Lecture 6, Datapath and Control

CISC vs RISC today

PC Era

- Hardware translates x86 instructions into internal RISC instructions
- Then use any RISC technique inside MPU
- > 350M / year !
- x86 ISA eventually dominates servers as well as desktops

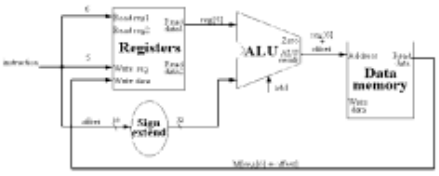
PostPC Era: Client/Cloud

- IP in SoC vs. MPU
- Value die area, energy as much as performance
- > 20B total / year in 2017
  - x86 in PCs peaks in 2011, now decline ~8% / year (2016 < 2007)
  - x86 servers ⇒ Cloud ~10M servers total (0.05% of 20B)
- 99% Processors today are RISC

Currently 99% of processors are RISC.

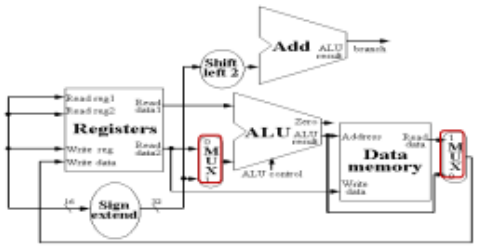
Datapath and control: 3-step derivation

1. different datapaths for:
   - register-based instructions e.g. add, sub
   - memory-access instructions e.g. lw, sw
   - branch instructions e.g. beq, j

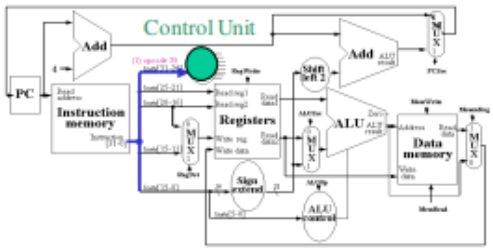


2. combined datapath:

   for different instructions
   - add multiplexors (without control signals)



3. Control Unit: activate relevant parts of the combined datapath for a given instruction
   - control signals for multiplexors + ALU



Similar to how we understand the ALU, for the datapath we will start from simple and obvious designs which are then transformed into more complex and realistic ones. The datapath looks complex because it needs to support different types of instructions. So we will break down the development of a MIPS datapath into the following three steps.

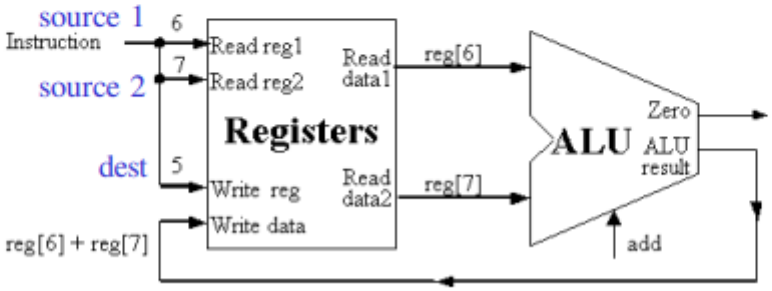
1. Start with a datapath developed specifically for a particular type of instruction, e.g. R-type. Since it is designed only for that instruction, it is simple and obvious.
2. Once we have a datapath for R-type instructions and another datapath for I-type instructions, we produce a combined datapath by adding multiplexors; we repeat this until the combined datapath can cover all the instructions of interest to us.
3. We need the combined datapath to behave like the R-type datapath when dealing with R-type instructions, and to behave like the I-type datapath when dealing with I-type instructions; this is achieved by having a Control Unit which would produce the appropriate control signals for the multiplexors and the ALU etc in the combined datapath.

Register-based instructions

- select read/write registers from operands
- select ALU operation from opcode and function code fields

opcode	source 1	source 2	dest	shift amt	fn code

- add $5, $6, $7 # reg[5] = reg[6] + reg[7]



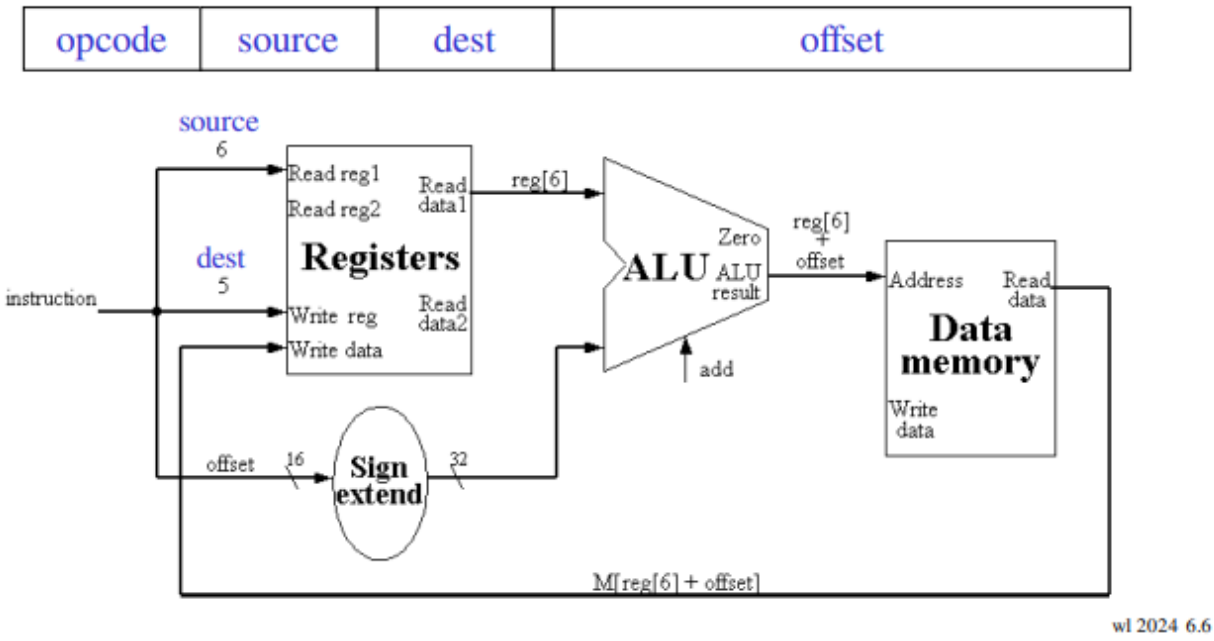
- completes in a cycle: single-cycle datapath

wl 2024

Lets begin Step 1 by considering the R-type datapath, which is simple. Just need to provide the appropriate fields in the instruction (such as source 1, source 2, dest etc) are sent to the relevant inputs of the Registers, and the two outputs from the Registers are connected to the two horizontal data inputs of the ALU. We also need to tell the ALU which function it should perform – this is indicated by the control signal at its vertical input which, as we would see later, is produced by taking into account the opcode and the `fn` code fields.

Memory access instructions: load

- lw $5, offset($6) # reg[5] = M[reg[6] + offset]
  assume data memory to return value in same cycle



wl 2024 6.6

For Load instructions, again the appropriate fields in the instruction are connected to the Registers and to one of the inputs of the ALU. There is a sign extension circuit that will extend the 16-bit offset field to 32 bits for the ALU. The ALU computes an address for the Data Memory, which would produce the data for the register specified by the dest field in the instruction.

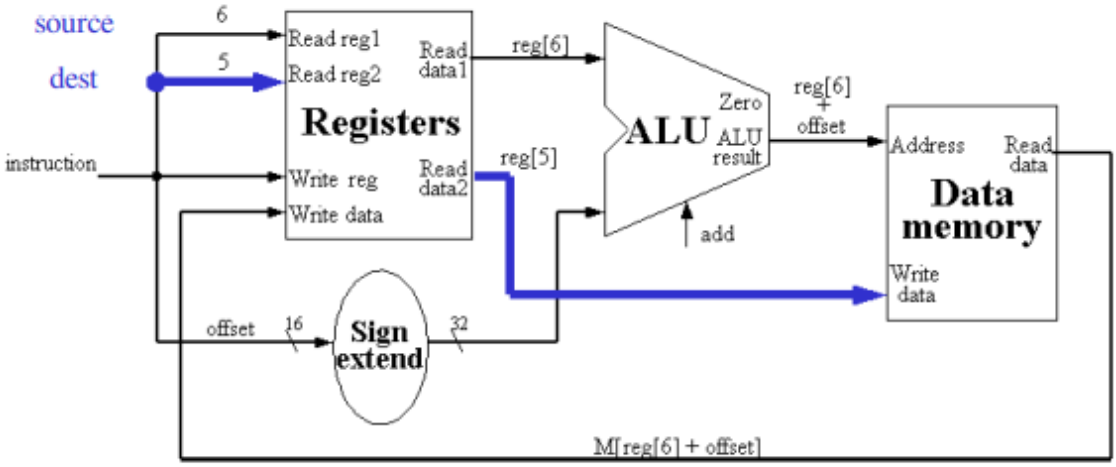
Memory access instructions: store

- sw $5, offset($6) # M[reg[6] + offset] = reg[5]

source dest

opcode	source	dest	offset

- connections for store

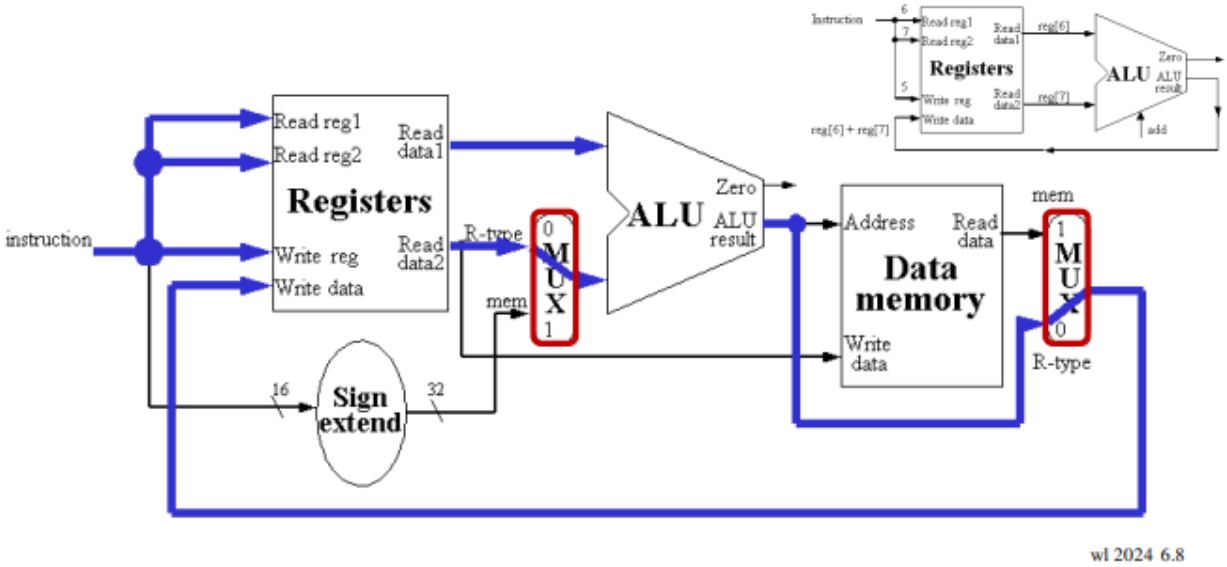


wl 2024 6.7

Store instructions are similar except that the dest field is used to specify the register whose value would be stored in a memory location with address given by the sum of: (a) the content of the register specified by the source field, and (b) the sign-extended offset field. Note that the datapath for the Store instruction is obtained by adding the two highlighted connections to the datapath for the Load instruction, so it can still support the Load instruction.

