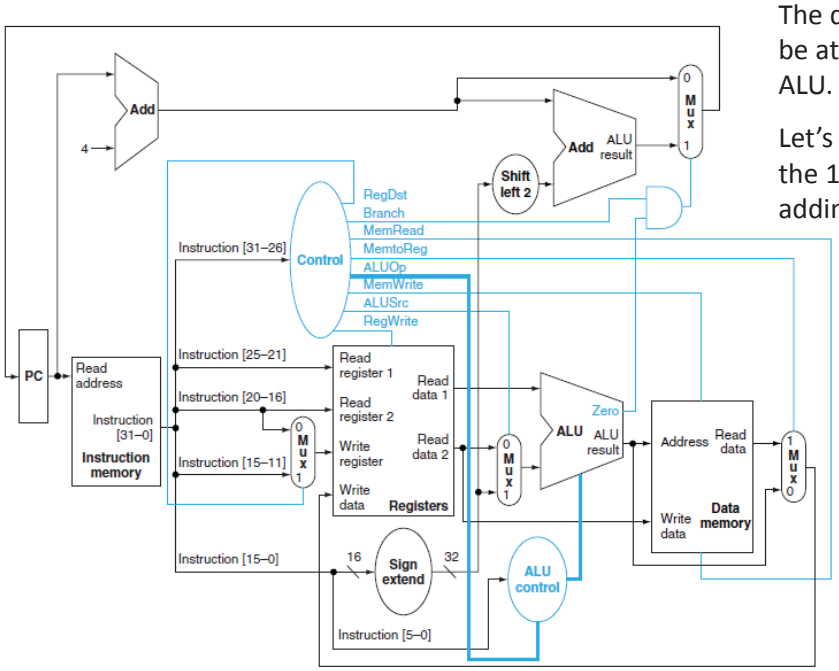
- Modify the single-cycle datapath to add the Shift Left Logical (sll) instruction

- 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 (cont'd)

- Where do we want to do the shifting?
- We could add a shifter circuit… but it's better to use the ALU

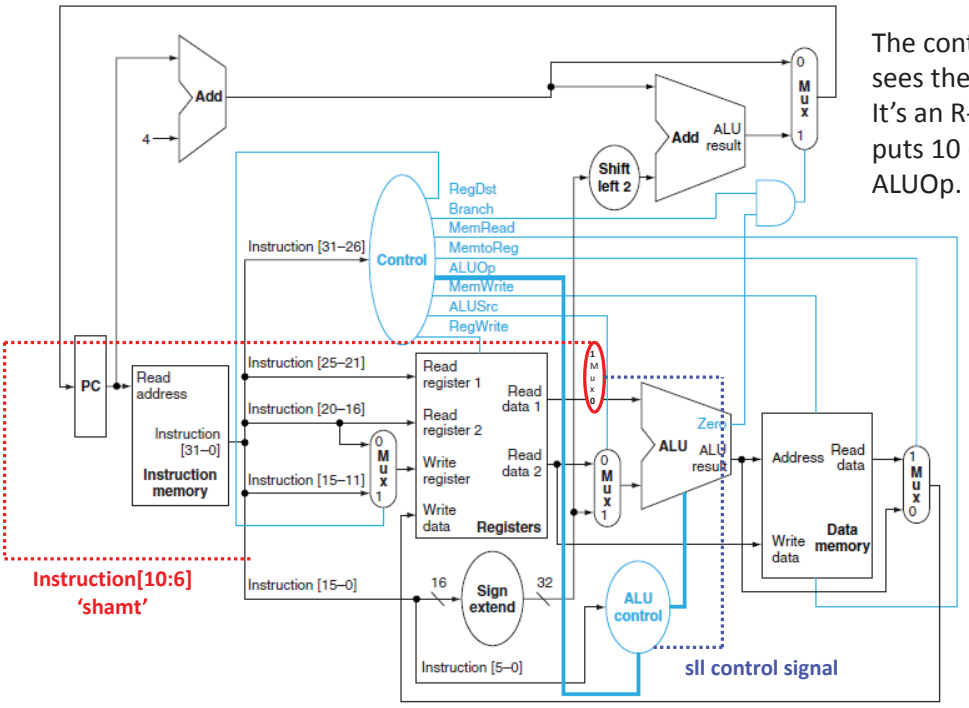


The data to shift, in 'rt' will be at the 2nd input of the ALU.

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

Control signals

- (RegDst=1), (Branch=0), (MemRead=0), (MemtoReg=0), (ALUOp=10 for R-type), (MemWrite=0), (ALUSrc=0 for 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 get ALUOp=10, looks at FUNCT fiels and sees 0. It recognizes the sll instruction. It sets the 4-bit field to the code of sll.

Exercise (cont'd)

