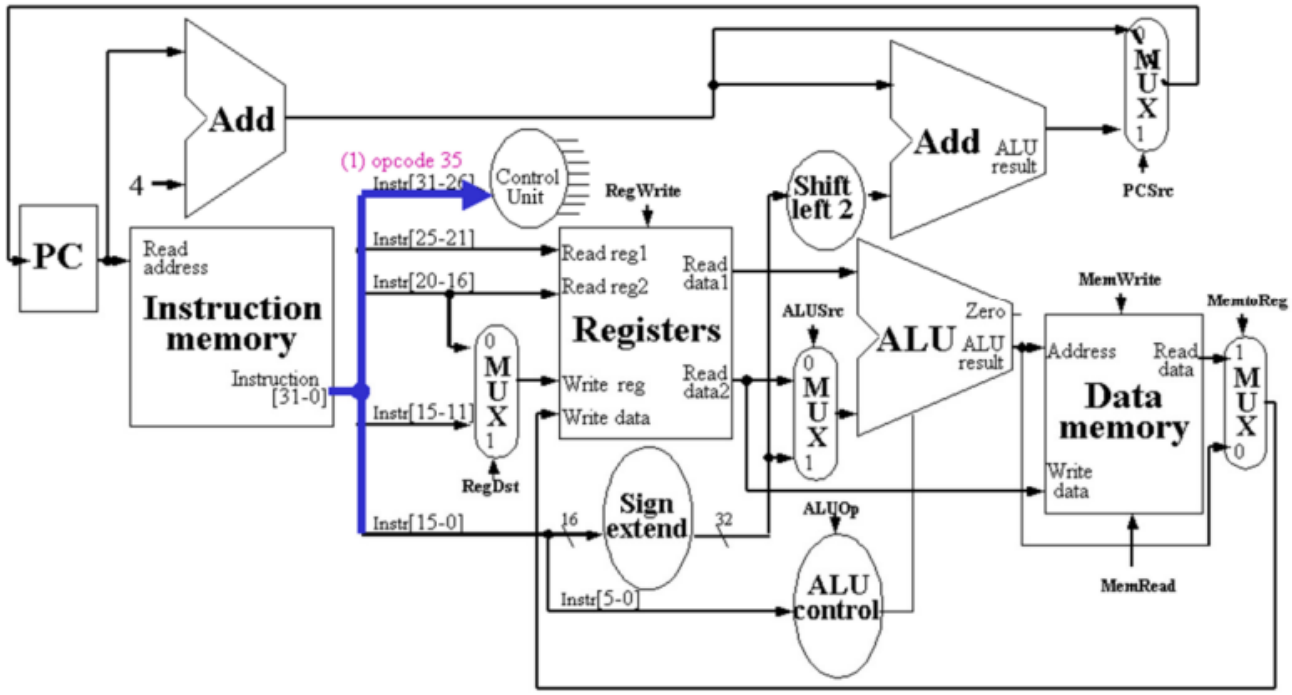
Lecture 7, Multi-Cycle Datapath

Single-cycle datapath: Load Instruction (1)

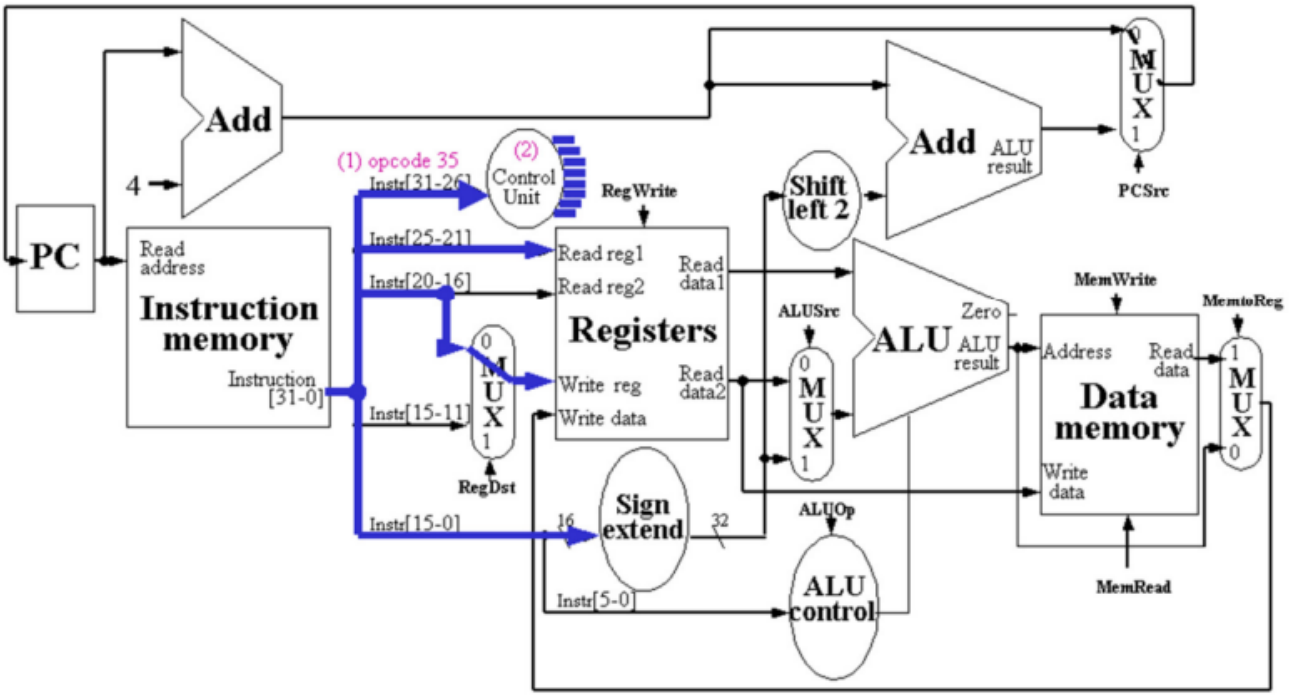


• lw $5,offset($6)

wl 2024 7.3

Let us first recap how the single-cycle datapath works when dealing with 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)

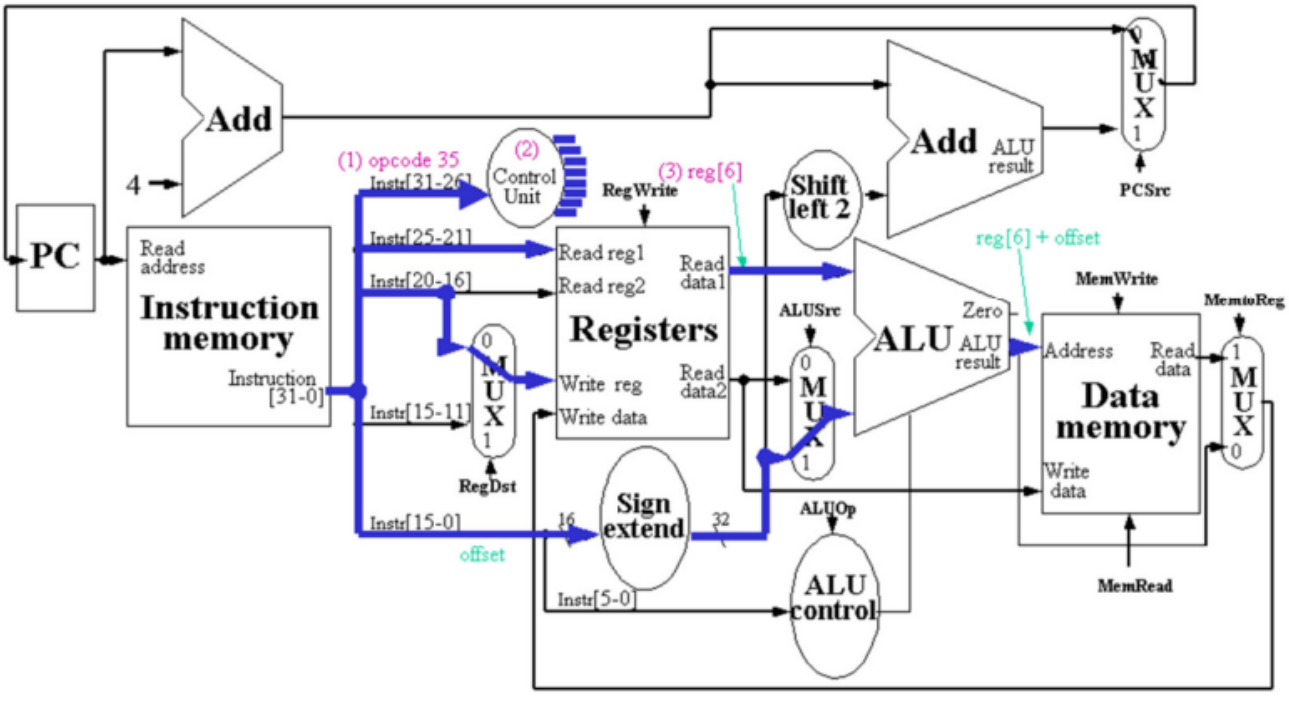


- lw $5,offset($6)

wl 2024 7.4

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)