Combine datapaths for R-type and memory instructions: R-type flow

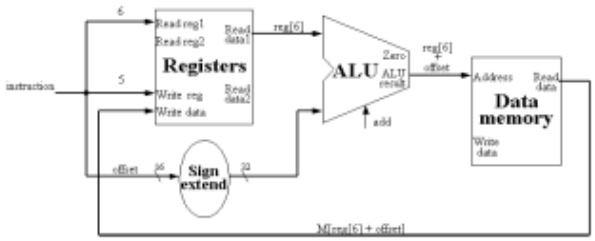
- 1 register file, 1 ALU
- ALU input mux: data from register or instruction
- register write mux: data from ALU or memory

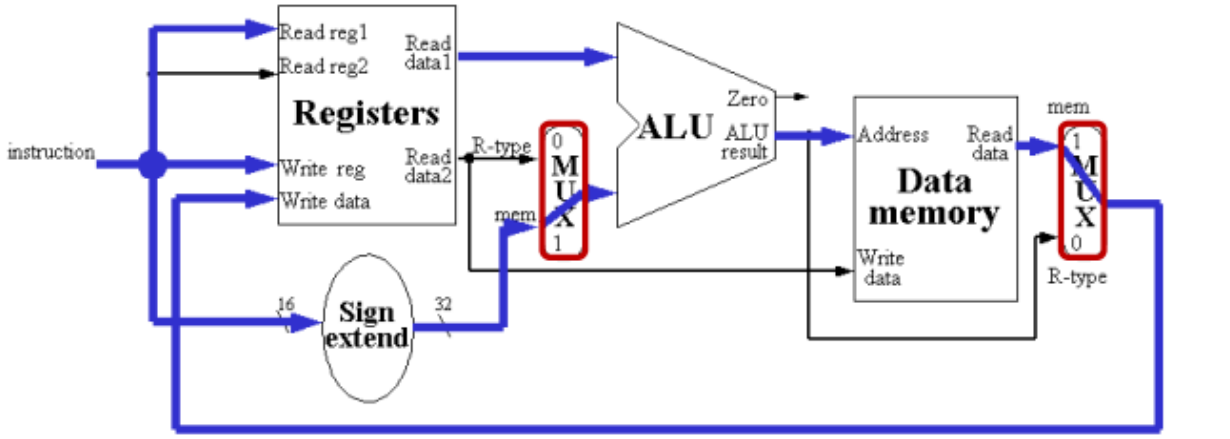


Lets begin Step 2: deriving a combined datapath for R-type and Load/Store instructions. We add two multiplexors to control: (a) the second input to the ALU, and (b) the Write data input to the Registers. When the two multiplexors receive control signals to behave as shown in the diagram, then this combined datapath behaves exactly like the R-type datapath as shown by the active connections which are highlighted.

Combine datapaths for R-type and memory instructions: memory flow

- 1 register file, 1 ALU
- ALU input mux
- register write mux



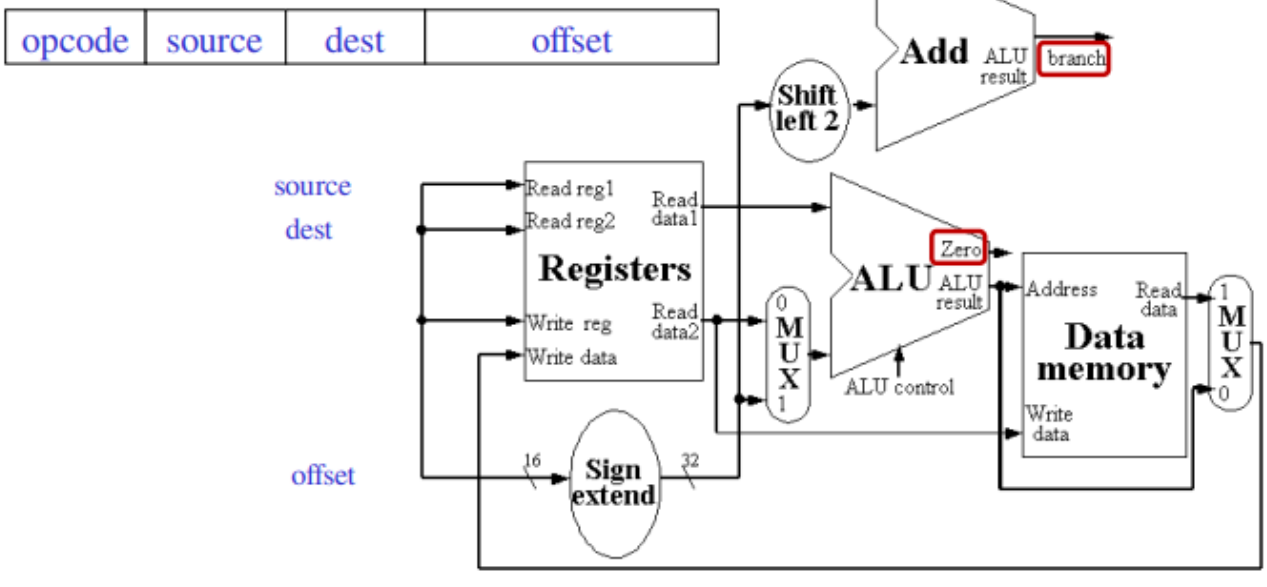


wl 2024 6.9

When the two multiplexors receive control signals to behave as shown in the diagram, then this combined datapath behaves like the Load datapath – check it out!

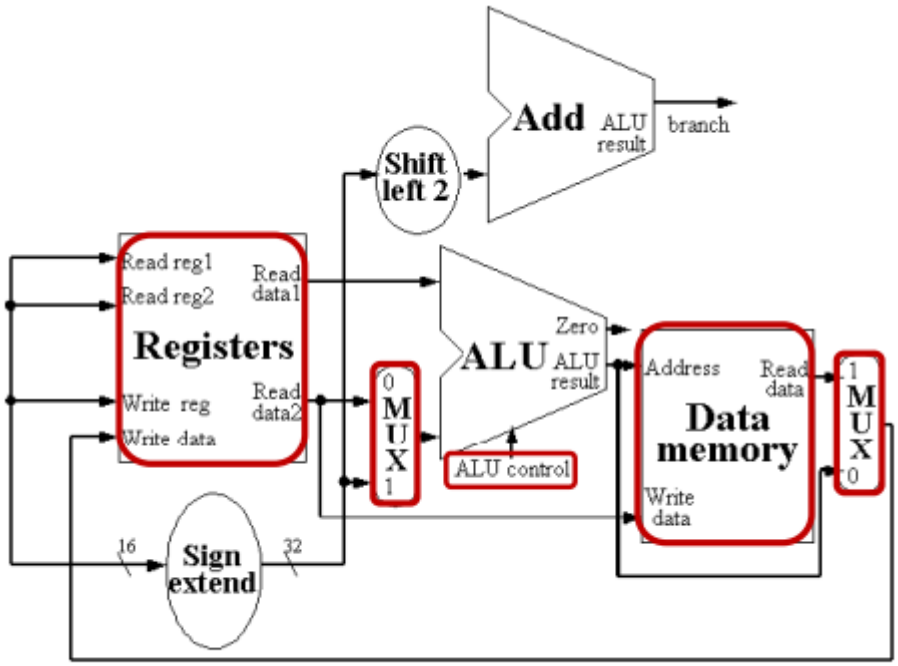
Branch instructions

- beq $5, $6, L



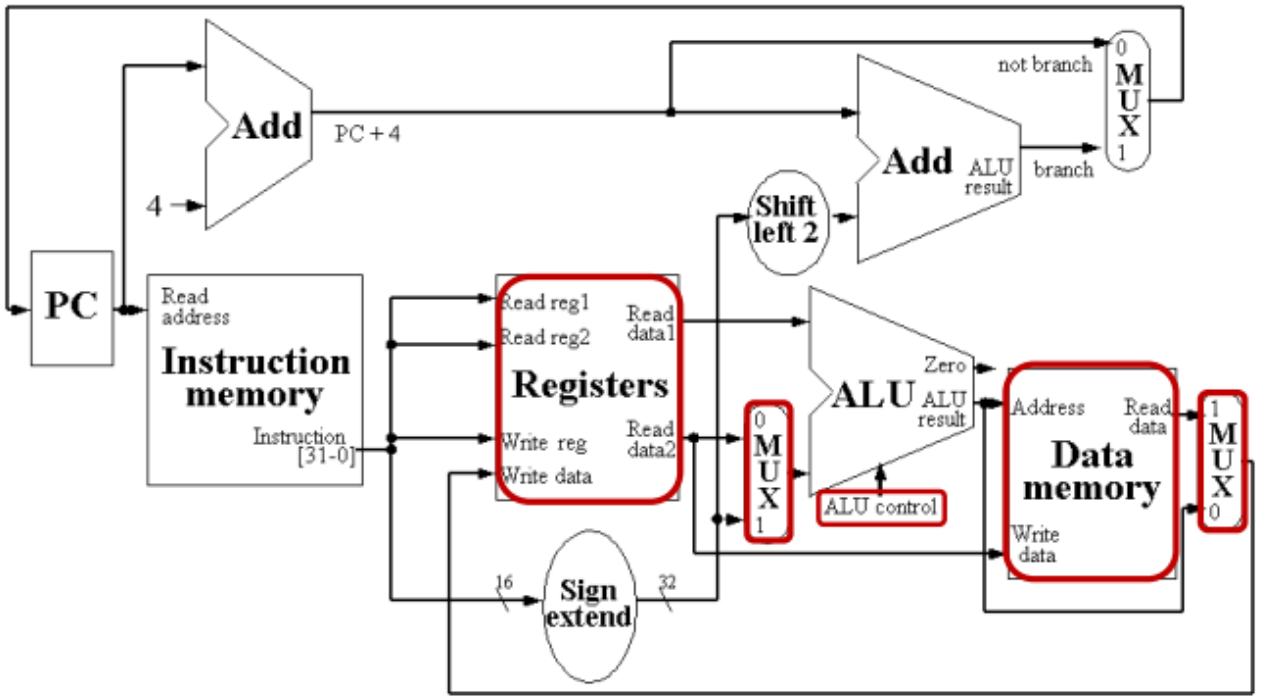
Now consider Branch instructions. The BEQ datapath needs to do two things: (a) to check if two registers contain the same value by subtracting one from the other and see if the zero output of the ALU is one or not; (b) if the zero output is one, then we need to find the amount of the branch specified by the offset field. Since offset specifies the number of instructions and each instruction takes 4 bytes, the corrsponding amount of memory addresses for the offset would be 4 x offset, or left shifting the offset by 2.

Combined datapath: without instruction memory



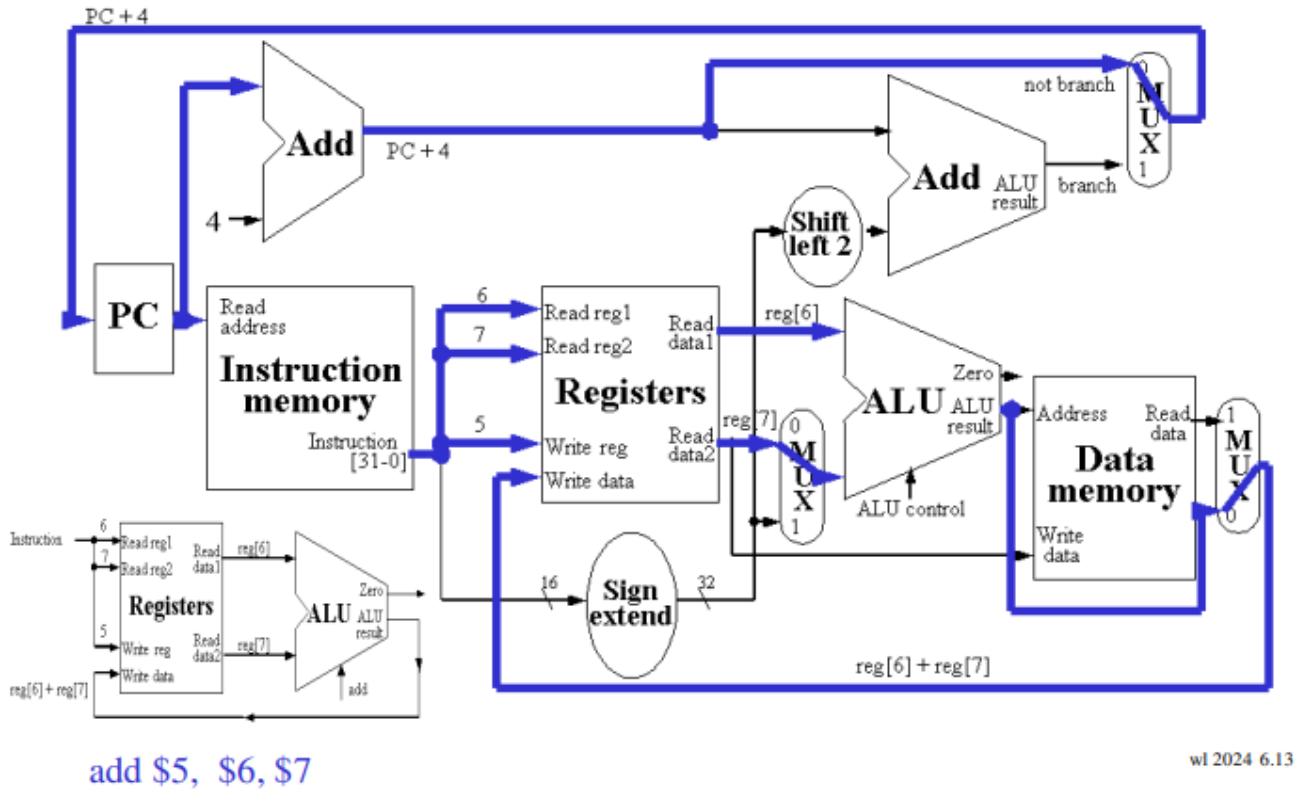
Now we have a combined datapath that can support several instructions. Let us highlight the components that would require control signals for correct operation. These are the two multiplexors, the ALU, the Registers, and the Data Memory (with control signals specifying whether to read or to write).