- 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

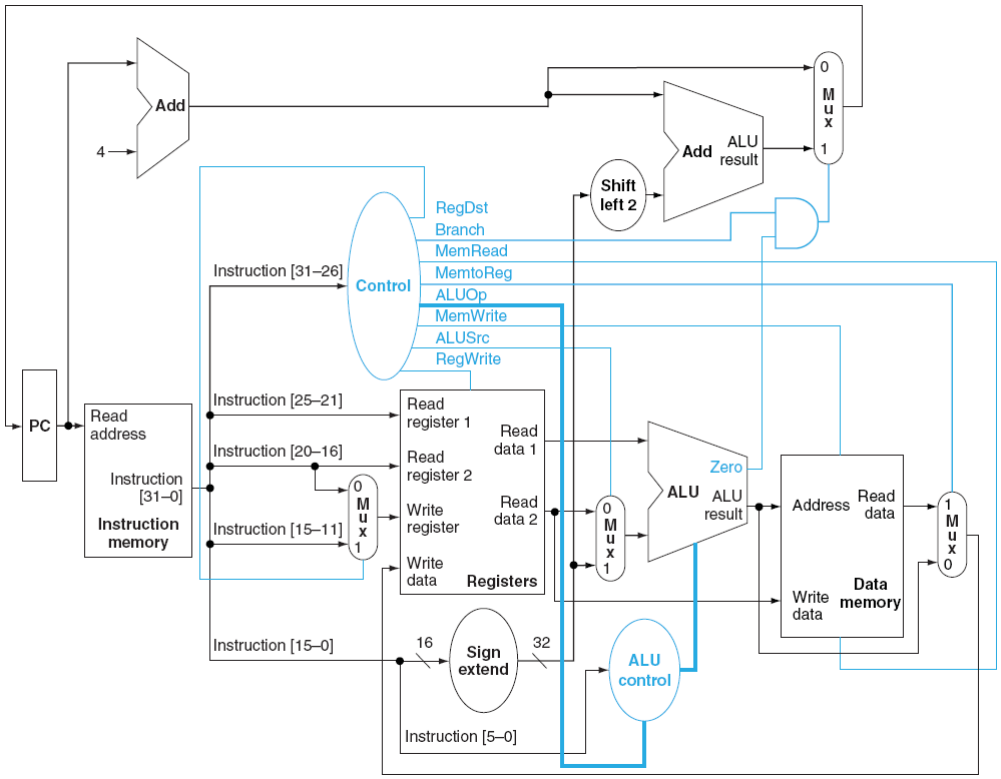
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR
????	**Shift Left**

Exercise 2

- 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

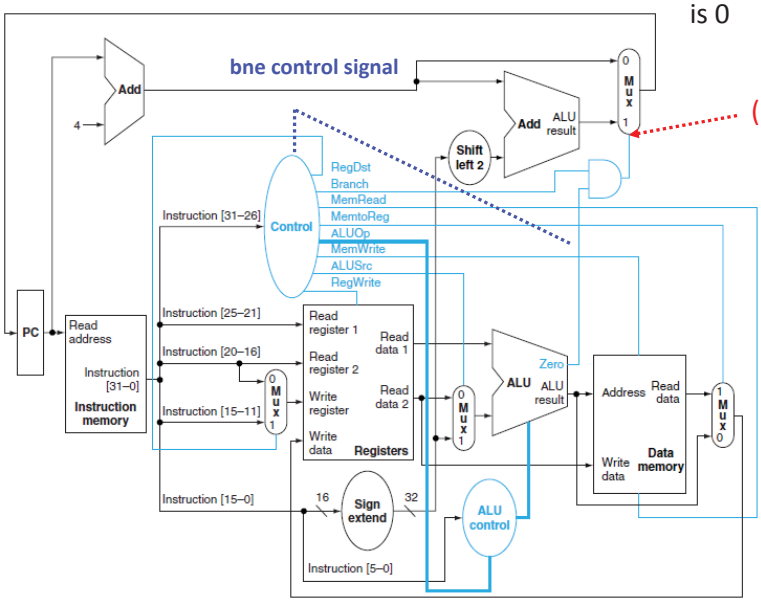
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Draw the changes



* 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 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



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. 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.

In previous examples, we showed the usefulness of "jal". Before branching to a procedure, we use "jal". The return address is saved in $ra. At the end of the procedure, we use "jr $ra" (Jump-to-register) so that we can get back to the main code.