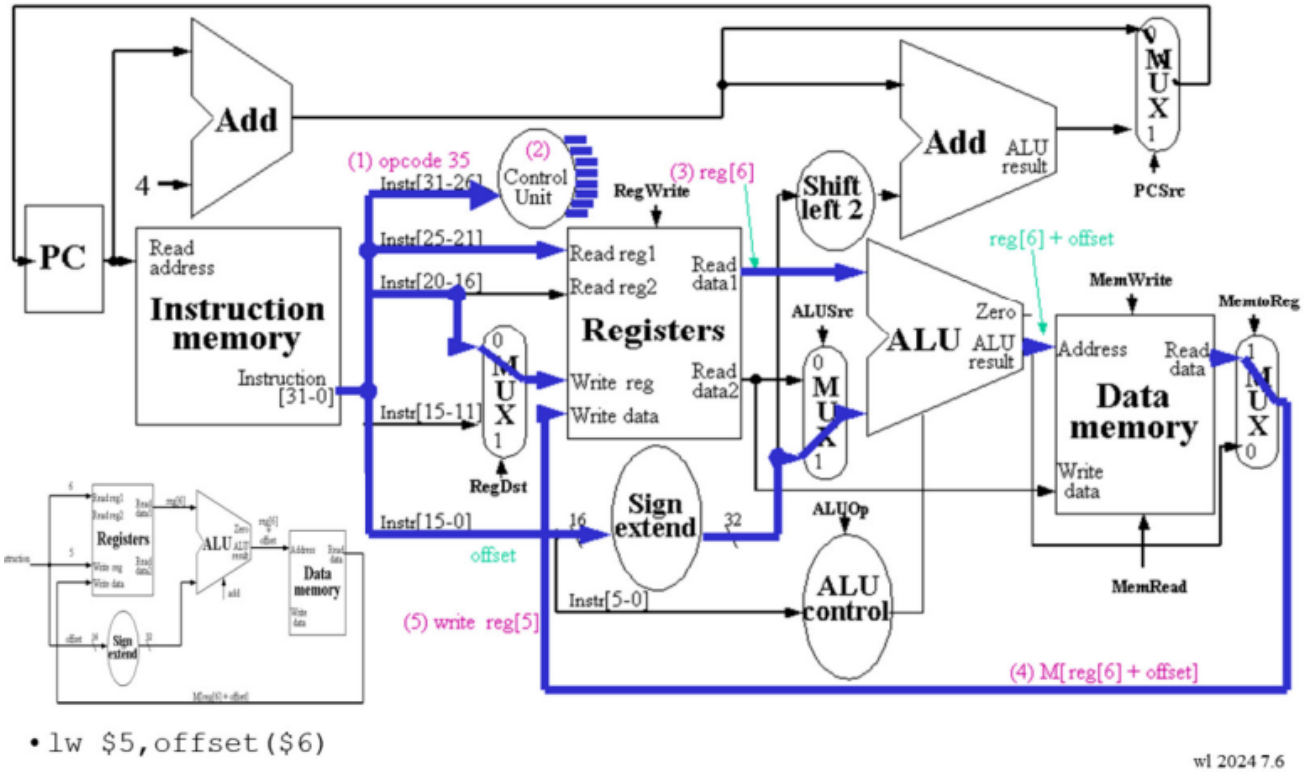


- lw $5,offset($6)

wl 2024 7.5

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.

Execution Cycle

	instruction fetch	read register	ALU operation	load/store data	write register
• R type	√	√	√		√
• load	√	√	√	√	√
• store	√	√	√	√	
• branch	√	√	√		
• jump	√				

This table shows that different instructions can involve different steps in the single-cycle datapath. The load instruction covers all 5 steps, while the jump instruction

involves just one step. Since all steps have to be completed within a clock cycle, this means that the cycle time is constrained by the slowest instruction – the one that involves the greatest number of steps. In particular, the execution of the jump instruction has to wait for a long time even if it could have finished long ago. Could different instructions take different amounts of time to finish, depending on the number of steps they involve?

Multi-Cycle Datapath

- improve single-cycle datapath
  - speed: each cycle covers only one stage of execution
  - resource usage: use ALU differently in different cycles
- multiple cycles per instruction
  (R-type: 4 cycles, load: 5 cycles)
- single memory: instruction and data
- reuse ALU for increment PC, but need instruction register IR
- faster, reuse hardware in different cycles
- more registers to save state, more complex control

wl 2024 7.8

The multi-cycle datapath allows different instructions to take different number of cycles to finish. It also allows reuse of resources like the ALU, at the expense of more complex control.

Building the Multi-Cycle Datapath

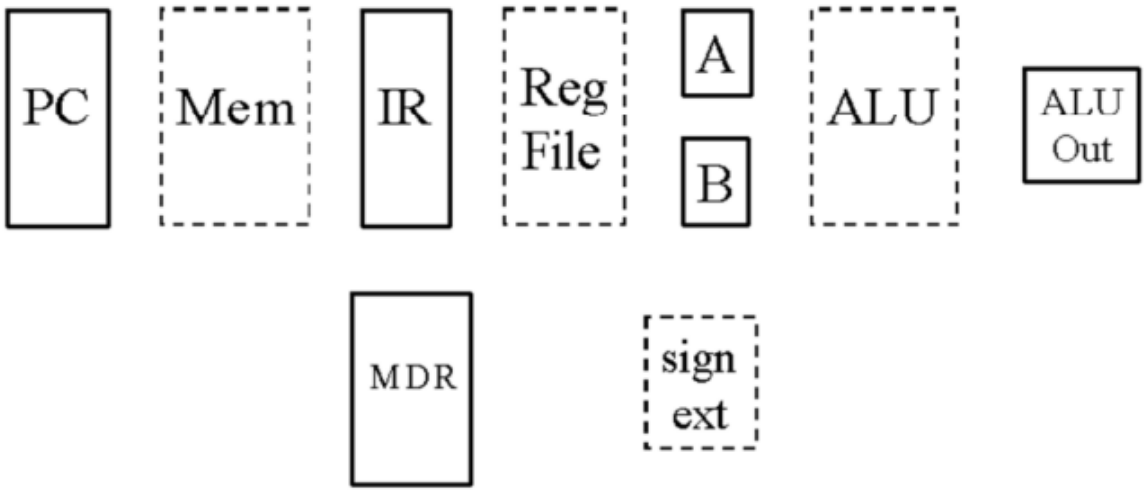
- basic components: memory, instruction register, register file, ALU, their input/output and control signals
- new internal registers: IR, A, B, ALUOut, MDR
- from instruction format, derive connections and multiplexors (mux)
  (1 mux for memory, 2 mux for register files and for ALU)
- feedbacks: ALU result to PC, ALUOut to memory address and register data, B to memory data
- next instruction logic: multiplexor to select next instruction, branch target or jump address, PCLoad = PCWrite or (PCWriteCond and ALUZero)

Let us build the multi-cycle datapath from scratch, by considering the key components needed for executing an instruction. There will be several new internal registers to store values during particular cycles when executing an instruction. We will derive the connections and multiplexors from the instruction format.