Combined datapath: with instruction memory



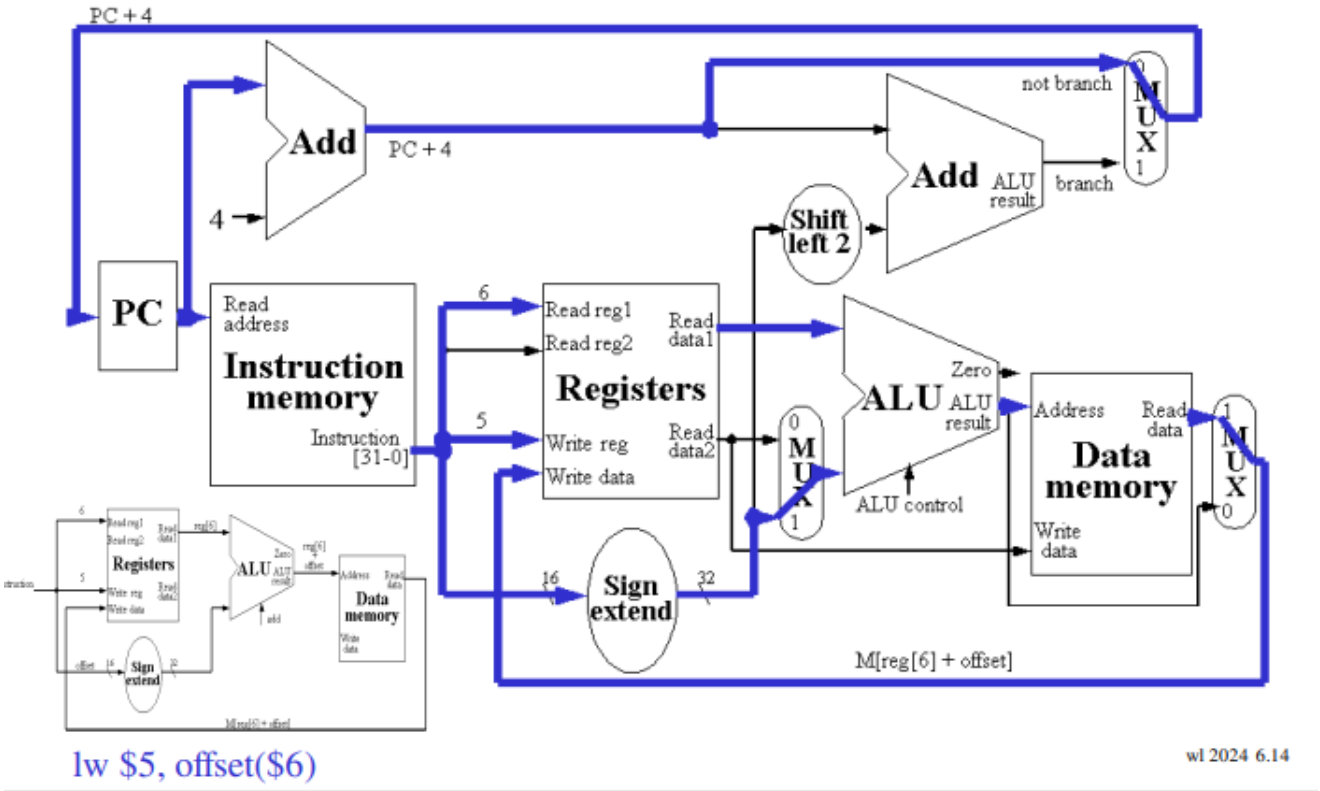
Let us extend the combined datapath by including the program counter PC, the Instruction Memory, the adder for incrementing the PC, and the multiplexor for selecting either the branch address or the not branch address for the PC.

Combined datapath for R-type



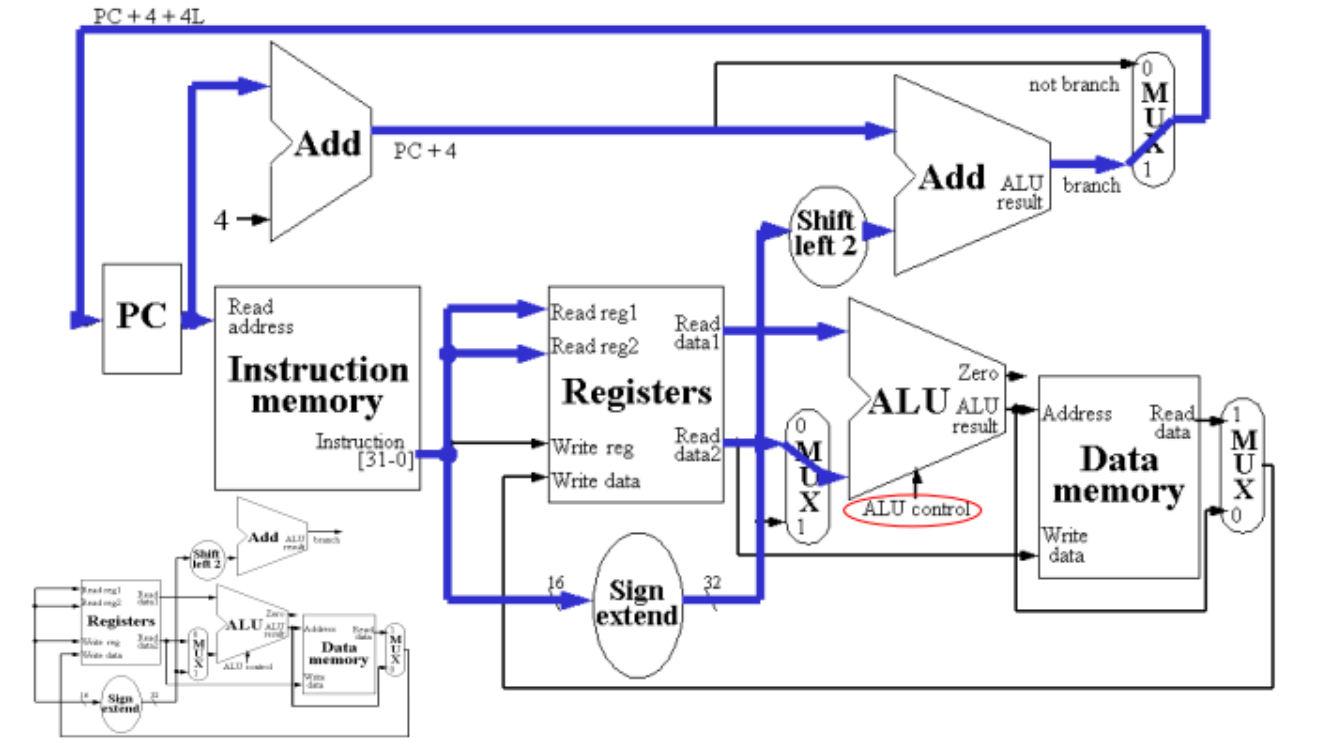
So let us check that, with the appropriate control signals for the R-type instructions such as Add, this combined datapath behaves like the R-type datapath as shown by the active connections

Combined datapath for load



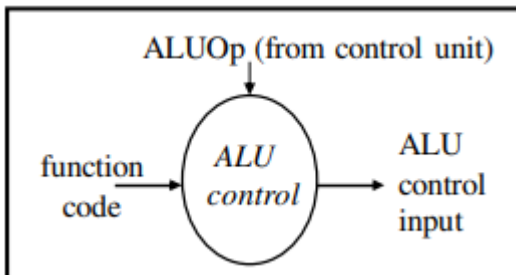
Similarly, with the appropriate control signals for the Load instruction, this combined datapath behaves like the Load datapath.

Combined datapath for branch



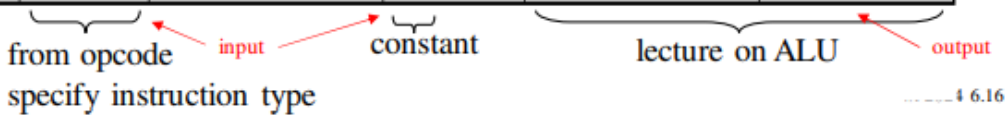
Check one more time: with the appropriate control signals for the BEQ instruction, this combined datapath behaves like the BEQ datapath. Next, lets look at the ALU control signal: where does it come from?

Logic for ALU control



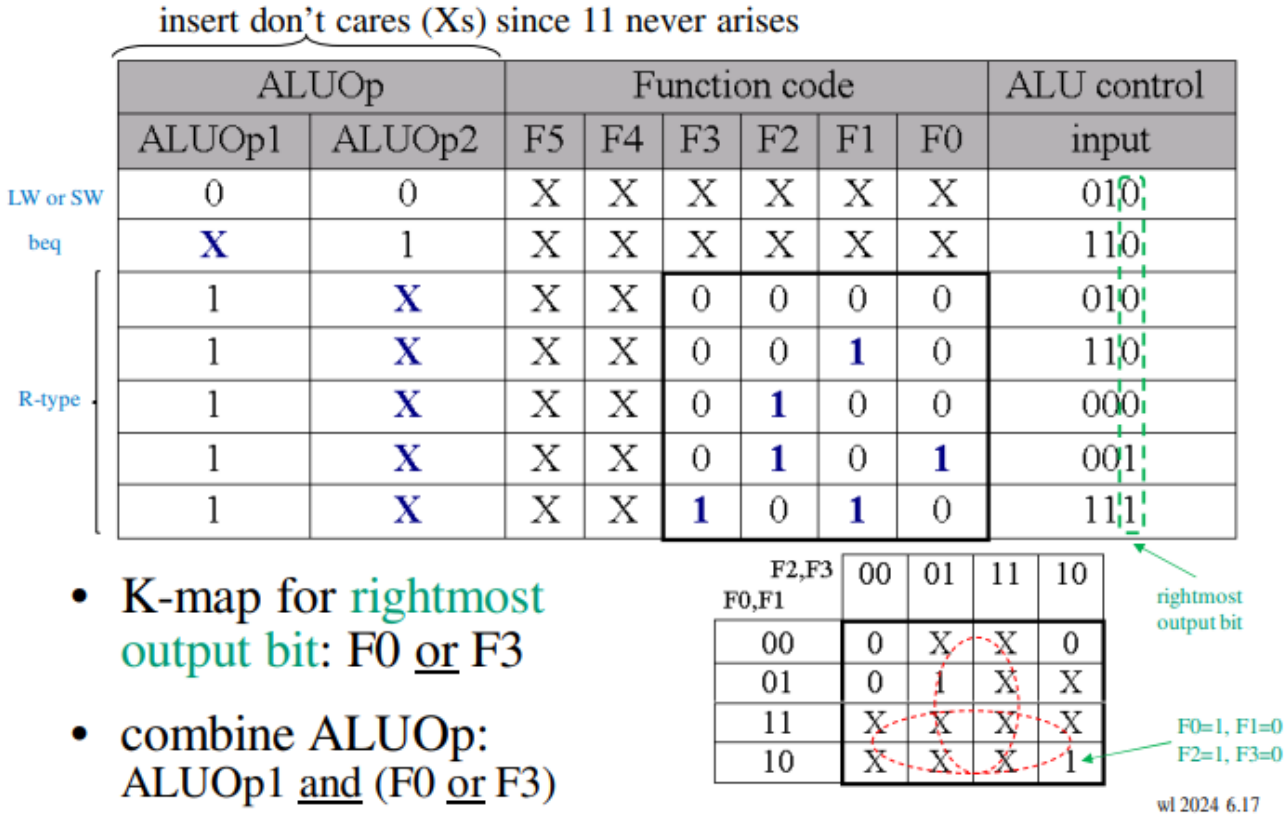
- from opcode and fn code, derive truth table for *ALU control*

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	**add**	**010**
SW	00	store word	XXXXXX	**add**	**010**
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	**add**	**010**
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set-on-less-than	101010	set-on-less-than	111