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
```

What does “jal” do?

The MIPS instruction format of “jal” is shown below.

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

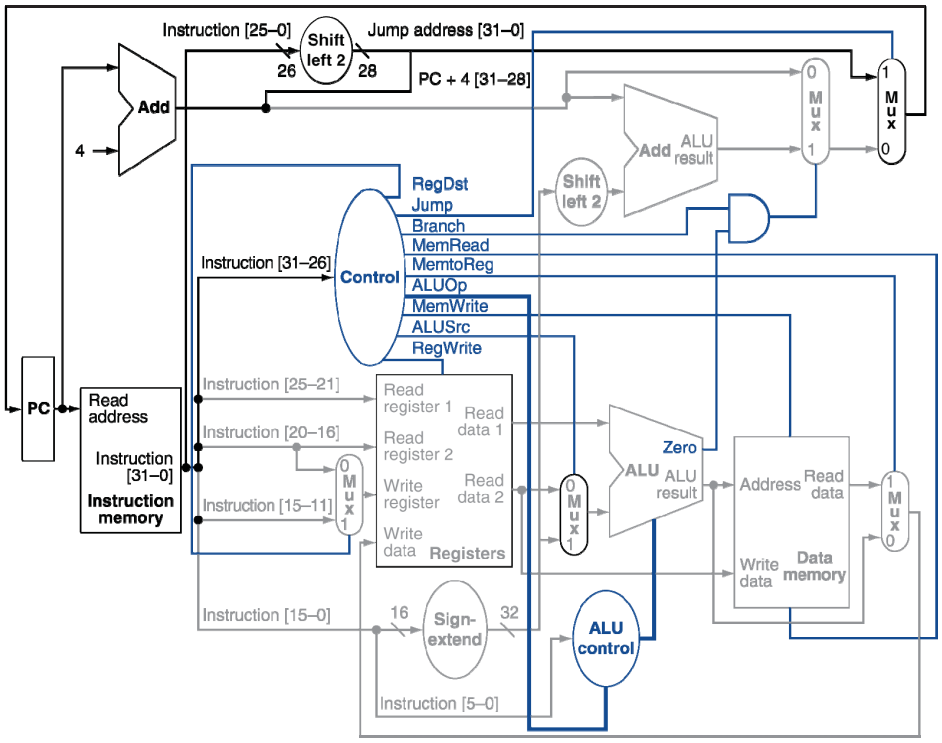
On top of that, it saves PC+4 in the return address register ($ra).

`jal target`

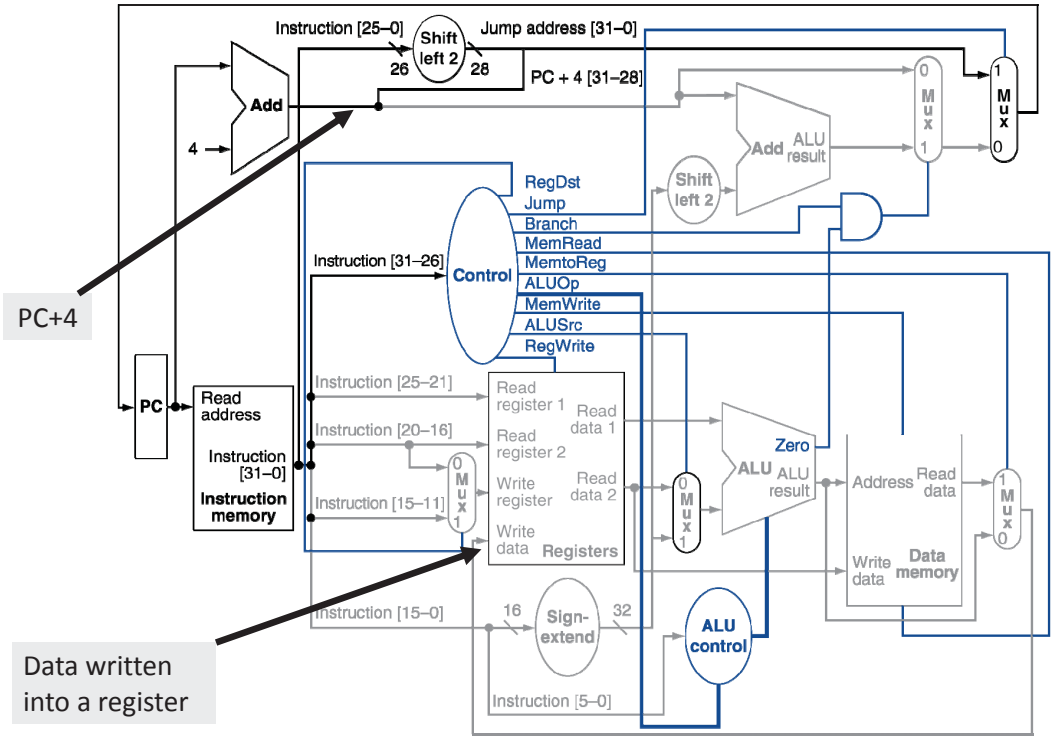
3	target
6	26

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


Draw the changes

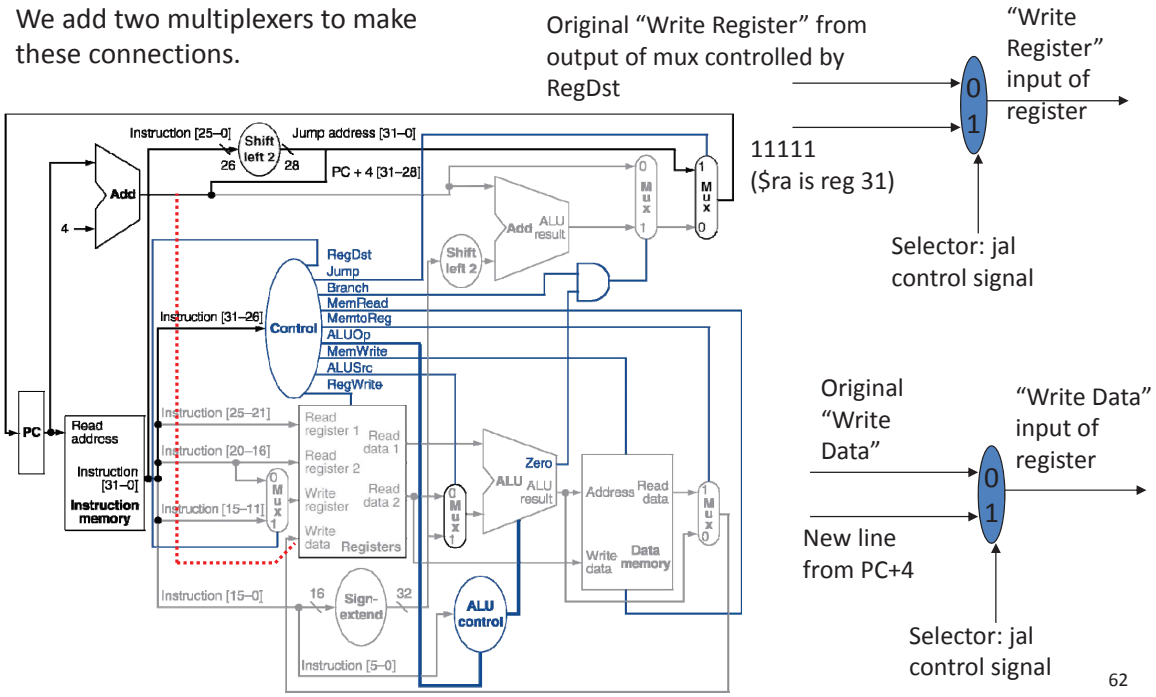


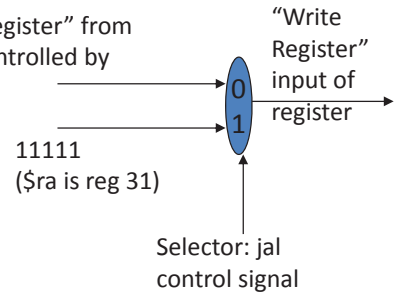