Additional Slides that he released later

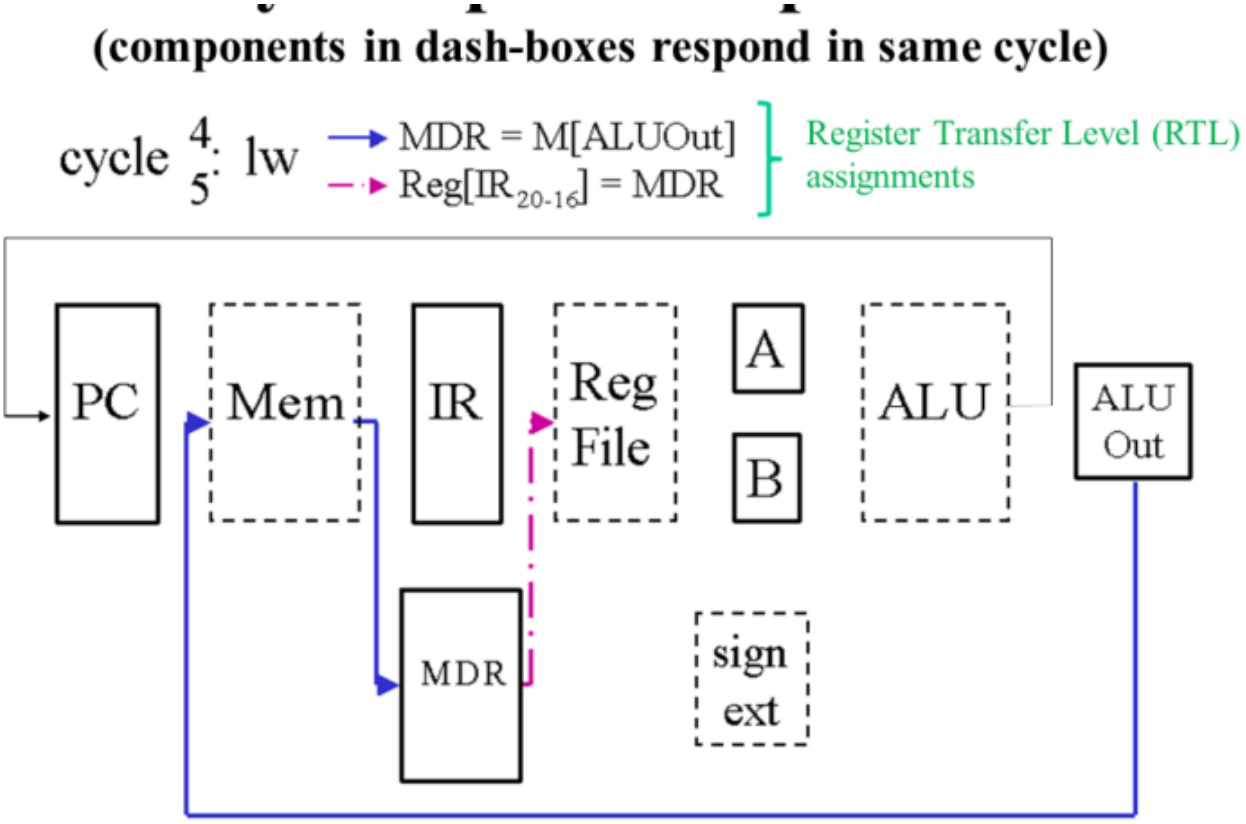
Key Datapath Components

(components in dash-boxes respond in same cycle)



To start from the simplest diagram, we shall just include the key datapath components and see what connections between them are required to execute an instruction. We would need 6 components, PC, IR, MDR (Memory Data Register), A, B, and `ALUOut`, which behave like registers, so we could think of them as a variable which can be assigned different values. Another 4 components, Mem, Reg File, sign ext, and ALU, would respond within the same cycle. We could think of IR as the output of the memory that stores instructions, while MDR as the output of the memory that stores data; and A and B as the outputs of the register file.

Key Datapath Components



As an example, lets see how the multi-cycle datapath works by considering the last two cycles in executing the load (lw) instruction. In cycle 4, the values from ALUOut will be used as the address for Mem and the content from Mem will be stored in MDR (see the blue connection). This behaviour can be described by the register transfer level (RTL) assignment: $MDR = M[ALUOut]$, which can be thought of as an assignment statement in a high-level language. Similarly in cycle 5, the value in MDR is stored in the register file specified by the 16th-20th bit in the instruction register IR (see the purple connection). So the RTL assignment is $Reg[IR_{20-16}] = MDR$.

Load Instruction: RTL for each step

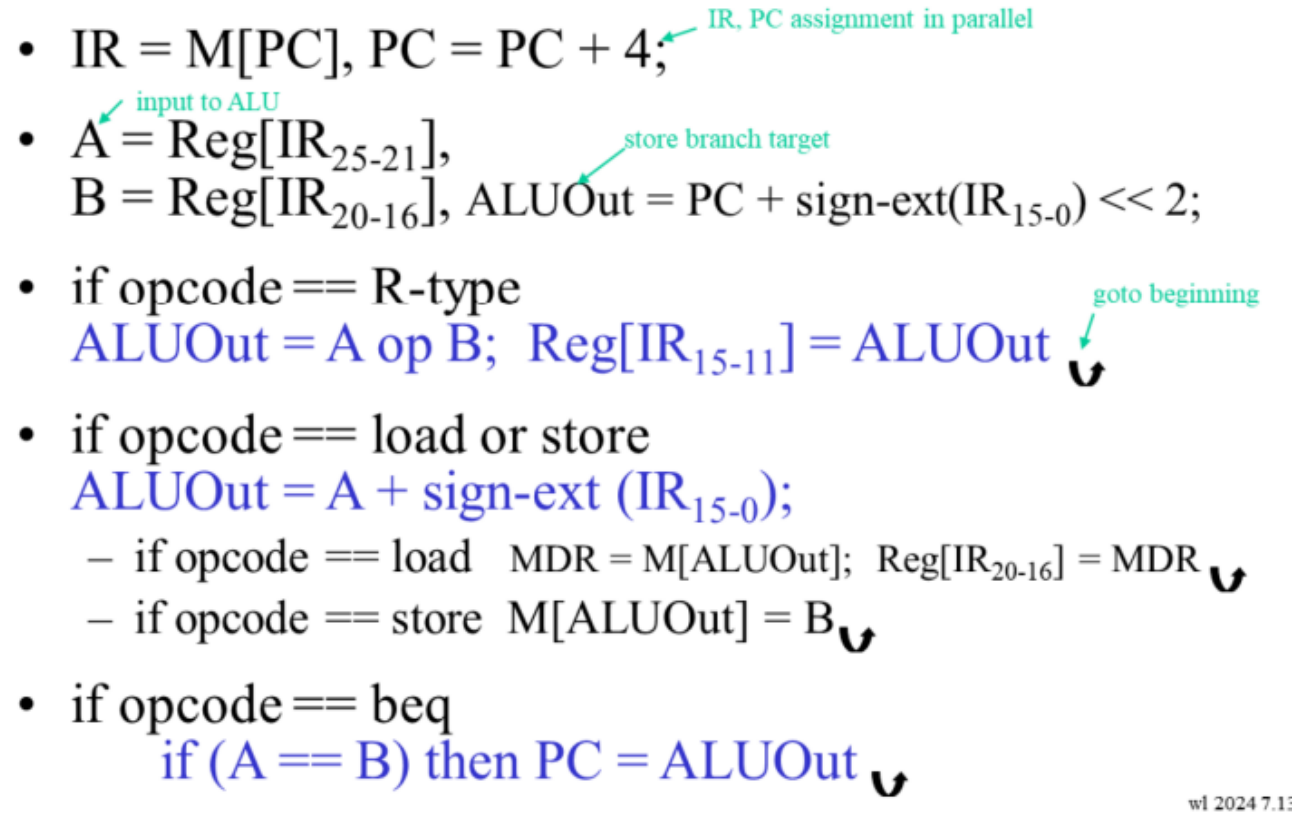
- effect: Reg[dest] = M[Reg[source] + sign-ext(addr)]
  dest = IR_{20-16} source = IR_{25-21} addr = IR_{15-0}
- break down the overall assignment into small steps
  - 1 small step (e.g. add or memory access) each cycle
  - need *local variables*: A, ALUOut, MDR such that
    A = Reg[source], ALUOut = A + sign-ext(addr), etc
- cycle 1: IR = M[PC], PC = PC +4
  cycle 2: A = Reg[source]
  cycle 3: ALUOut = A + sign-ext(addr)
  cycle 4: MDR = M[ALUOut]
  cycle 5: Reg[dest] = MDR

IR, A, ALUOut, MDR are new *internal registers* implementing the *local variables* for the multi-cycle datapath

We can derive the RTL assignments for each cycle by considering the effect of each instruction and how it can be broken down into multiple small steps, each step involving some local variables such as A, B, ALUOut, MDR etc which are provided by the internal registers. These local variables would have specific functions as specified by an RTL assignment: e.g. $A = Reg[source]$.

So we could see that within the 5 cycles, the effect $Reg[dest] = M[Reg[source] + sign-ext(addr)]$ is achieved by tracing the RTL assignments in each step, starting from cycle 5. From cycle 5, $Reg[dest] = MDR$. Then from cycle 4, $MDR = M[ALUOut]$, so $Reg[dest] = MDR = M[ALUOut]$. Then from cycle 3, $ALUOut = A + sign-ext(addr)$, so $Reg[dest] = M[ALUOut] = M[A + sign-ext(addr)]$. Then from cycle 2, $A = Reg[source]$, so $Reg[dest] = M[A + sign-ext(addr)] = M[Reg[source] + sign-ext(addr)]$, which is the intended effect.