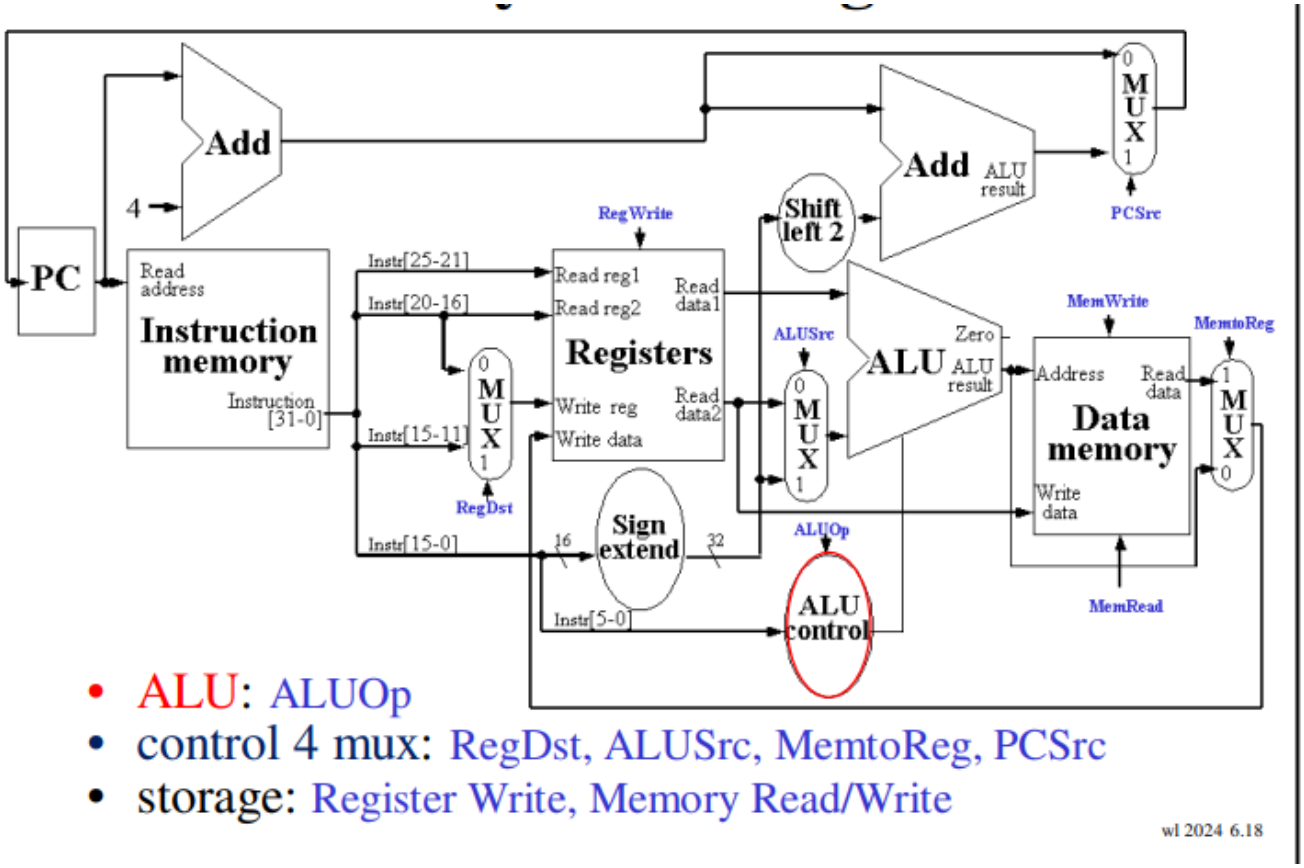
The ALU control signal comes from a block called ALU control, which takes as input the function code field and the `ALUOp` signal from the Control Unit. The Control Unit takes the opcode field as its input – we will cover it in a minute. The output of the ALU control block is called “ALU control input”, but beware that it is the output of this block! This table shows the truth table for the ALU control block. So for the Load instruction, the `ALUOp` signal from the Control Unit would be 00, and we need the ALU to perform addition, so the output to the ALU would be 010. Note that this value is different from the 110 value for the ALU to do addition on slide 4.11, but is compatible with Figure 4.12 in the course textbook. For R-type instructions, we need the function code field, in addition to the `ALUOp` signal, to specify the intended function for the ALU.

ALU control: optimisation



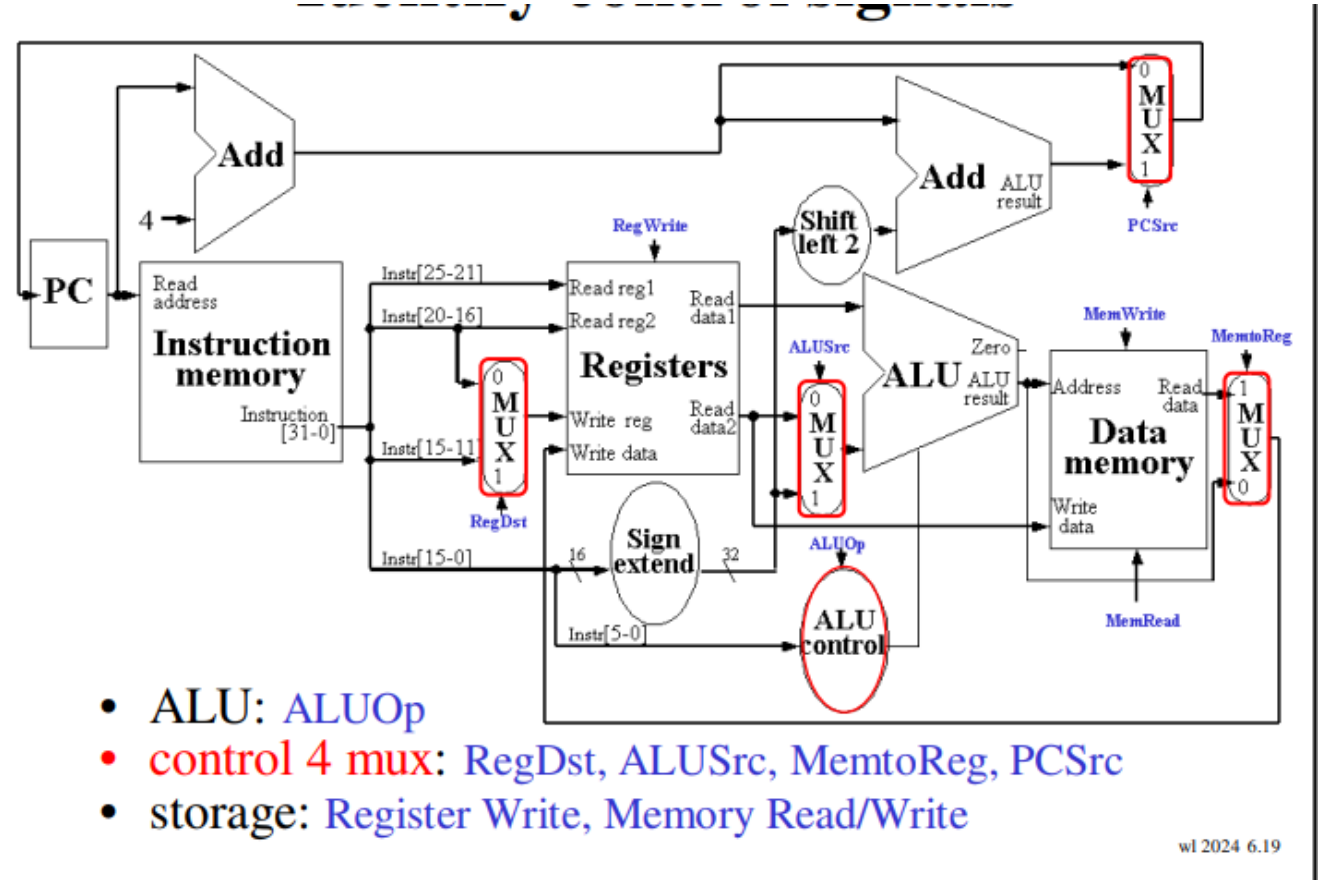
From the truth table, we can derive the logic equation for each of the three output bits of "ALU control input". One can use Karnaugh map techniques to obtain an optimised circuit for the rightmost output bit, with inputs from $F_0 . . F_3$ of the function code. This can then be "and" (Boolean conjunction) with `ALUOp1`, since when `ALUOp1=0`, then we want the output to be zero irrespective of the value of the function code.

Identify control signals



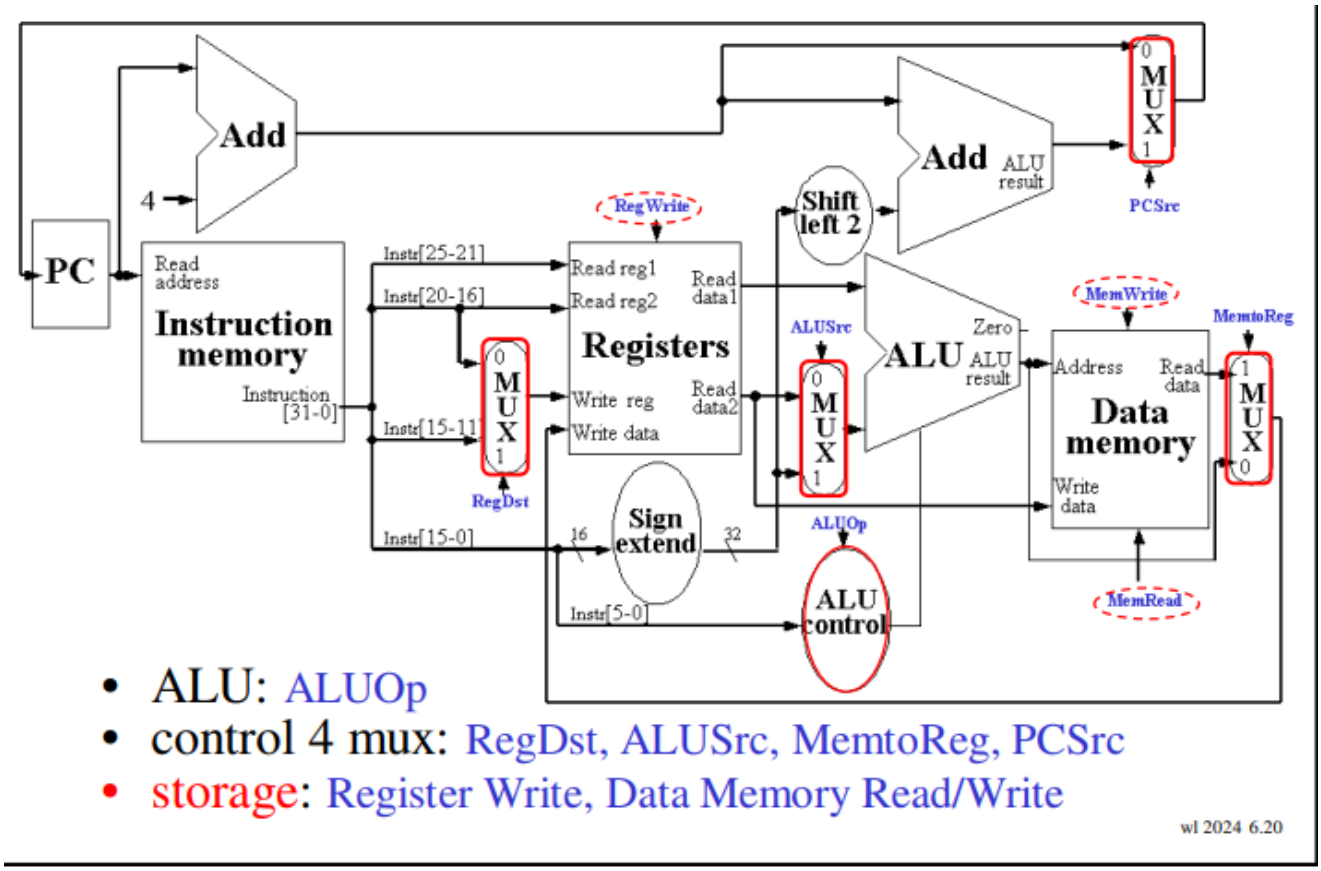
We are now ready for Step 3: deriving the control signals. Let us start by identifying all the control signals for this architecture. First, for controlling the ALU, there are 2 bits for `ALUOp`.

Identify control signals



Second, there are 4 multiplexors, each requiring a control input.