We need the PC+4 value to be saved inside a register. By looking at the datapath below, we see that there's no way for the PC+4 (output of adder) to be sent to a register. We need to modify the datapath.



We add a new line that takes the PC+4 and sends it to the "Write Data" input of the register file. We need to add a multiplexer. We also need to make sure the "Write Register" field is set to the address of $ra. Register $ra is register number 31.

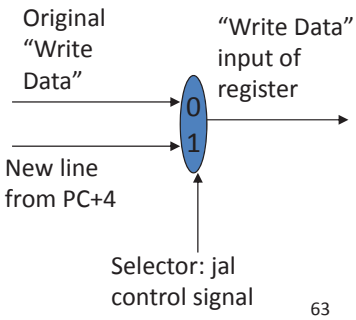
We add two multiplexers to make these connections.





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

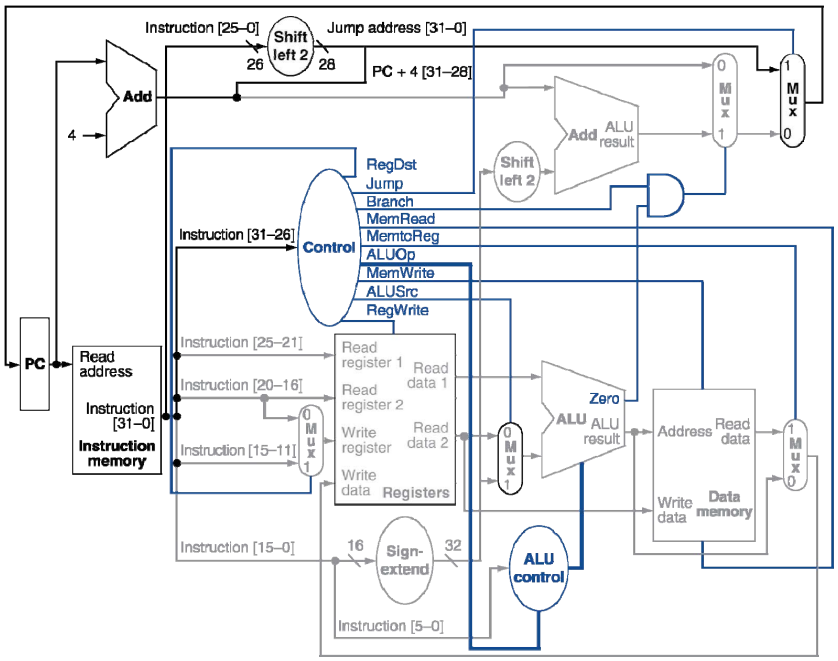