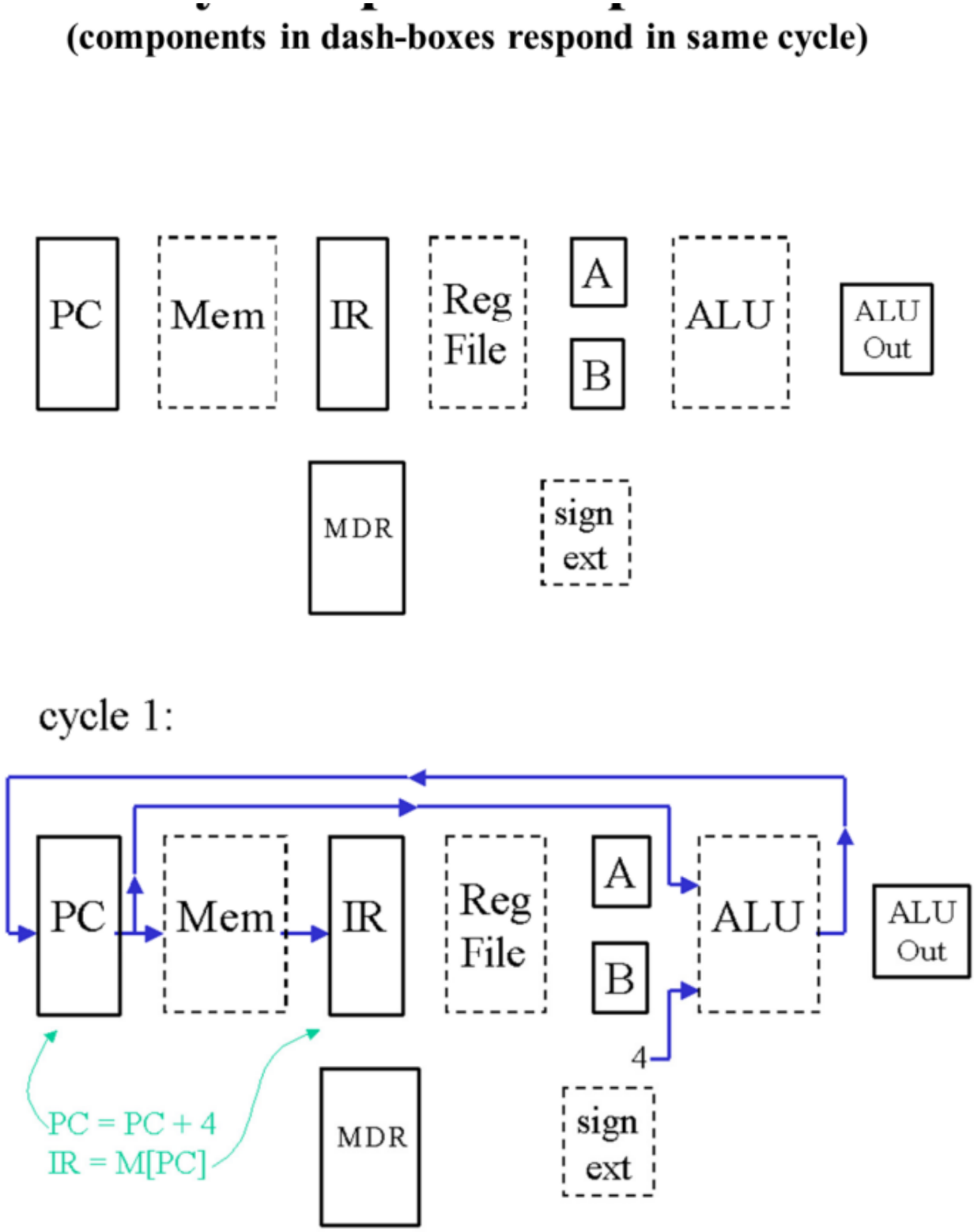
Execution Steps: Program Format



- IR = M[PC], PC = PC + 4;
- A = Reg[IR_{25-21}],
  B = Reg[IR_{20-16}], ALUOut = PC + sign-ext(IR_{15-0}) << 2;
- if opcode == R-type
  ALUOut = A op B; Reg[IR_{15-11}] = ALUOut
- if opcode == load or store
  ALUOut = A + sign-ext (IR_{15-0});
  - if opcode == load MDR = M[ALUOut]; Reg[IR_{20-16}] = MDR
  - if opcode == store M[ALUOut] = B
- if opcode == beq
  if (A == B) then PC = ALUOut

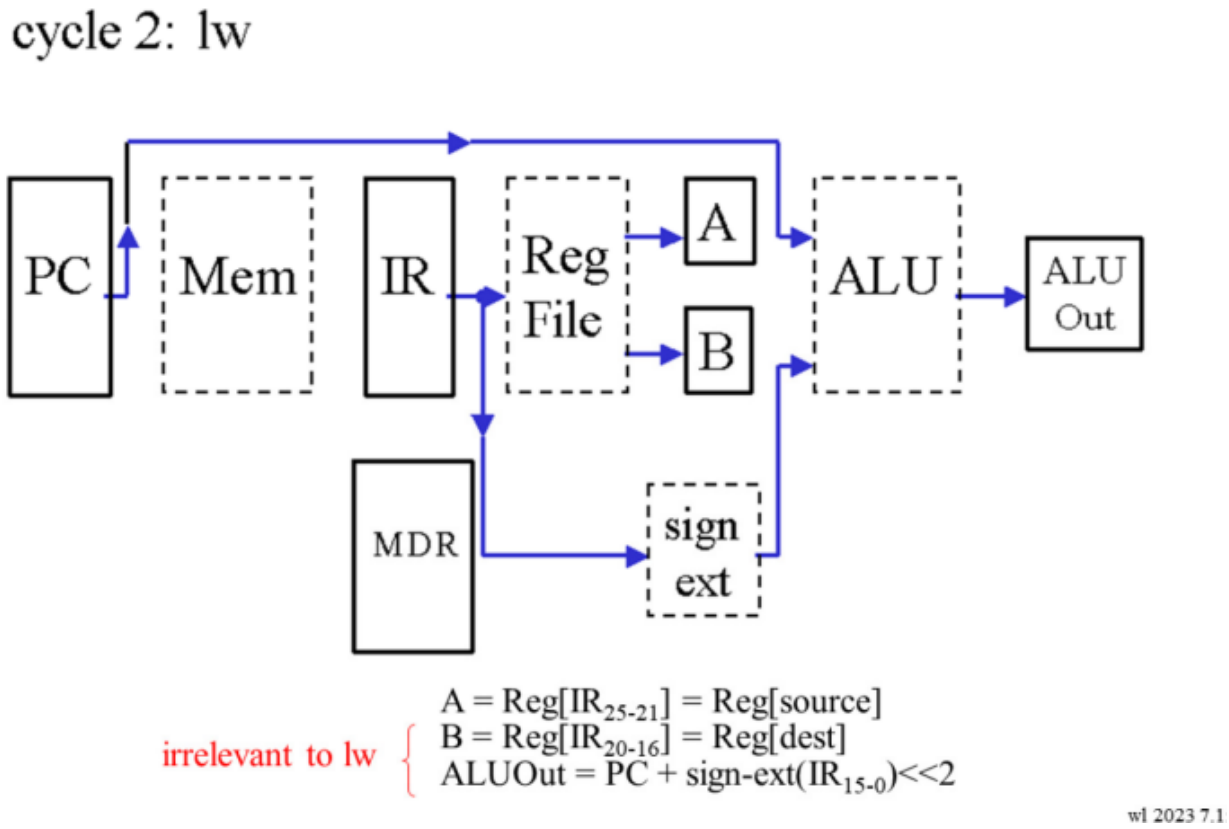
We just derive the execution steps for the load instruction. After deriving the execution steps for the other instructions, we get this summary of execution steps for multiple instructions in a program format. The first two bullet points correspond to the first two cycles common to all the instructions. From cycle 3, the execution steps will be instruction specific. For R-type instructions, two more steps would be involved; for load instructions, three more steps would be involved.