Identify control signals

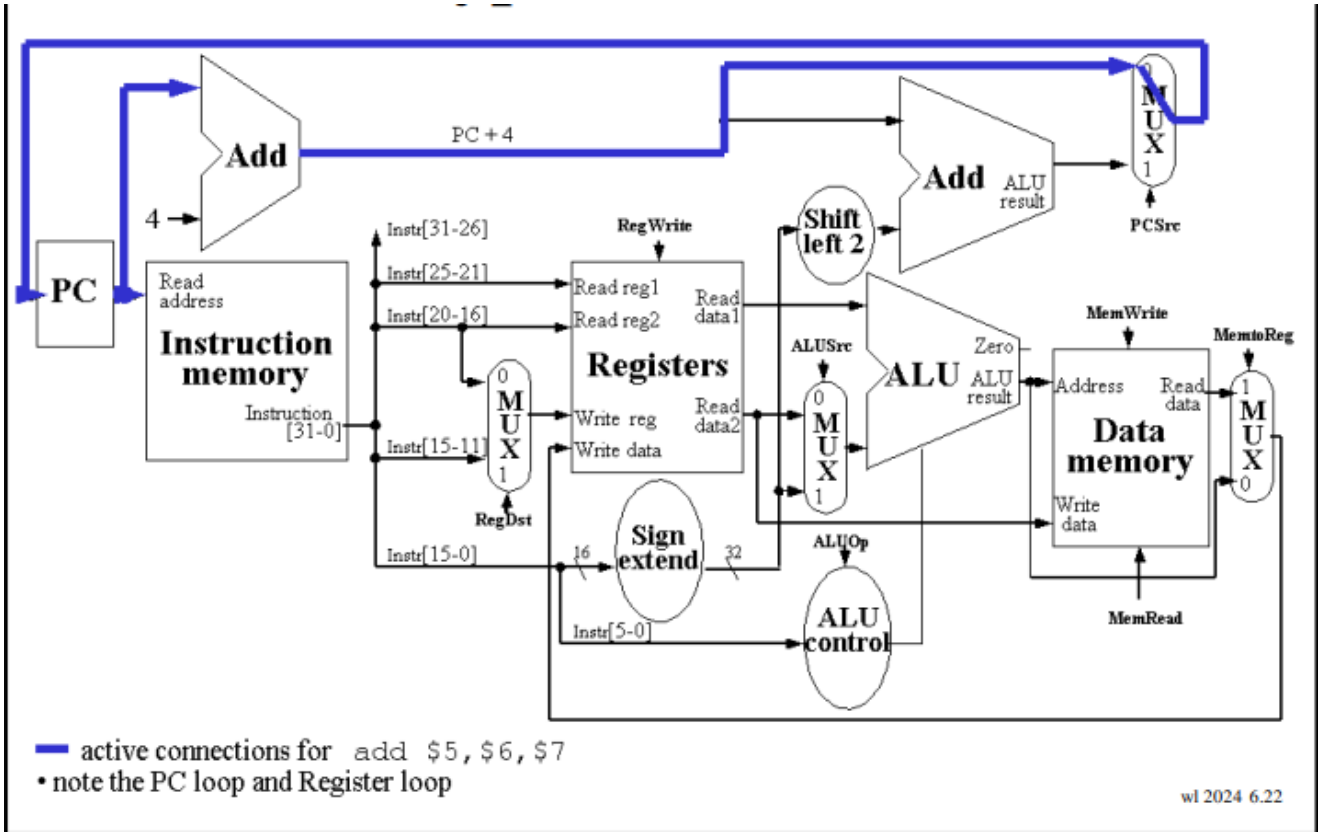


Third, we need a write signal for the Registers (`RegWrite`) and read (`MemRead`) and write (`MemWrite`) signals for the Data Memory.

Control signal summary

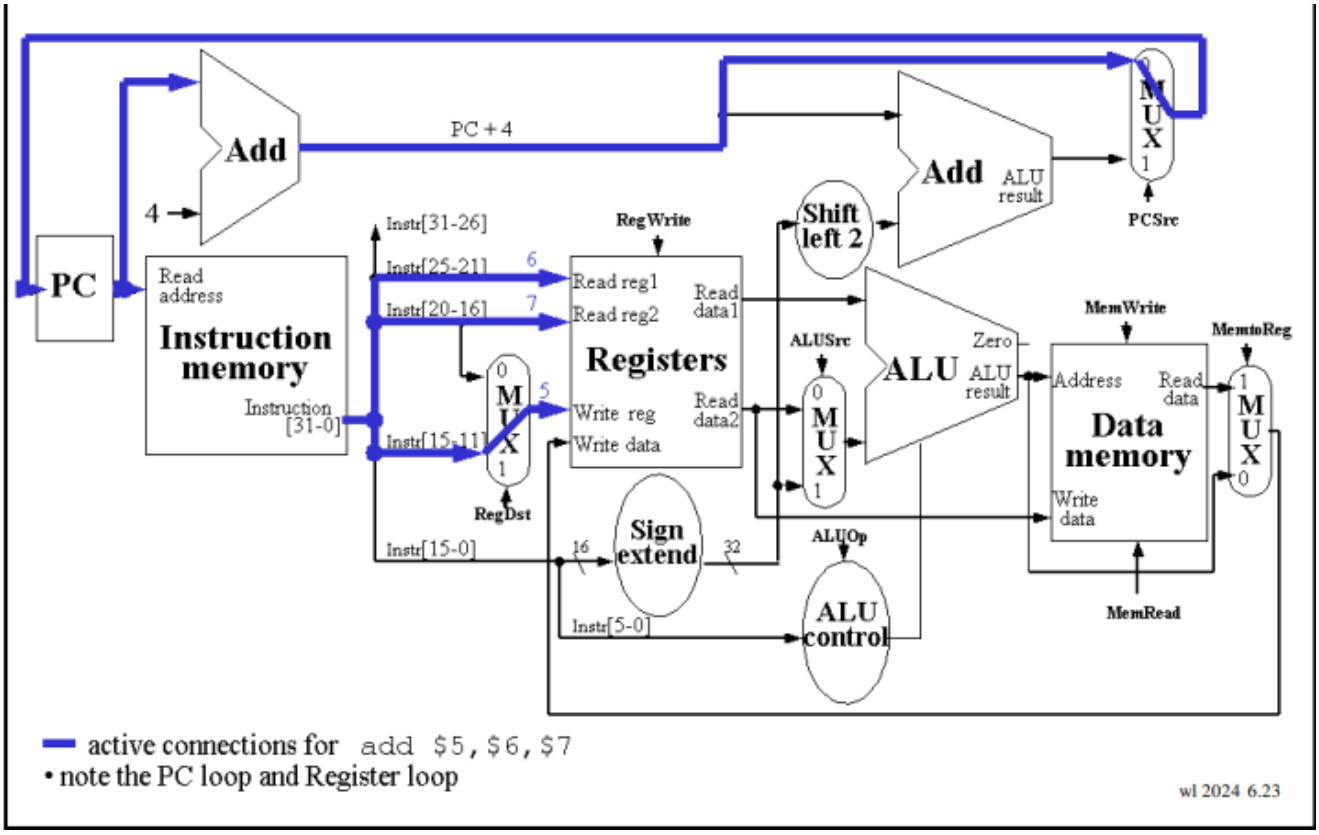
Signal name	Effect when deasserted	Effect when asserted
MemRead	None (R, sw, beq)	Data memory contents at the read address are put on read data output. (lw)
MemWrite	None (R, lw, beq)	Data memory contents at address given by write addtess is replaced by value on write data input. (sw)
ALUSrc	The second ALU operand comes form the second register file output. (R, beq)	The second ALU operand is the sign-extend lower 16-bits of the instruction. (lw, sw)
RegDst	The register destination number for the Write register comes from the rt field. (lw)	The register destination number for the Write register comes from the rd field. (R)
RegWrite	None (sw, beq)	The register on the Write register input is written into with the value on the write data input. (R, lw)
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target.
MemtoReg	The value fed to the register write data input comes from the ALU. (R)	The value fed to the register write data input comes from the data memory. (lw)
Branch	0 (R, lw, sw)	1 (beq)

R-type instruction(1)



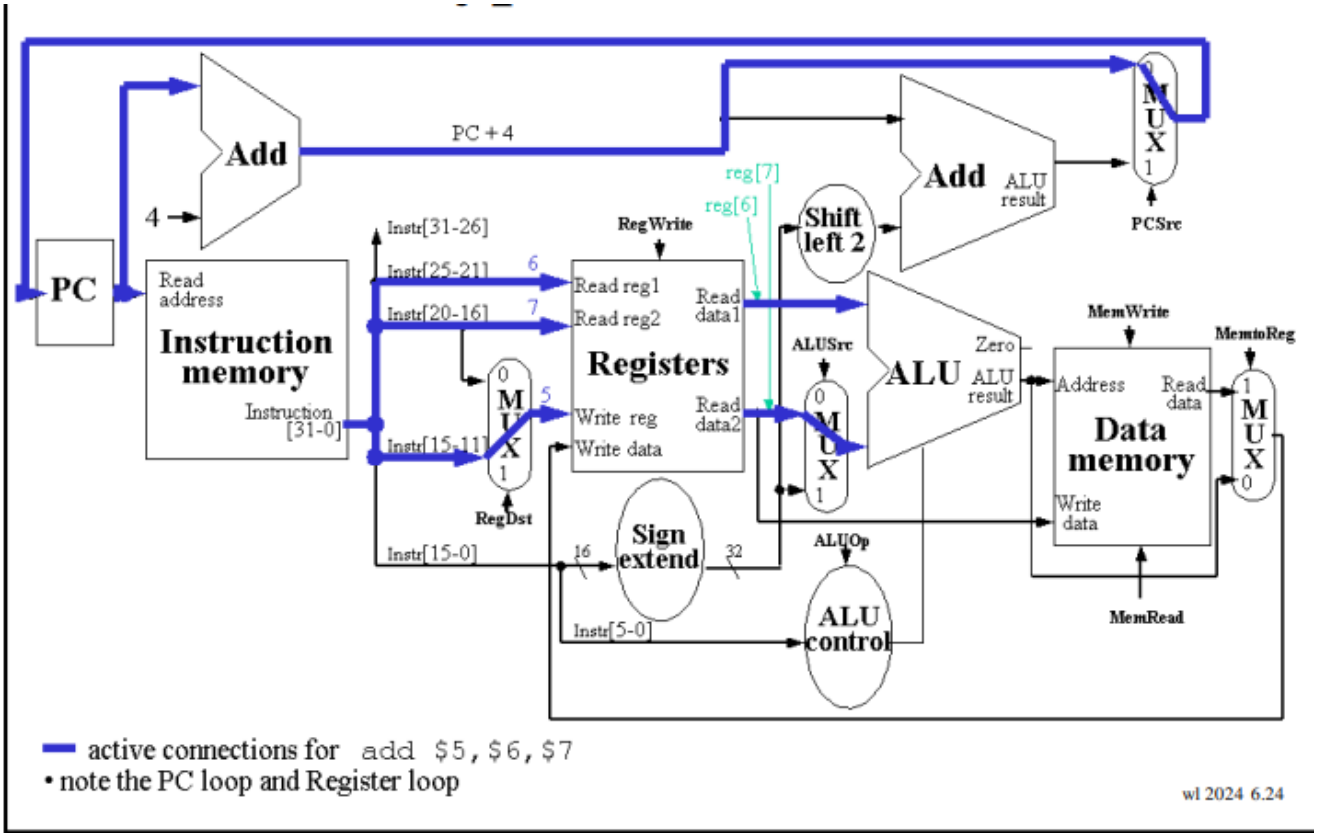
Let us now study the effects of the control signals. We begin by looking at how the R-type instruction is processed step by step by the combined datapath, given the appropriate control signals. First, the PC is incremented for the next instruction, while the current value of the PC is being used as an address to fetch the relevant instruction from the Instruction Memory.

R-type instruction(2)



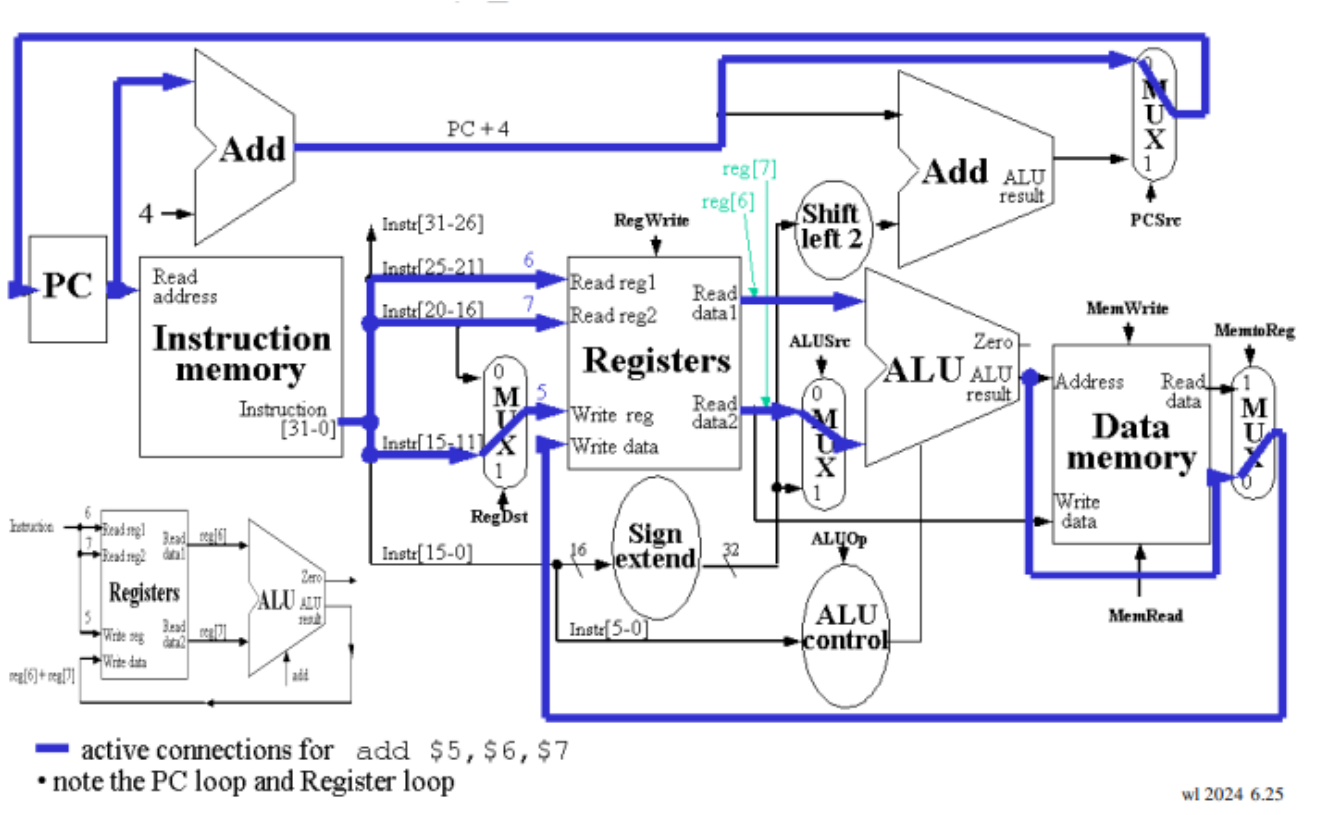
Next, the appropriate fields from the instruction are connected to the Registers.