The MIPS format of the "jal" instruction is shown below. It uses opcode=3, so the control unit recognizes it through this opcode.



The main control unit should produce these control signal values when the "jal" instruction is in the datapath.

jal: 1

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



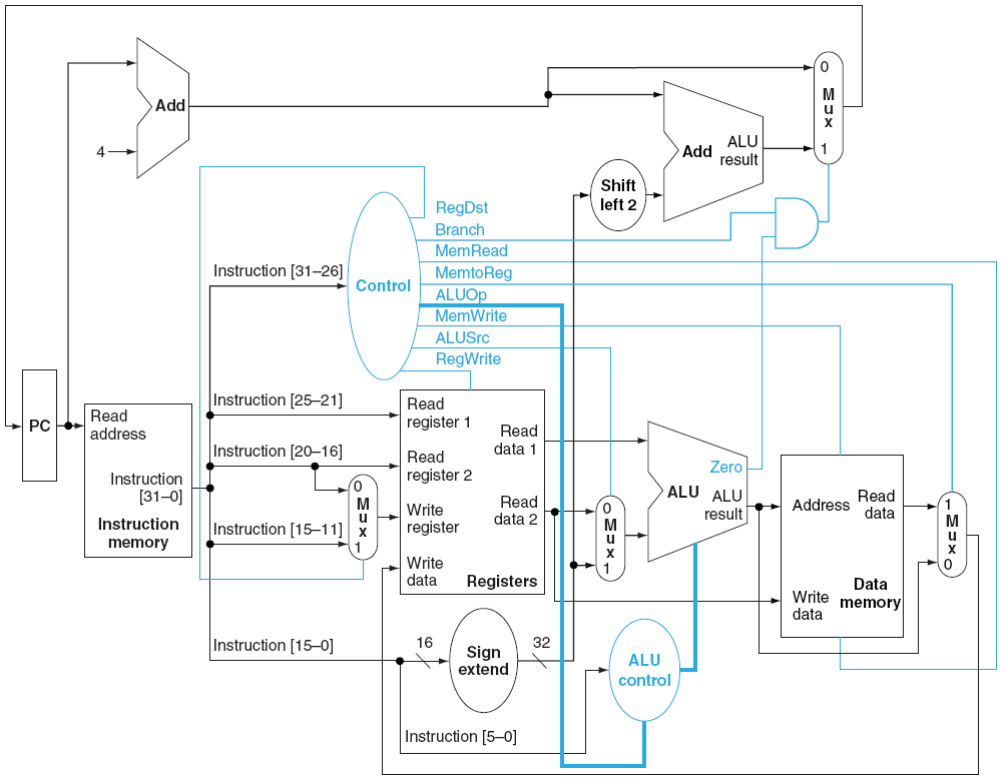
Exercise 4. Modify the single-cycle datapath so it implements the "jump-to-register" (jr) instruction. The "jr" jumps to the 32-bit address that's stored inside a register. Typically, it's used as "jr $ra" to jump to the return address after finishing a procedure.

- 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