RTL Assignments (without control elements)



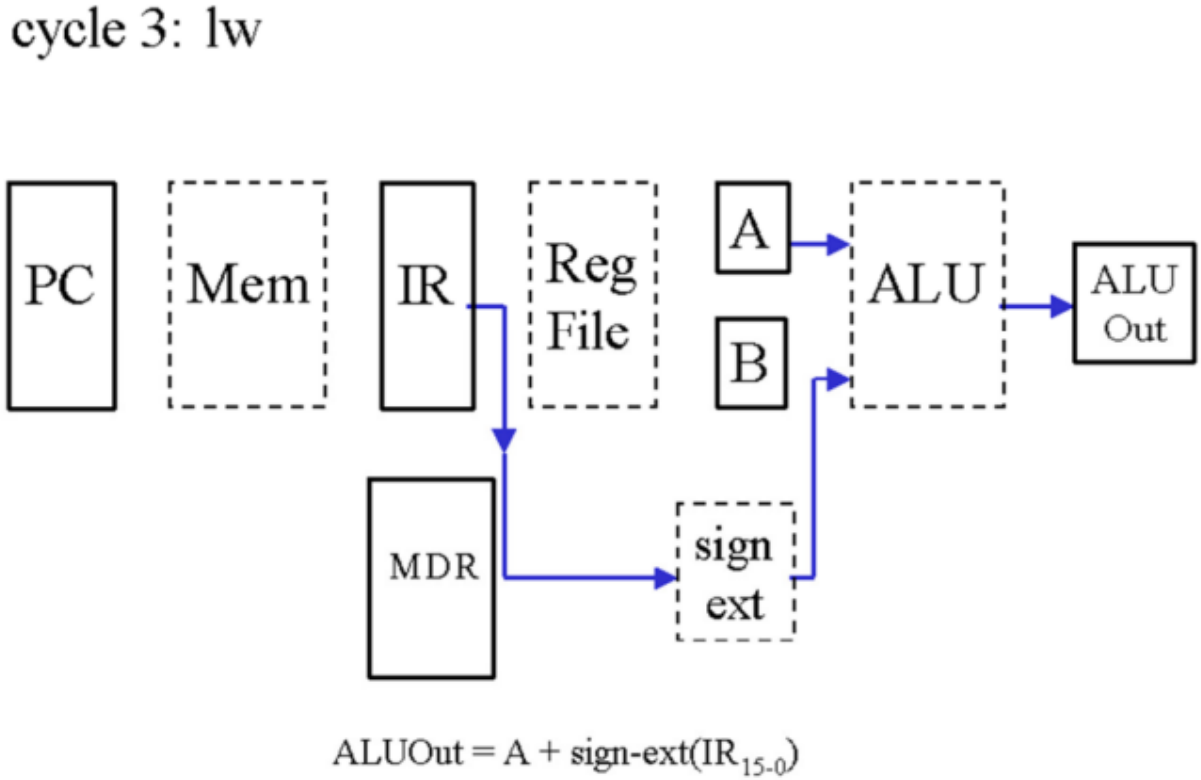
Lets see how these execution steps work in our simplest diagram without any control elements such as multiplexors. The first cycle always involves fetching the instruction from the memory Mem, and placing it in the instruction register IR: $IR = M[PC]$. At the same time, the program counter PC is incremented: $PC = PC + 4$.

RTL Assignments (without control elements)



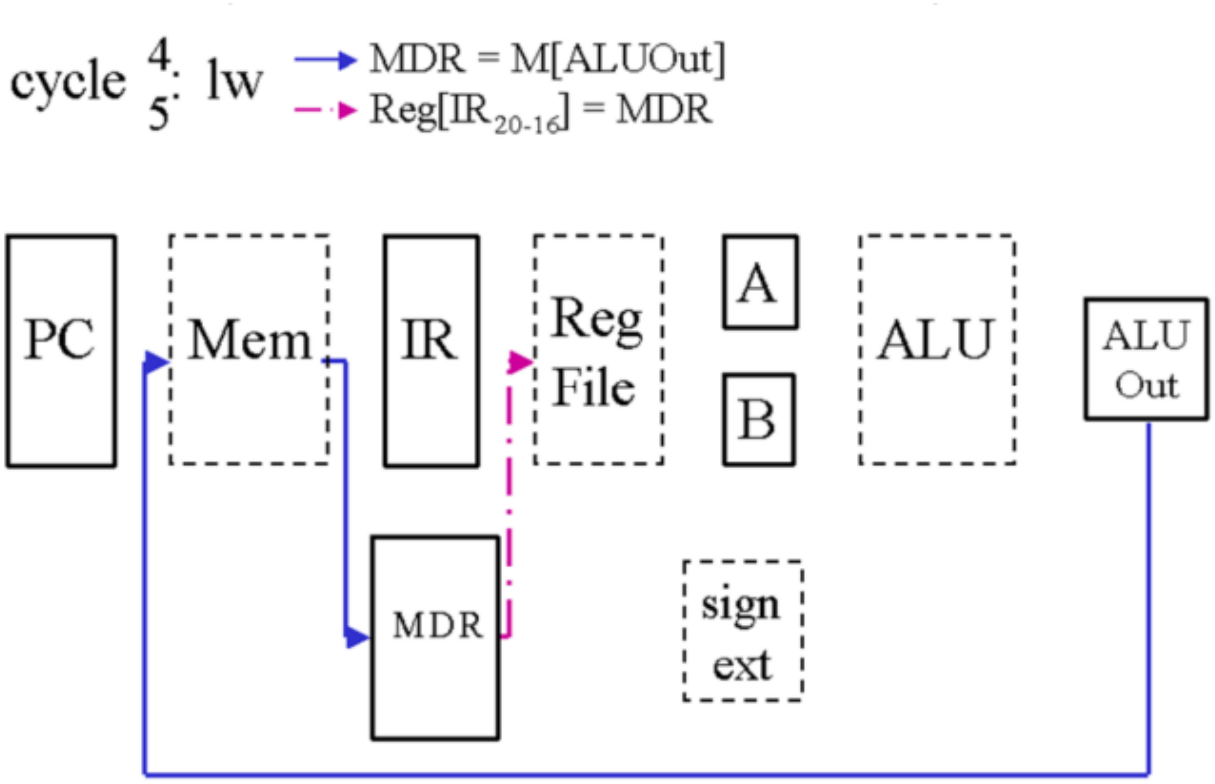
In cycle 2, there are three RTL assignments involving A, B and ALUOut. However, only the one involving A is relevant to executing the load instruction. The other two RTL assignments are useful for executing other instructions.

RTL Assignments (without control elements)



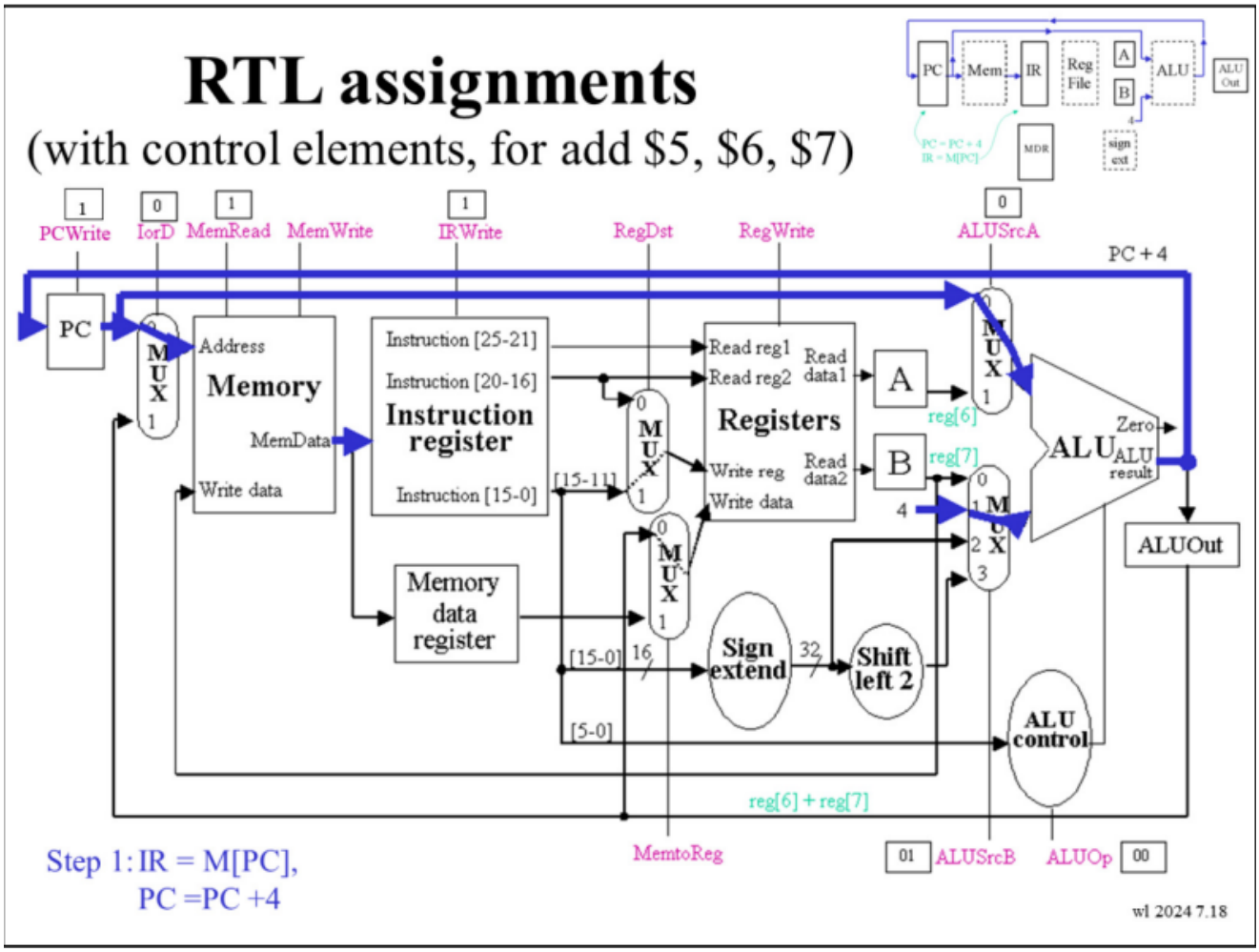
In cycle 3, the offset field from the instruction stored in IR is signextended and fed into the ALU which adds it to the value stored in A, and the result is stored in $ALUOut : ALUOut = A + sign - ext(IR15 - 0)$.