R-type instruction(3)



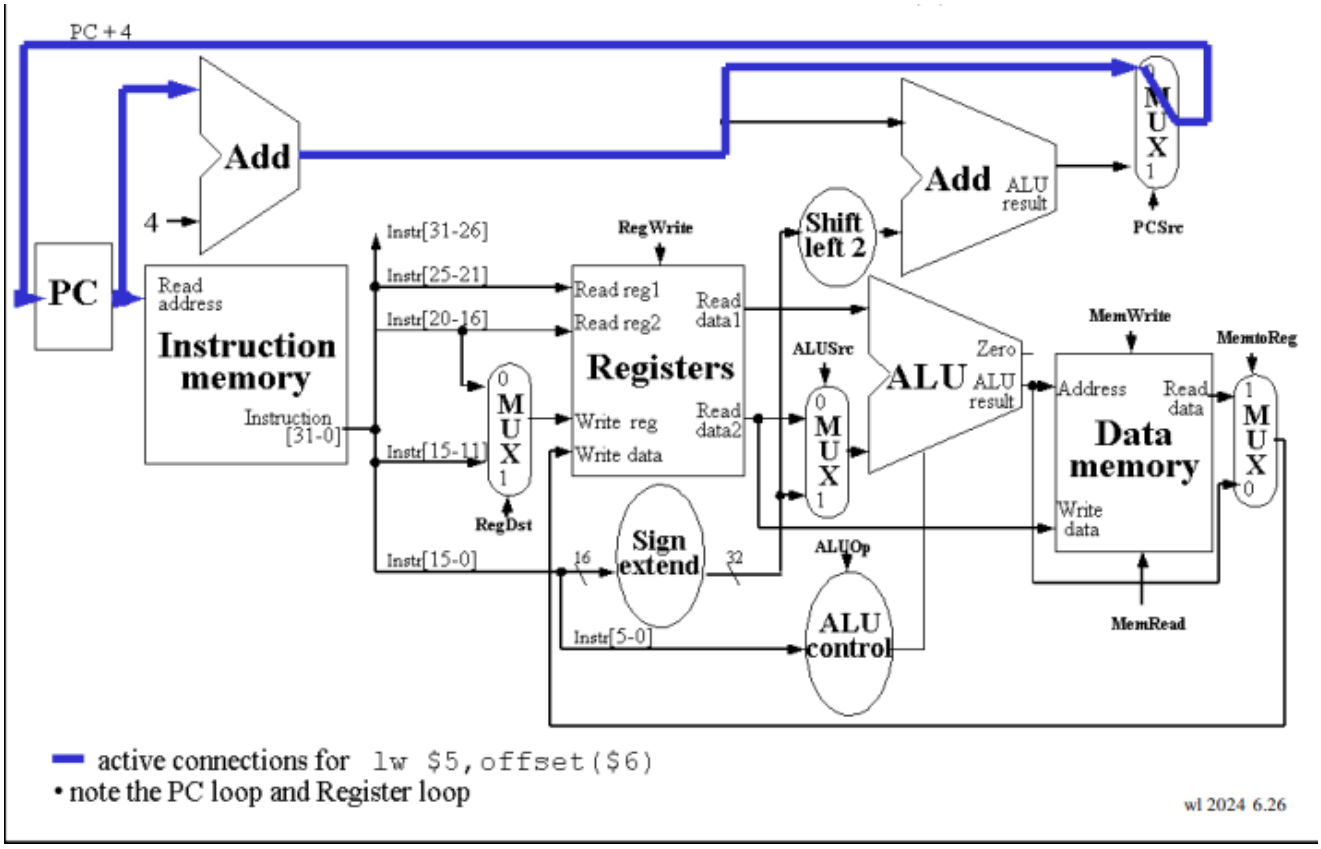
The values from the relevant source registers are then sent to the ALU.

R-type instruction(4)



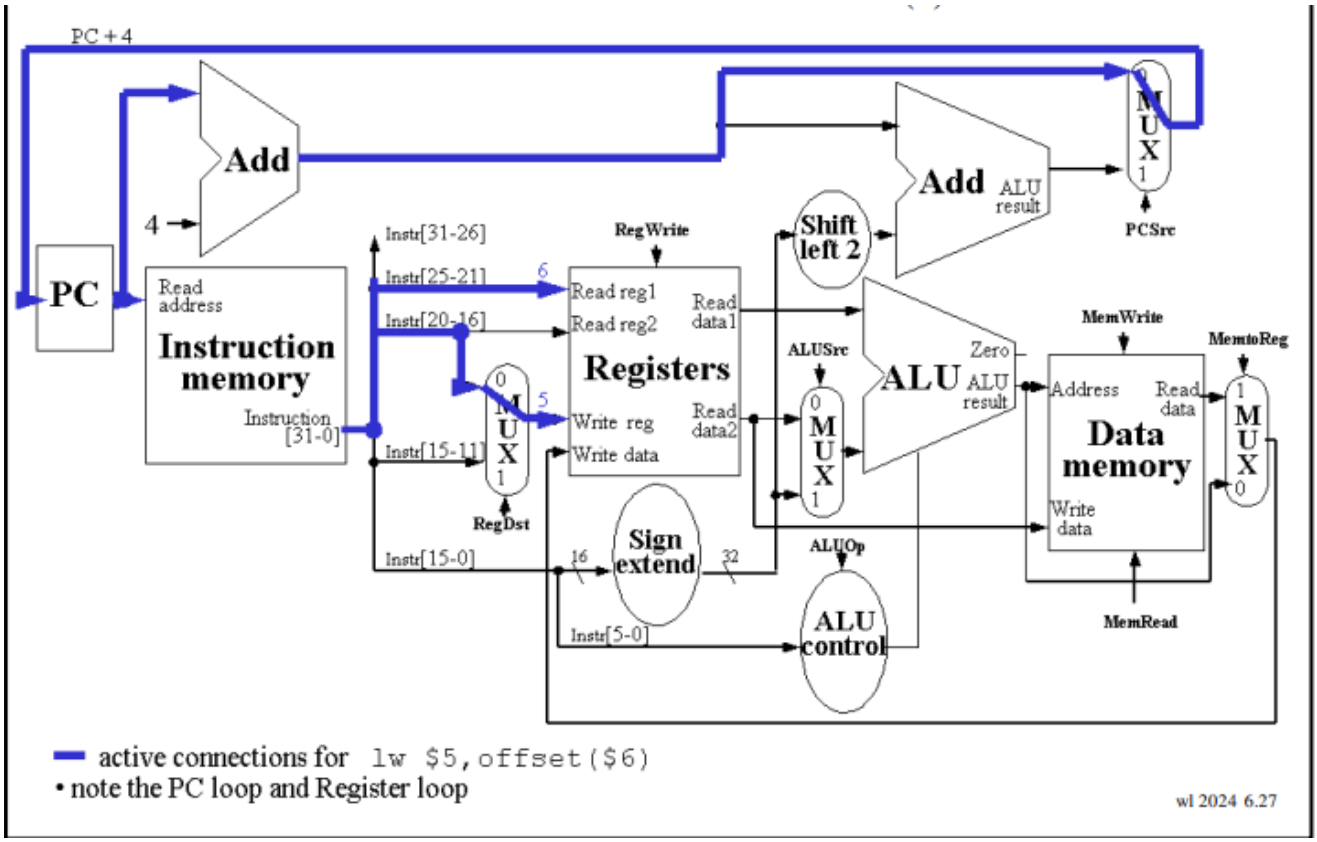
Finally, the result from the ALU is stored in the destination register. Note that all the above take place within one clock cycle. We can check, from the active connections, that the combined datapath behaves exactly the same as the R-type datapath.

Load instruction(1)



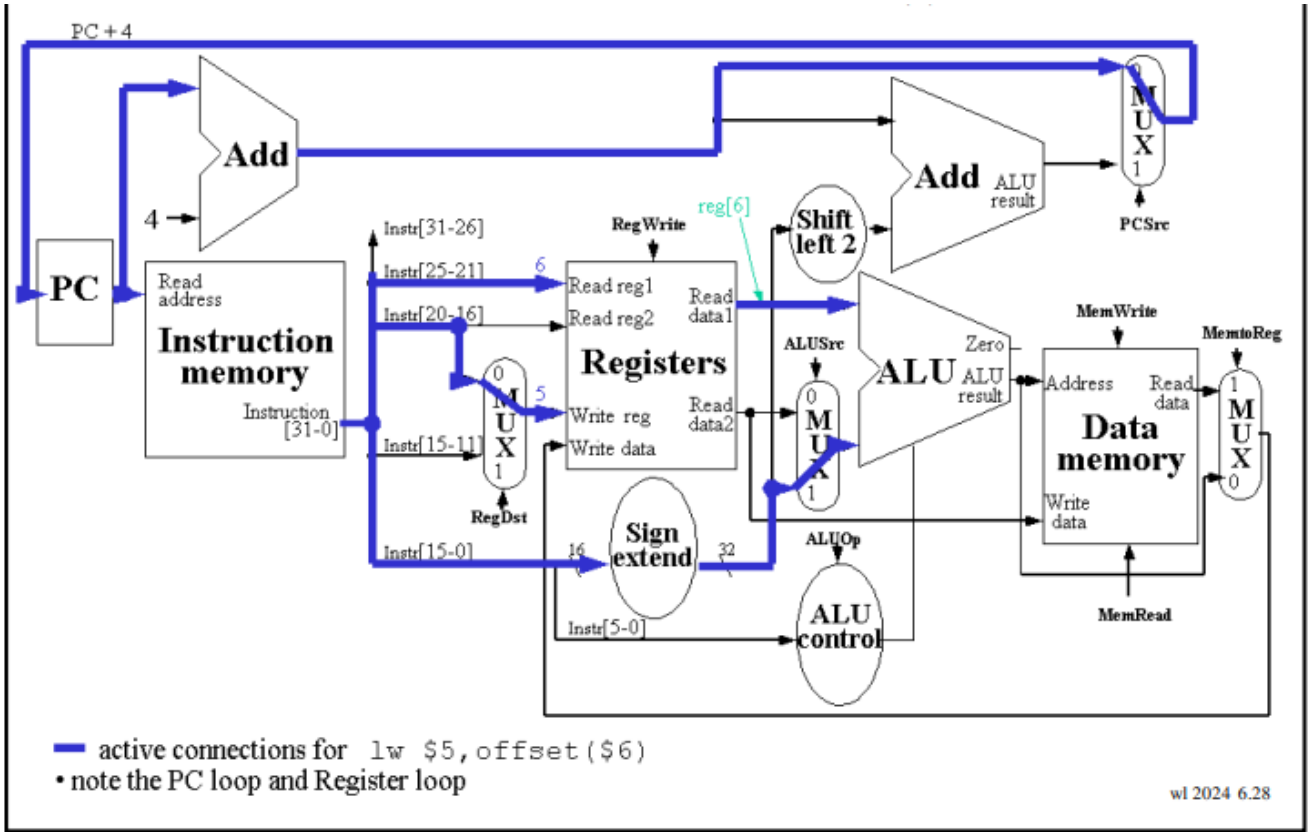
Now try the same for the Load instruction. The first step is the same for all instructions.