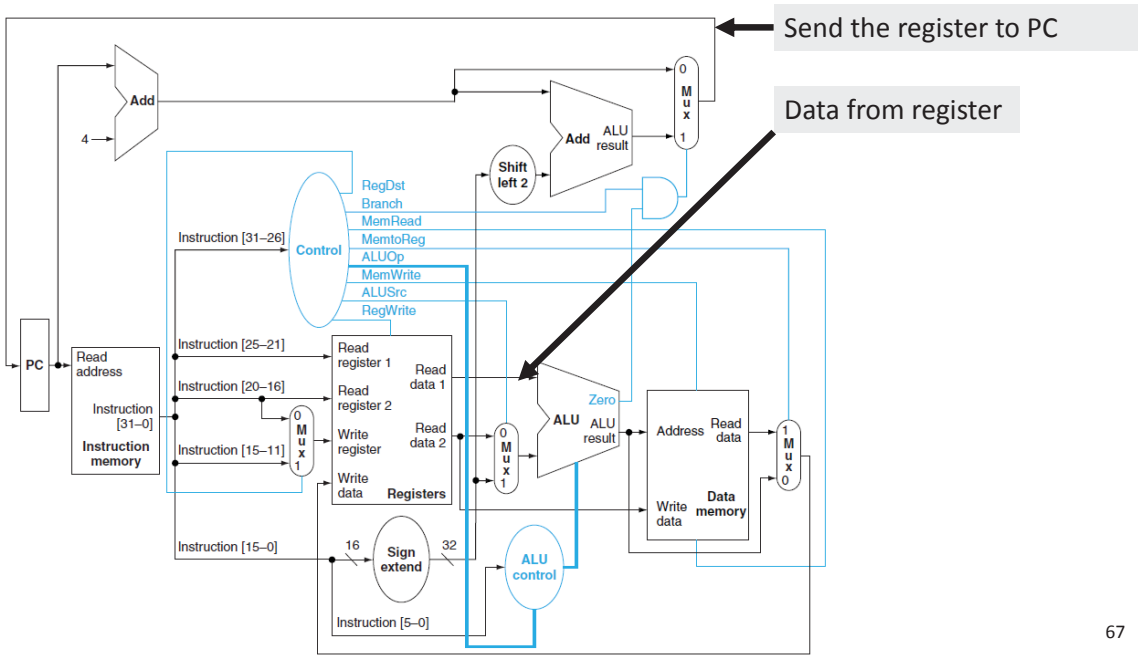
jr rs

0x0	rs	0	0x8
6	5	16	5

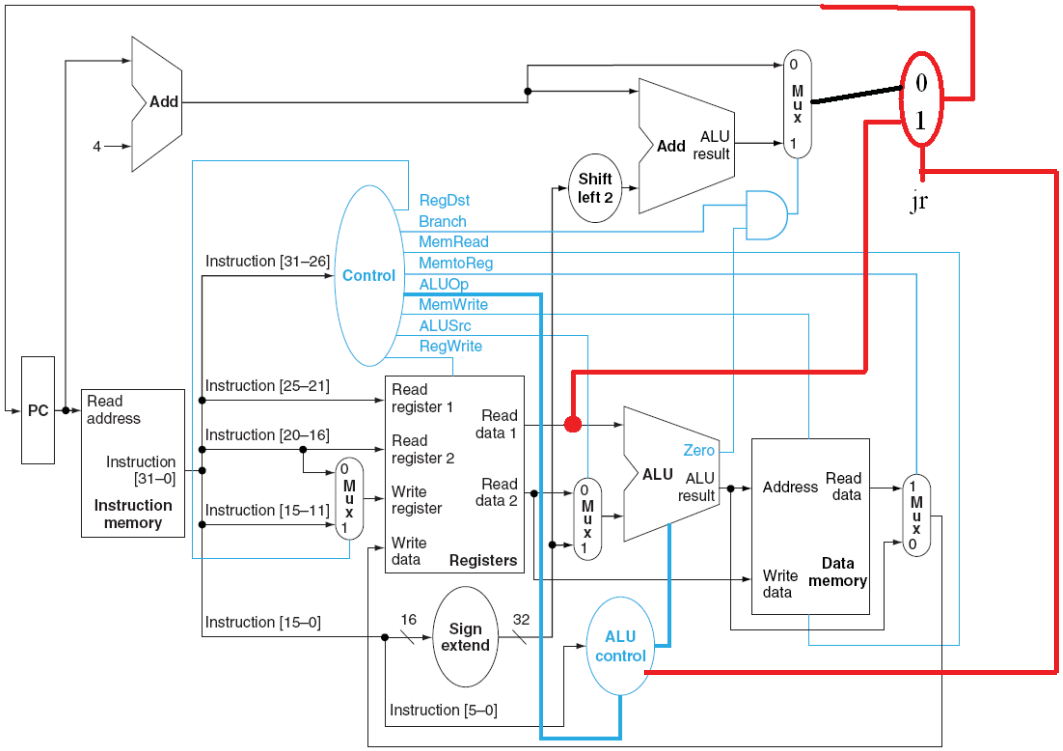
Draw the changes



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.



This is the new line. We also added a multiplexer that's controlled by a control signal 'jr'. This control signal should identify the 'jr' instructin. It's generated by the ALU control unit.



The main control unit sees Opcode=0 when there's a 'jr' instruction. The ALU cannot distinguish between the 'jr' instruction and other R-type instructions.

Accordingly, the main control unit sets the control signals as the following