RTL Assignments (without control elements)



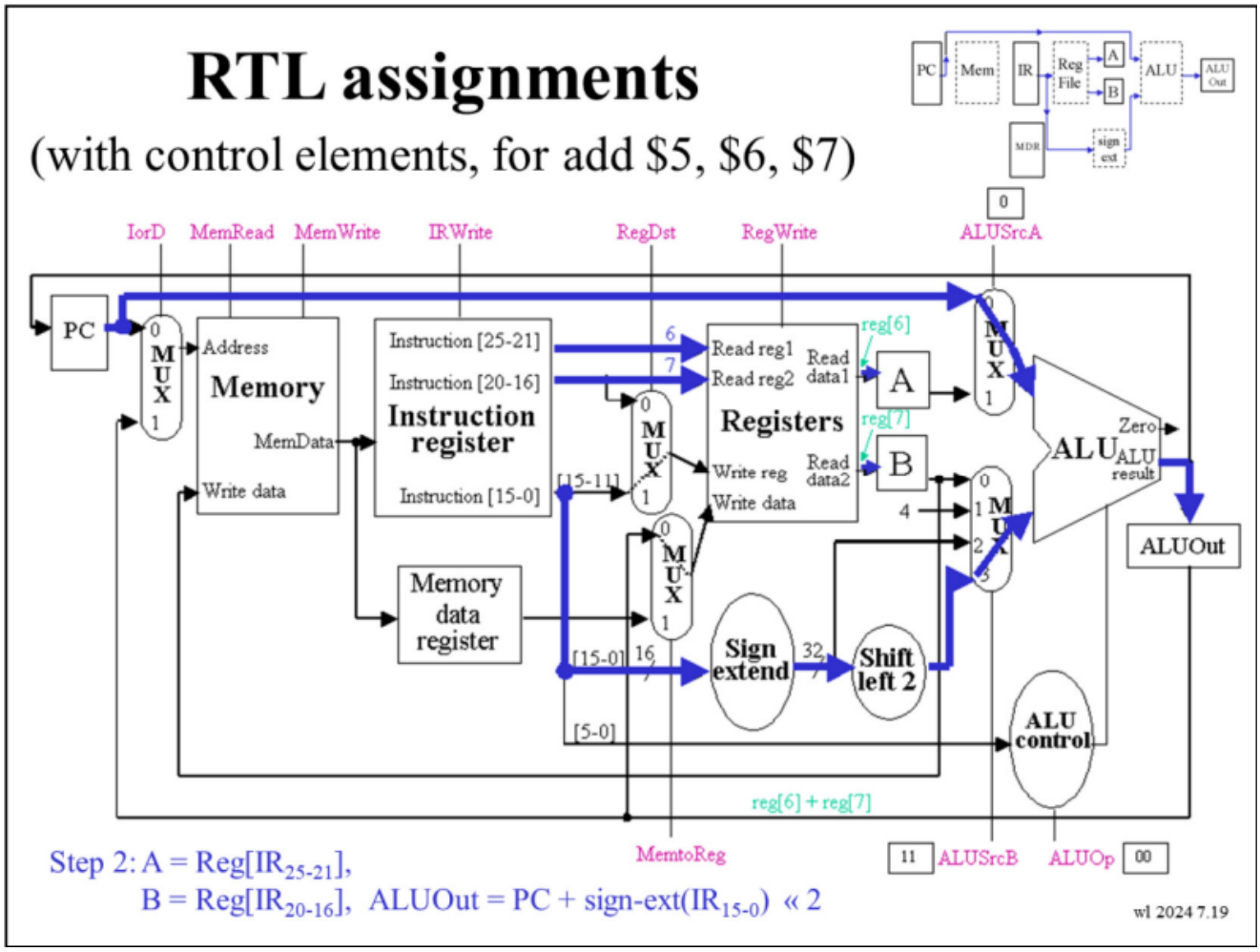
We have already explained what happens during cycle 4 and cycle 5, so the following is just a repetition. In cycle 4, the values from ALUOut will be used as the address for Mem and the content from Mem will be stored in MDR (see the blue connection). This behaviour can be described by $MDR = M[ALUOut]$. Similarly in cycle 5, the value in MDR is stored in the register file specified by the 16th-20th bit in the instruction register IR (see the purple connection). So the RTL assignment is $Reg[IR_{20-16}] = MDR$.

RTL Assignments (with control elements, for `add $5, $6, $7`)



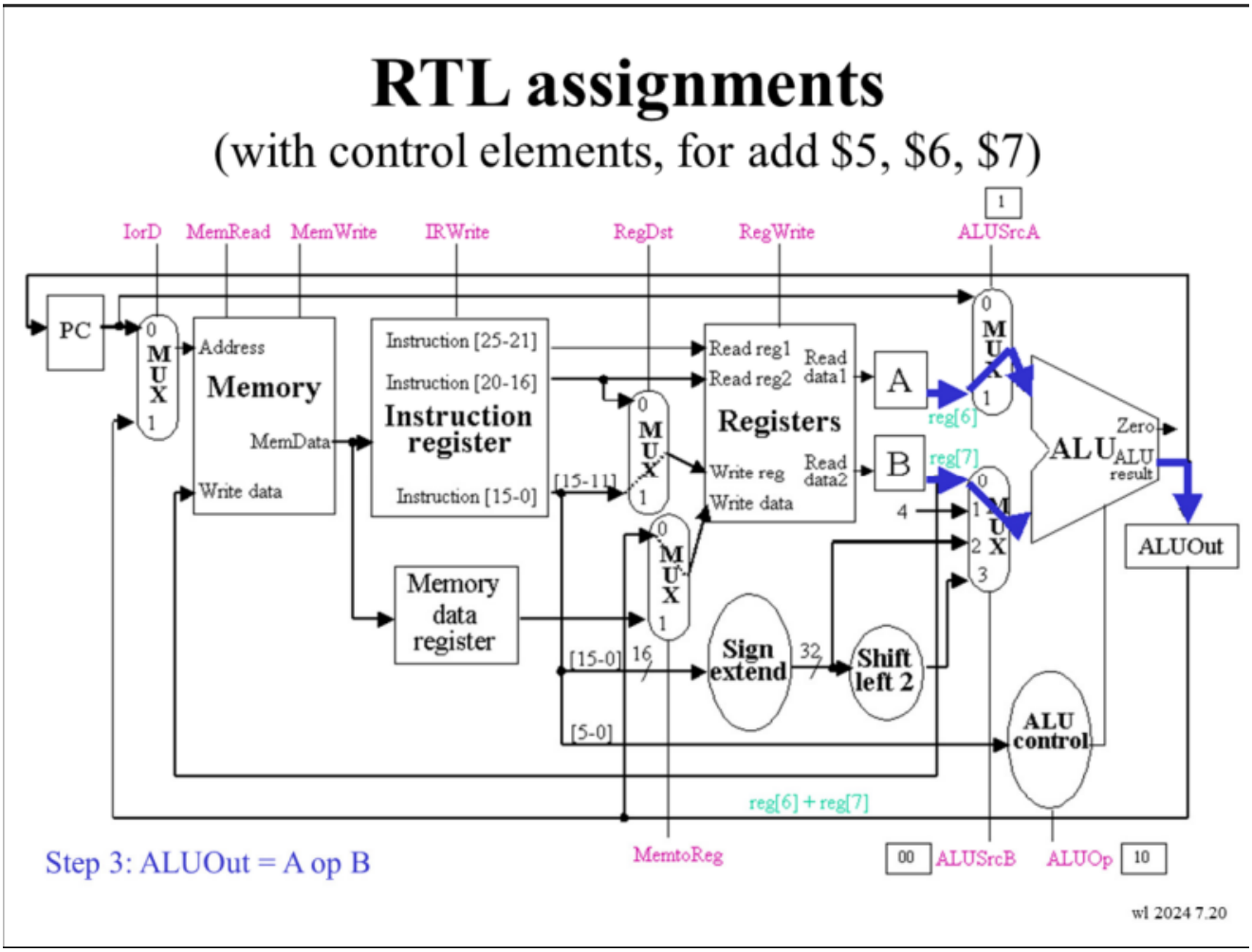
Now let us add the control elements such as multiplexors (but without the control unit – that would come later), and make explicit the control input to PC, IR, the register file etc. The highlighted connections show what happens during the first cycle (step 1) of executing the add instruction – how each multiplexor would need to be controlled to support a particular connection to make the execution work. The RTL assignments are the same as before; this just shows how the RTL assignments are implemented by the relevant control signals.