Load instruction(2)



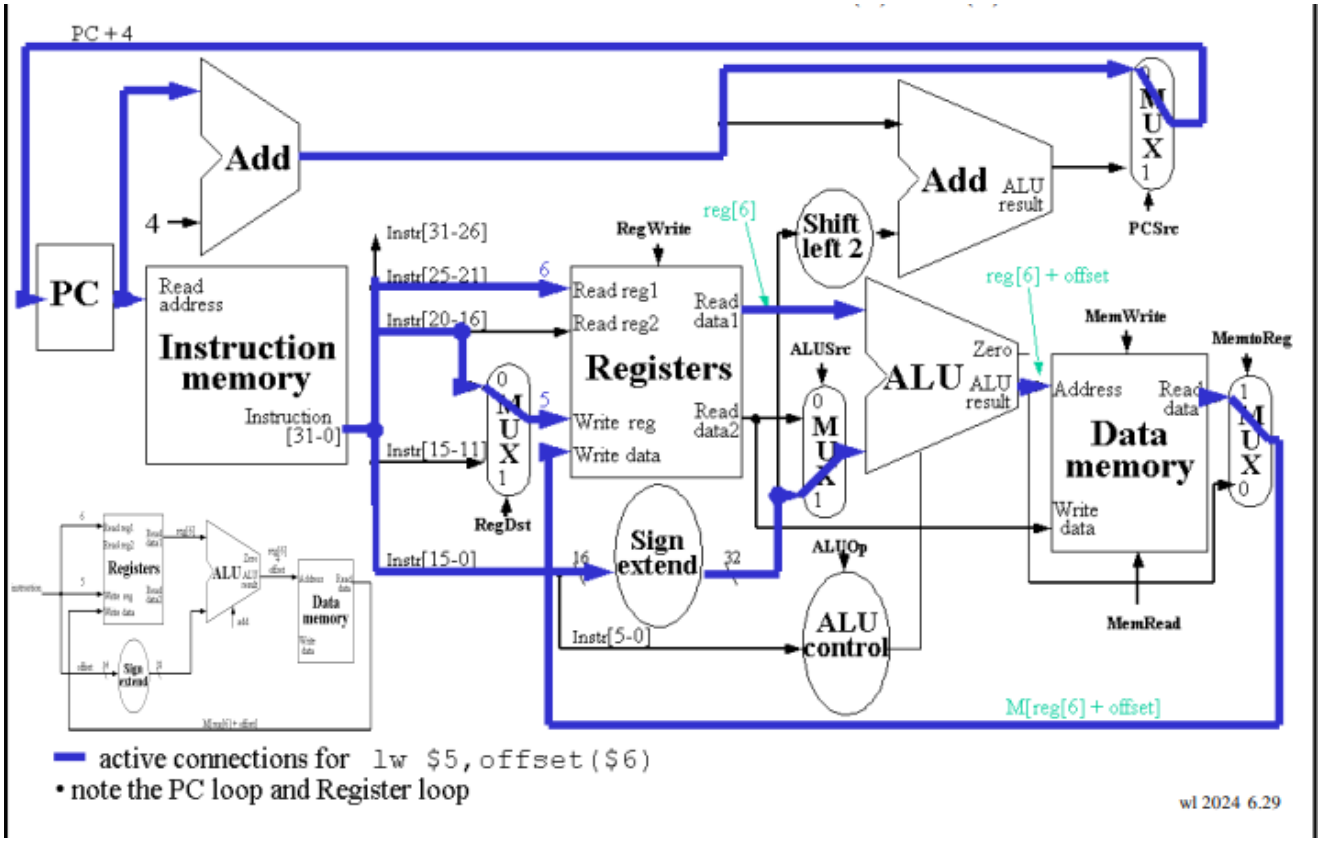
Then the source and destination fields from the instruction are connected to the Read reg1 and Write reg inputs to the Registers.

Load instruction(3)



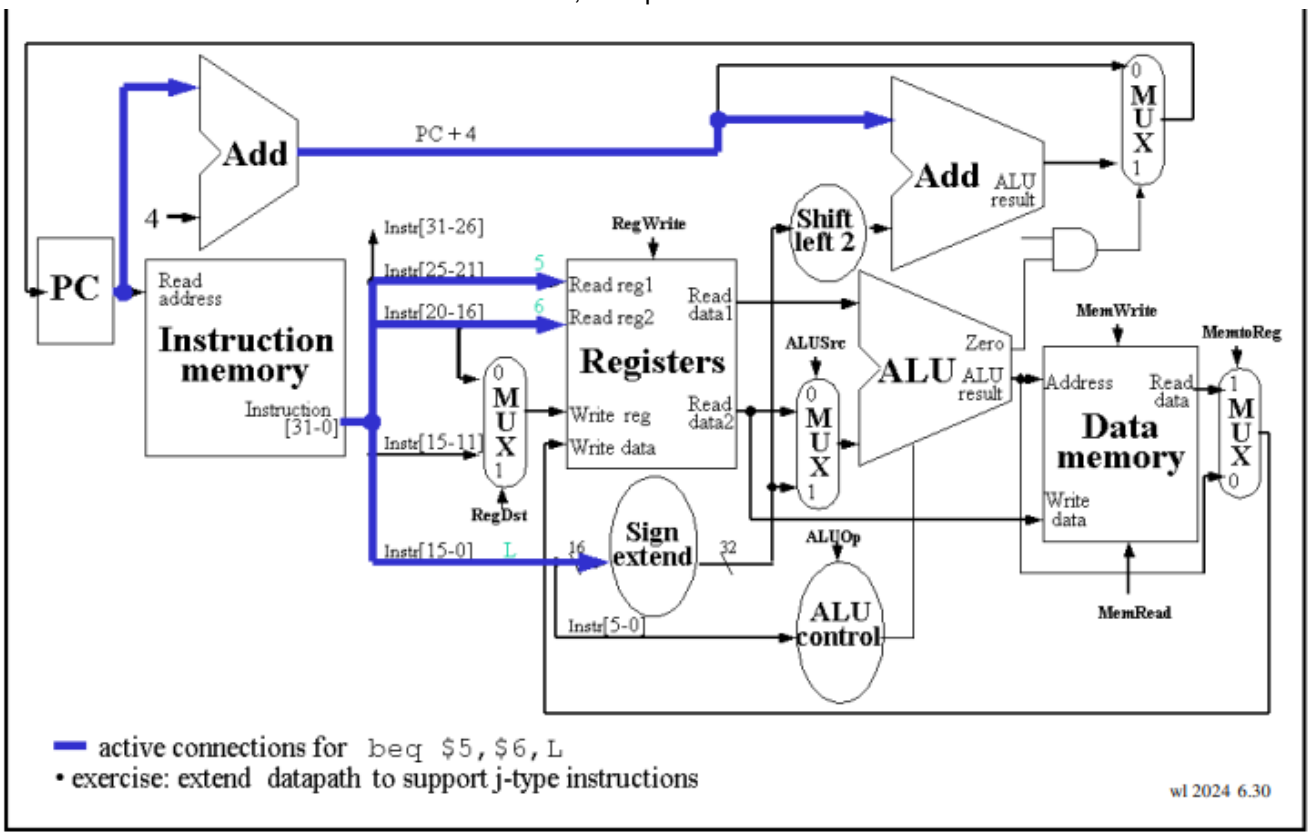
Then we get the data ready for the ALU…

Load instruction ((4) and (5))



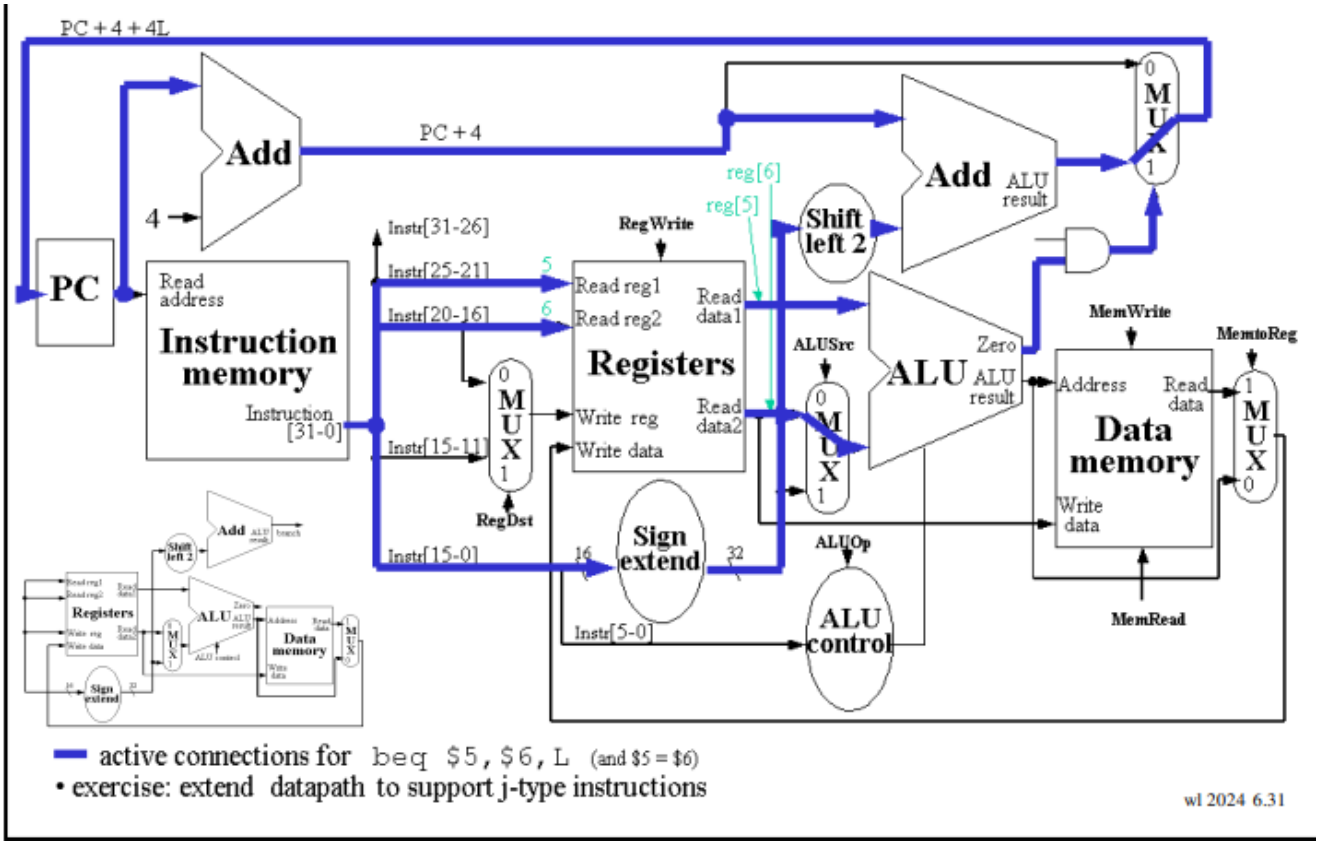
Next, the ALU produces an address for the Data Memory, which allows the corresponding data to be fetched and sent to the Registers. We can check that, again, from the active connections, the combined datapath behaves exactly the same as the Load datapath.

Branch instruction(1)



Lets do one more. The BEQ instruction starts the same as the other instructions.

Branch instruction(2)



We can check that, from the active connections, the combined datapath now behaves like the BEQ datapath.