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'

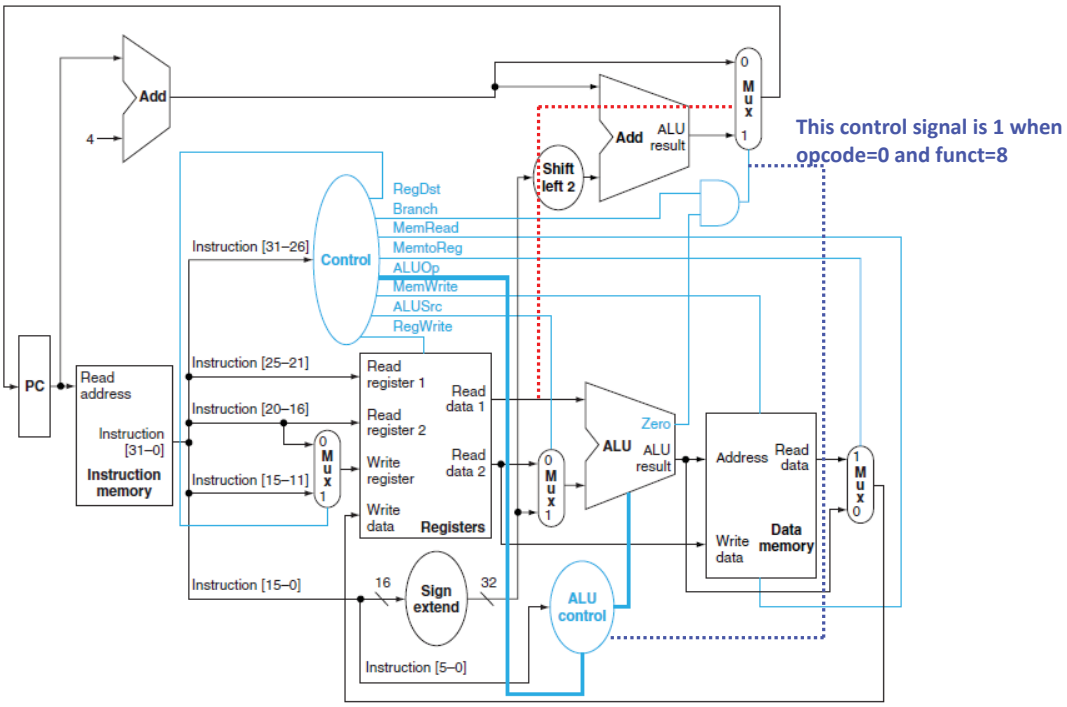
ALUSrc=0 # That's ok for 'jr'

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

- This is another way to do the 'jr' instruction
- Instead of adding a new multiplexer, we upgrade the existing 2-to-1 mux to a 4-to-1 mux



Question 5. 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

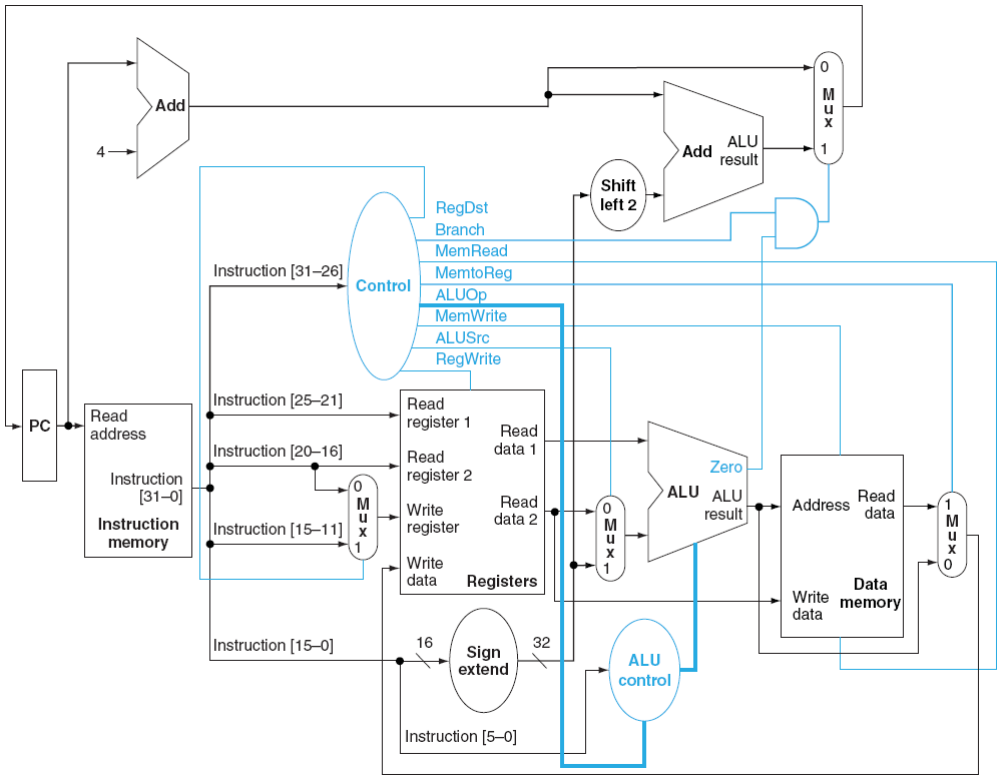
Example of use:

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

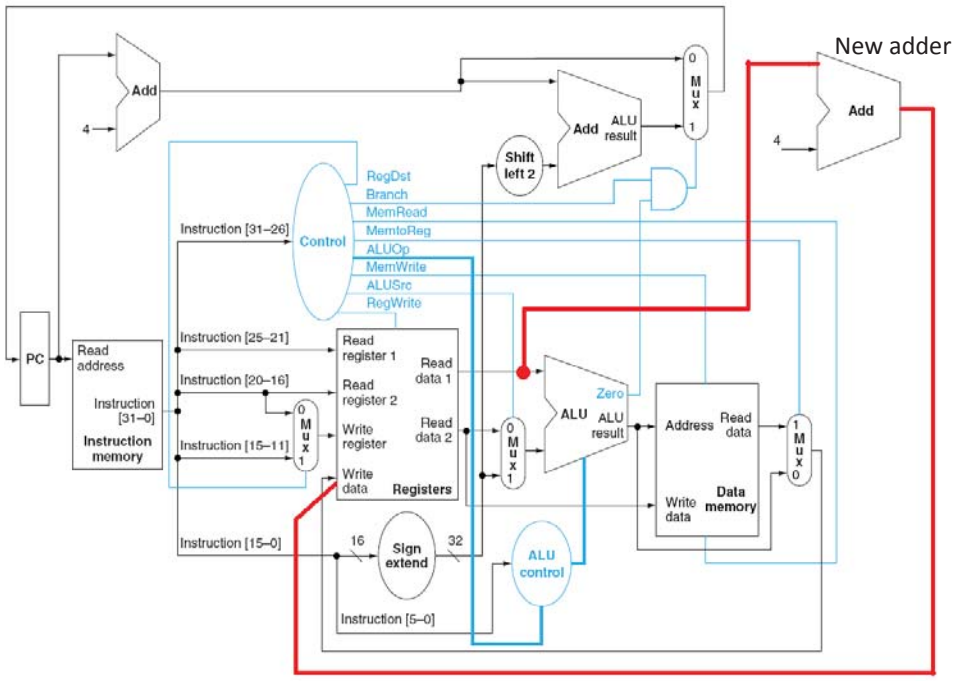
To store $t0, $t1, $t2 in an array with the base in $s0.