RTL Assignments (with control elements, for `add $5, $6, $7`)



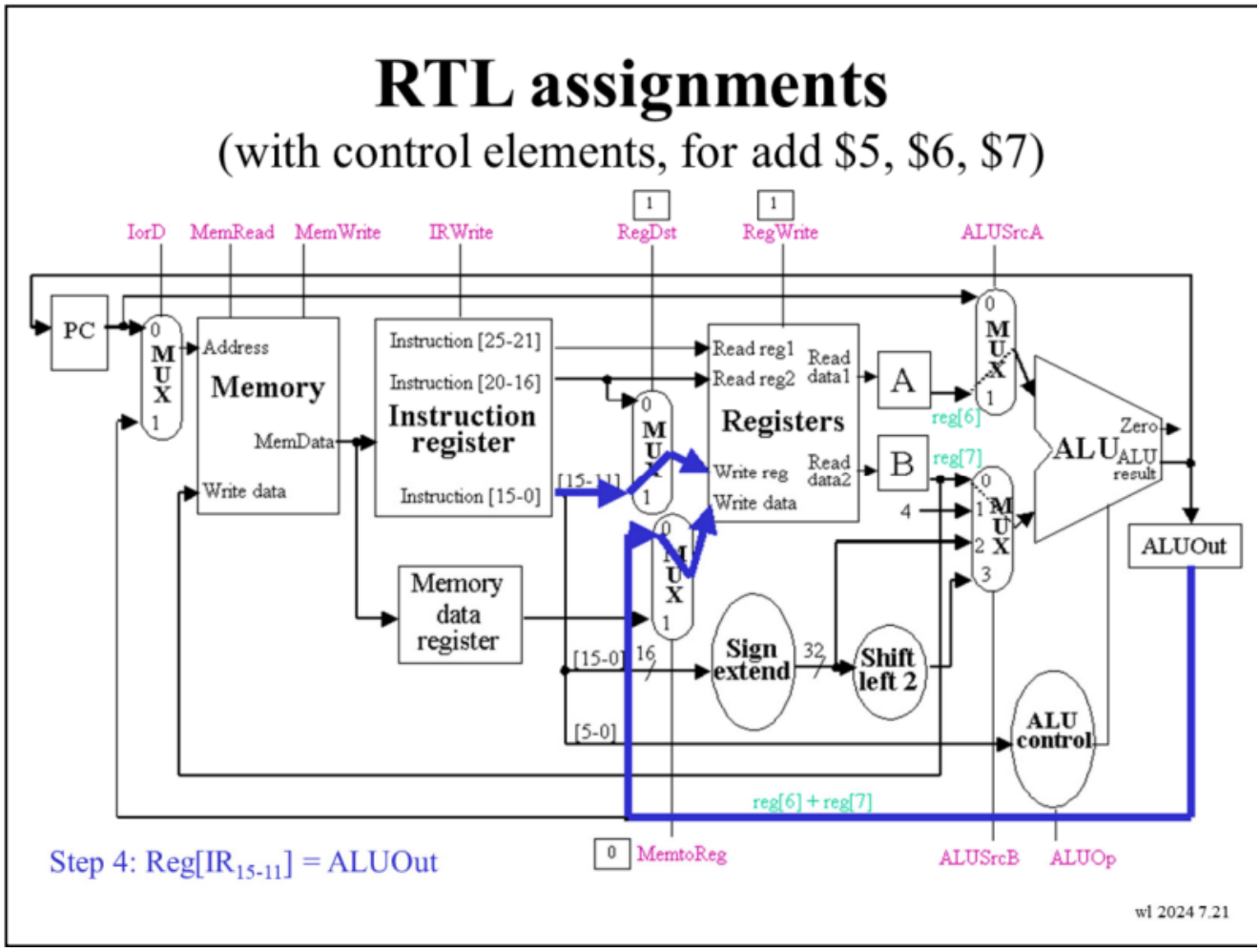
This shows step 2 with control elements. Compare this with the version without control elements – the point again is to show how the control elements and the control signals are used to implement the three RTL assignments.

RTL Assignments (with control elements, for `add $5, $6, $7`)

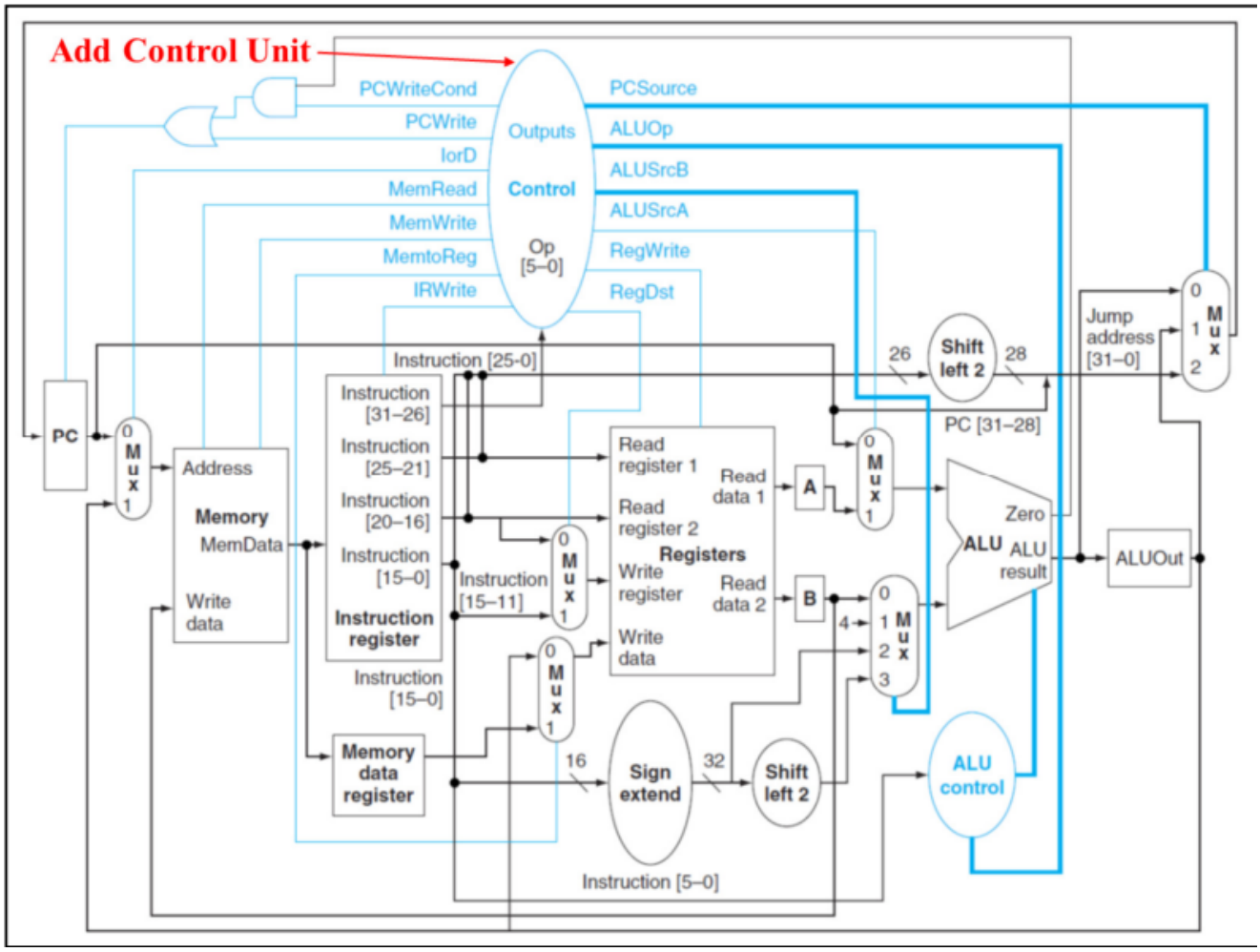


Step 3 is easy: just store the result from the ALU into ALUOut. The operation (A op B) is specified by the ALUcontrol block as explained in the last lecture.