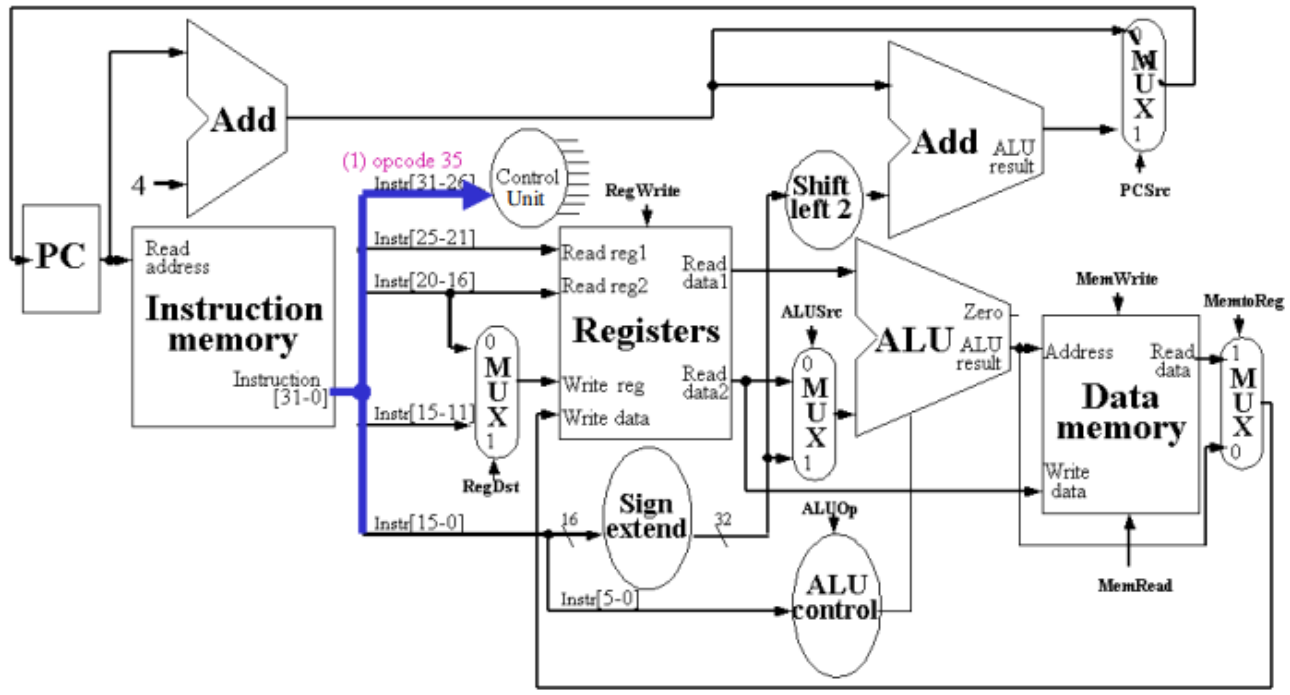
Control unit for single-cycle datapath

- combinational circuit: no registers
- 6-bit opcode input
  e.g. lw 100011 (35_{ten})
- 9-bit output, control mux, ALU, read/write op
- e.g. lw:

mux	memory/register	ALU
RegDst = 0	MemRead = 1	ALUOp = 0
Branch = 0	MemWrite = 0	
MemtoReg = 1	RegWrite = 1	
ALUSrc = 1		

Let us complete Step 3 by deriving the control signals for the Control Unit. The Control Unit for this single-cycle datapath is a combinational circuit: it takes the opcode as input and produces 8 control signals as output to control the single-cycle datapath – since `ALUOp` has 2 bits, so there are 9 bits output in total.

Load instruction(1)

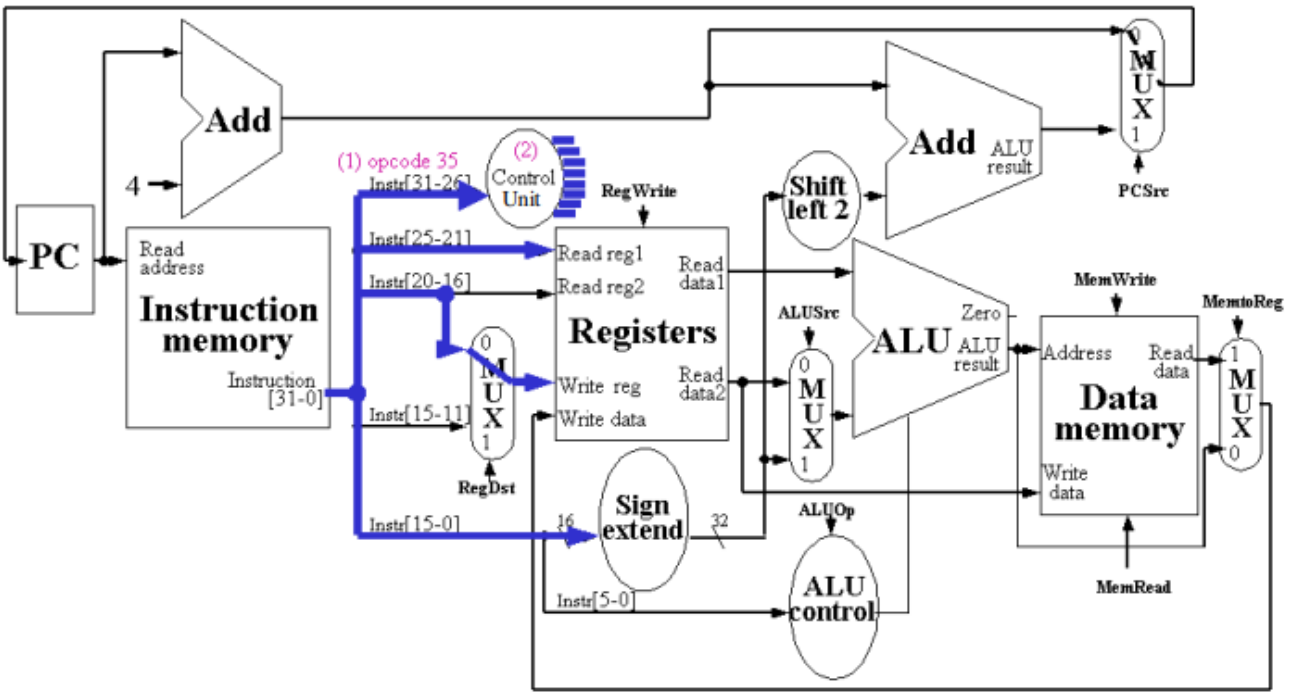


• lw $5,offset($6)

wl 2024 6.33

Now let us make the Control Unit explicit in the single-cycle datapath, and see how it supports the Load instruction. After the instruction is fetched from the Instruction Memory, the opcode field is used as input to the Control Unit.

Load instruction(2)

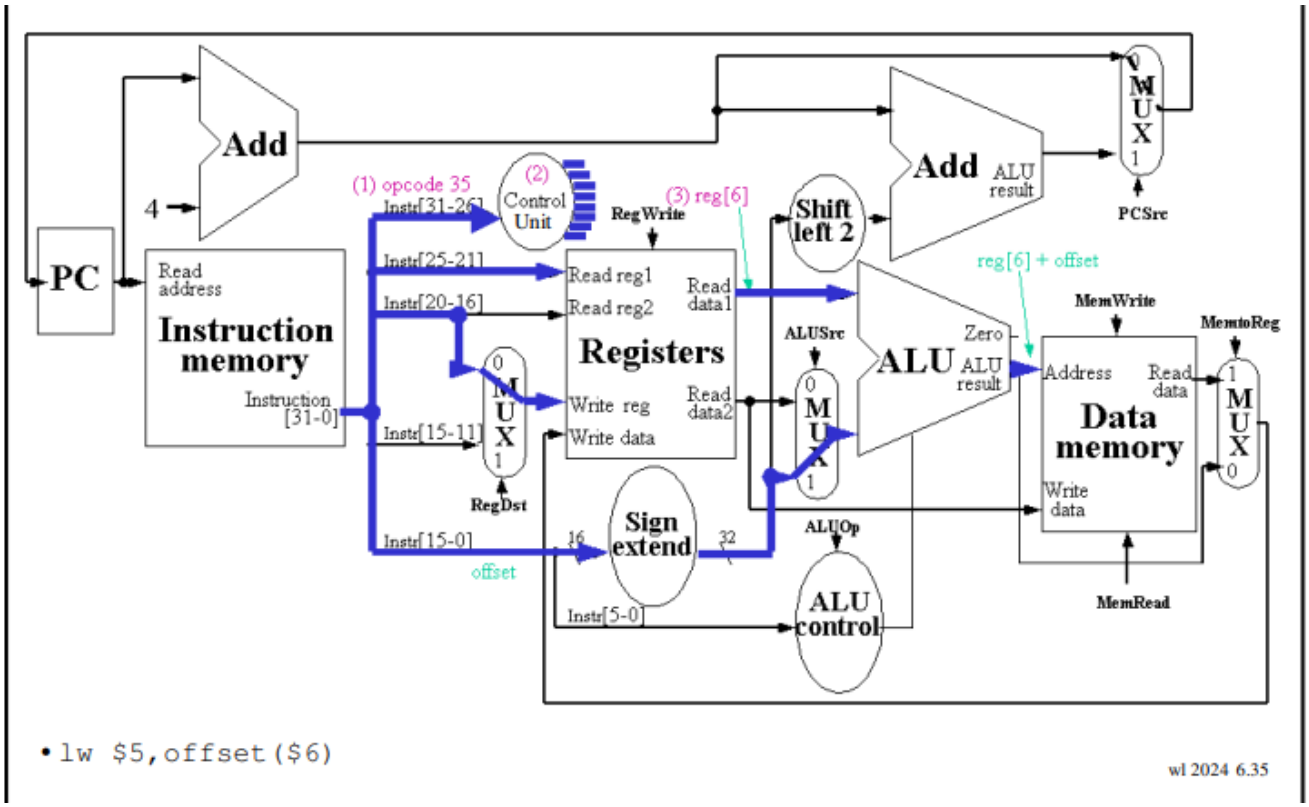


• lw $5,offset($6)

wl 2024 6.34

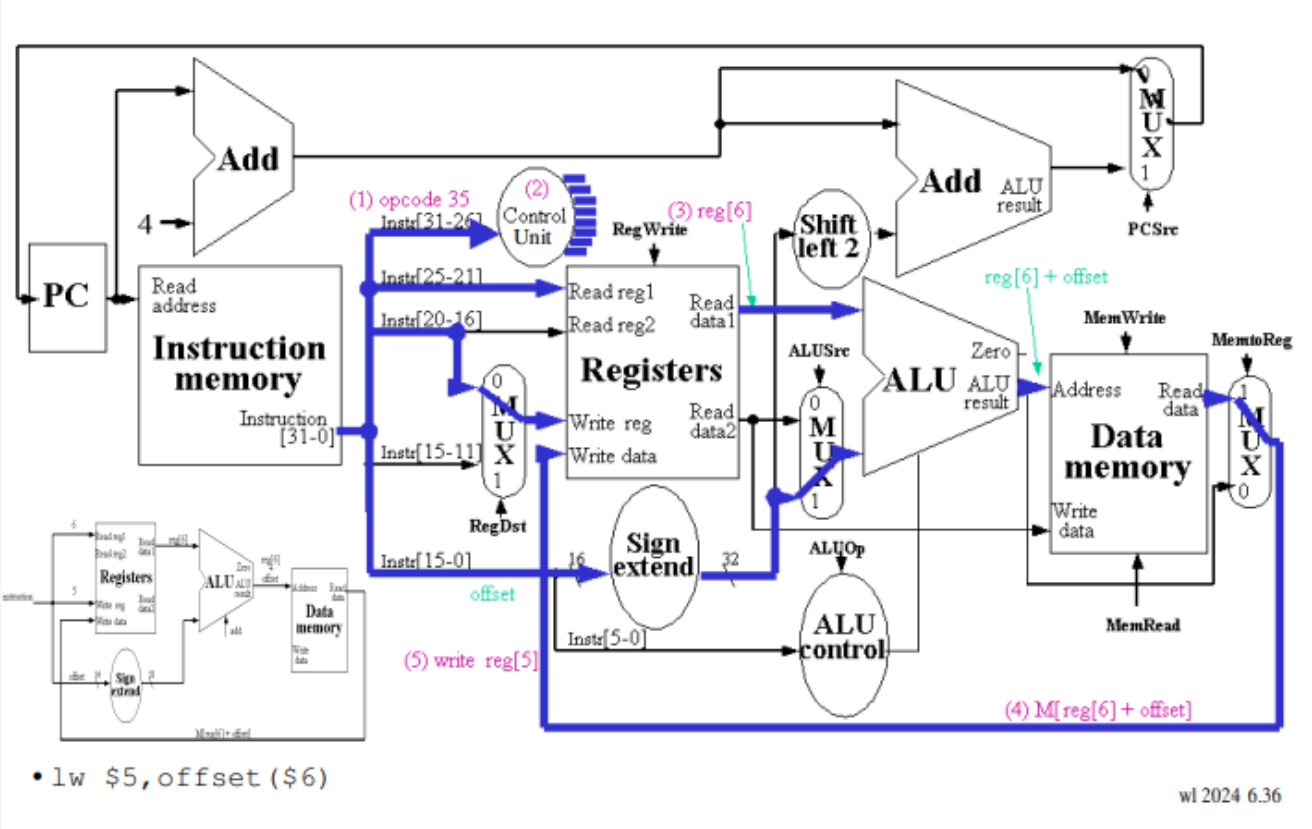
The Control Unit now produces the control signals for the multiplexors and the Registers etc. At the same time the appropriate fields from the operand of the instruction are sent to the relevant inputs of the Registers and the ALU.

Load instruction(3)



The ALU then produces the address for the Data Memory.

Load instruction ((4) and (5))



Finally the content of the Data Memory with the address from the ALU is sent to the Registers. We can check, once more, that the active connections of this combined datapath is the same as those of the Load datapath.

Summary