```
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)
```

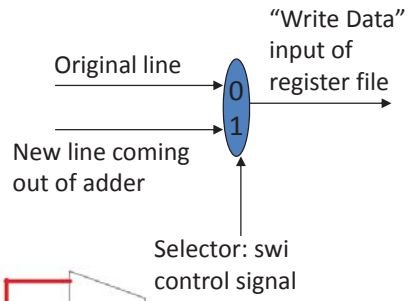

Draw the changes

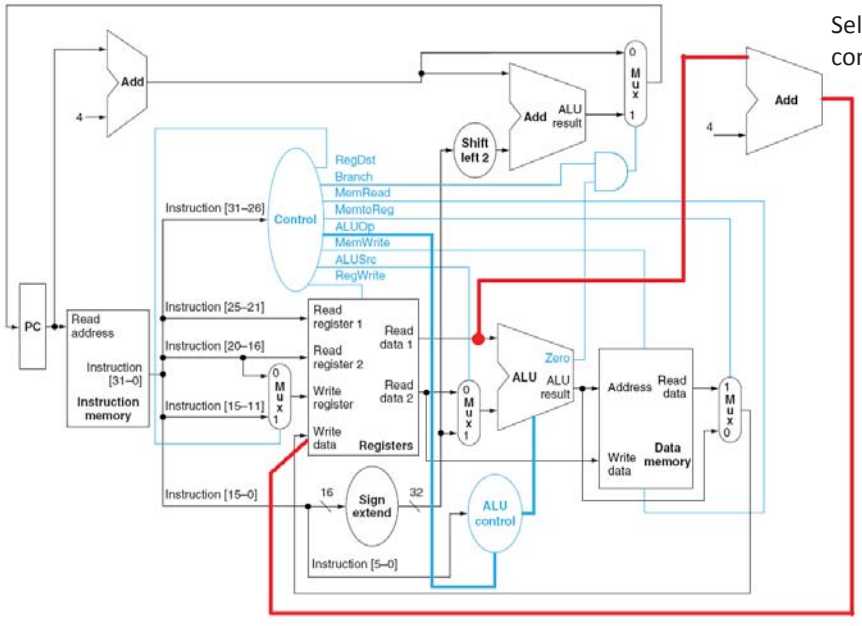


In the single-cycle datapath, the ALU computes the address. Where do we increments ‘rs’ by 4? We put a new adder.

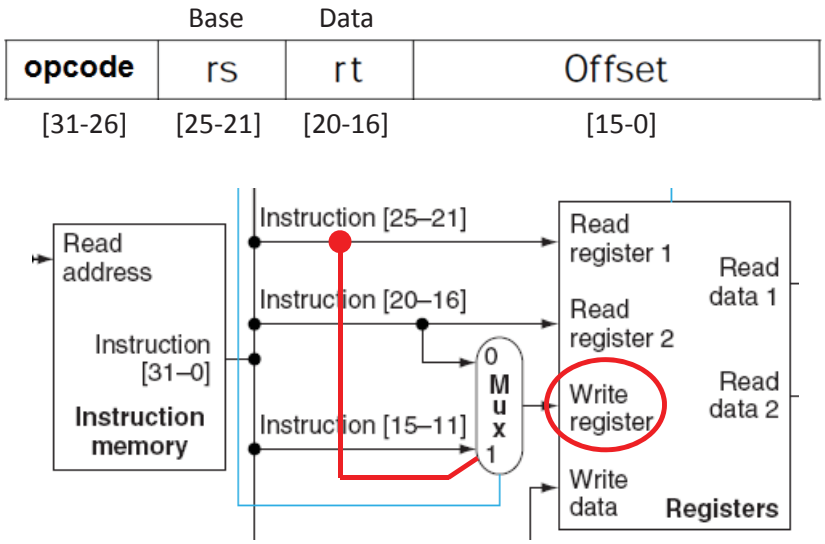


The multiplexer at the register file selects between the original line and the output of the new adder.

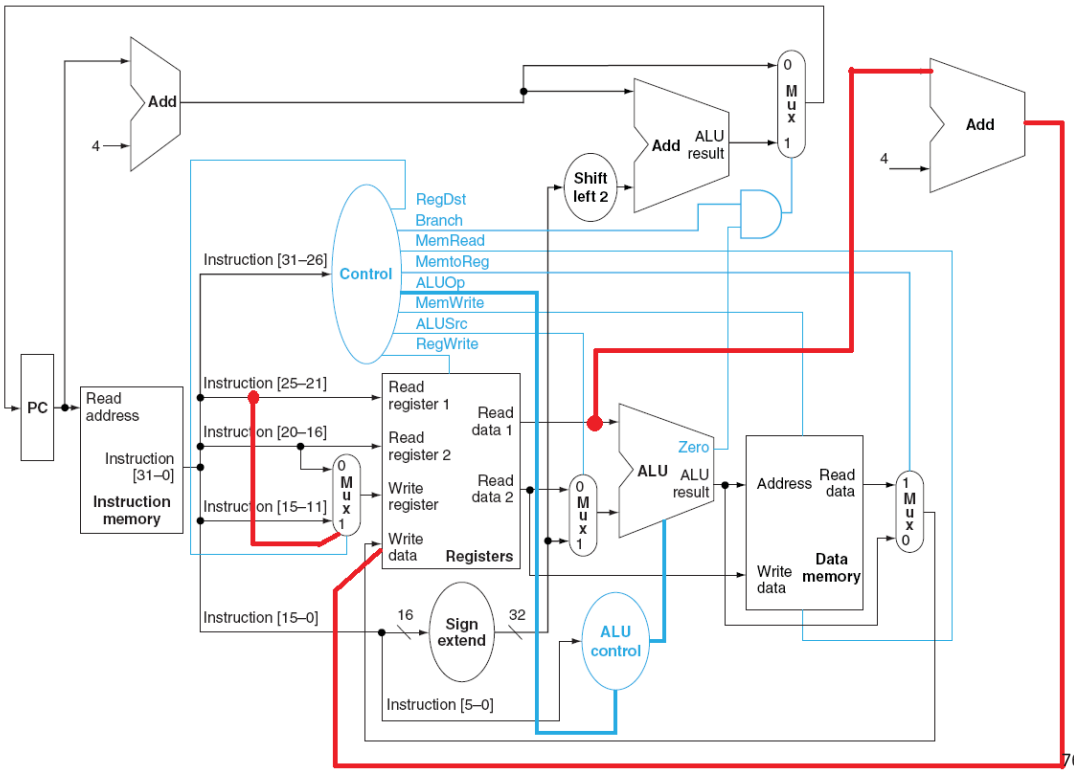




- The datapath is set up to write to fields [20-16] or [15-11]
- However, with ‘swi’ we want to write to field [25-21]
- So we’ll add a new line to do this
- Now we make ‘RegDst’ a 2-bit selector. It’s equal to 2 for ‘swi’

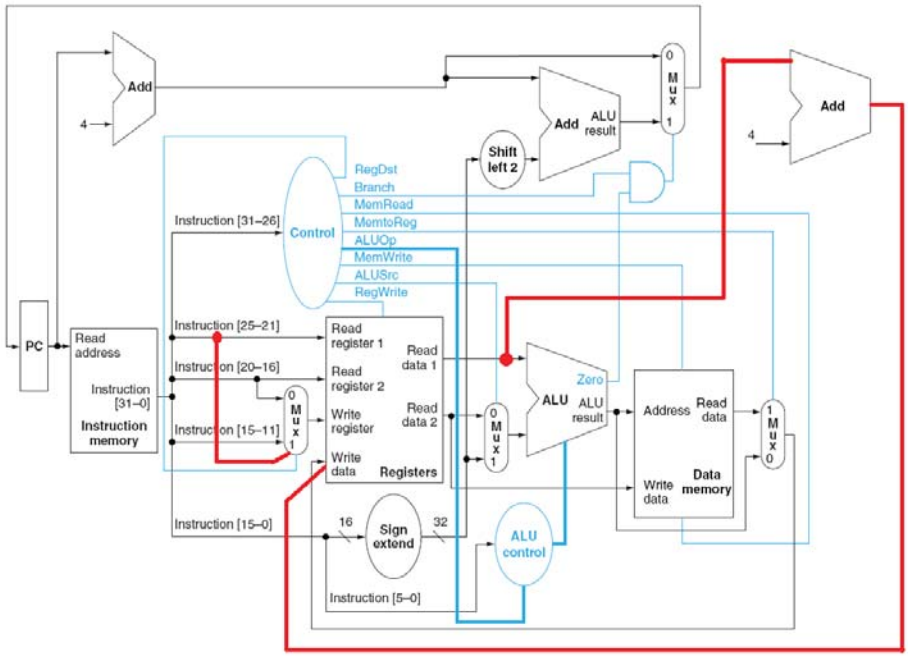


We added a new line from [25-21] to the register ‘Write Data’ input



Let's find all the control signals for 'swi'

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



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

Exercises (2)

- 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

Draw the changes