RTL Assignments (with control elements, for `add $5, $6, $7`)



The final step, step 4, involves writing the value from ALUOut to the register specified by the destination field (bit 11-15) of the instruction register, IR_{15-11}



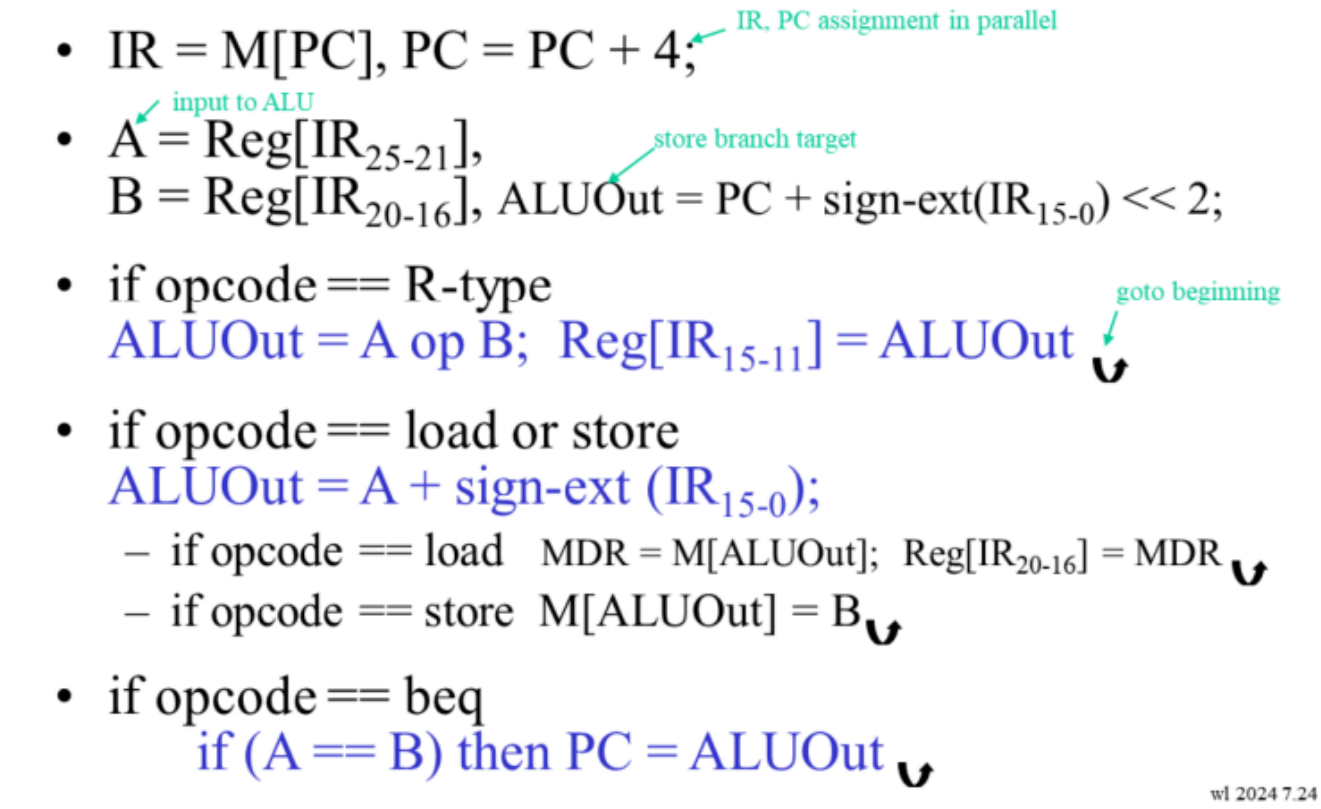
Now lets include the control unit in our diagram, since it is where the control signals come from. So now the diagram gets much more complex, but its function is as described before!

Multi-Cycle Datapath: Control Description

- overview: while (TRUE) {fetch / decode / execute}
- execution step summary: Register Transfer Level
- state diagram (this lecture):
  specify execution steps / control signals in each cycle
- microprogram (next lecture):
  another way of specifying control signals
  (P&H Appendix C: from course homepage)
- datapath without control signals:
  less clutter, but does not deal with branch or jump instructions

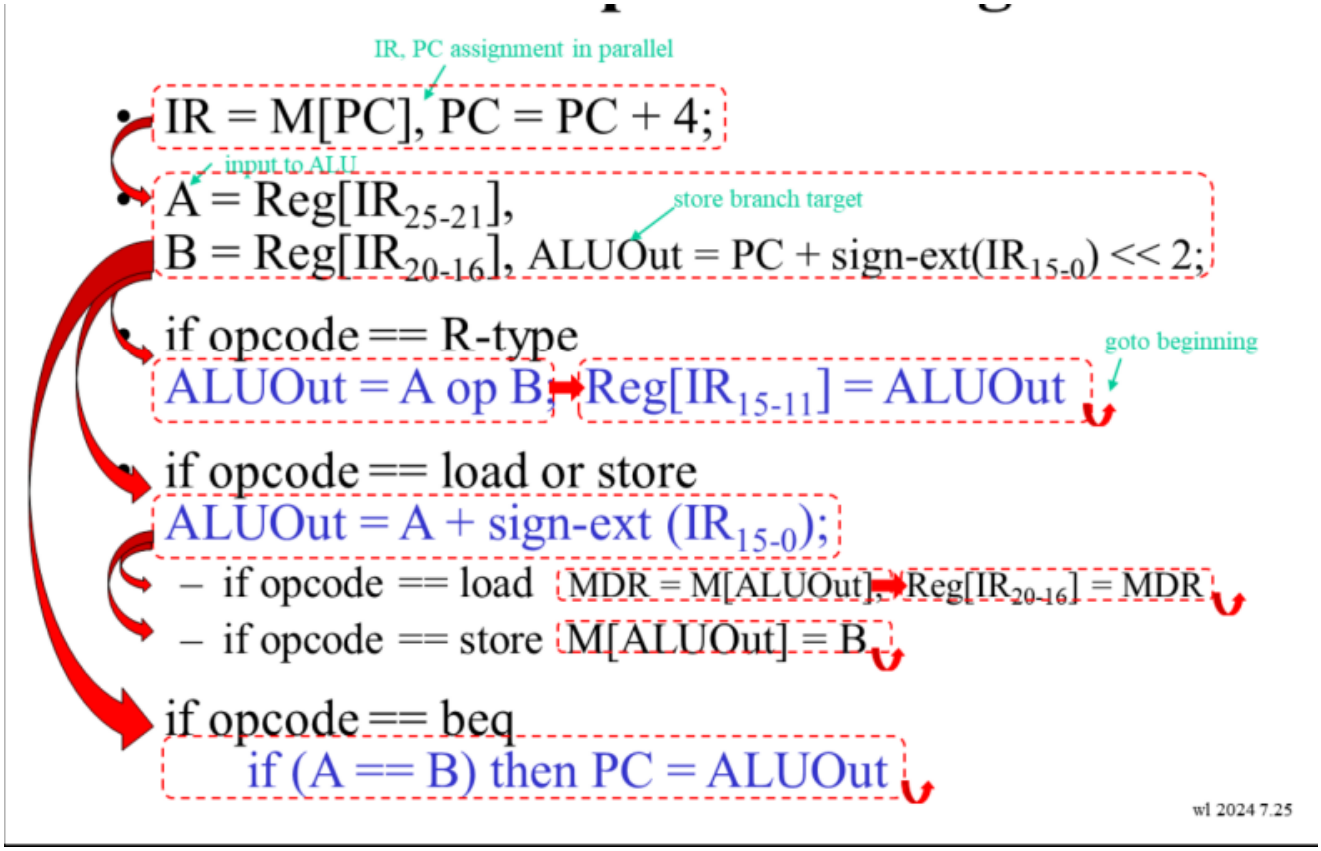
Lets consider how control can be described, and how we can derive the logic for the control unit. RTL assignment is the high-level description, capturing behaviour as specified by an assignment statement. To implement an RTL assignment, we need to make the control explicit. This will be achieved in a state diagram, in which control signals in each cycle can be specified. Lets do that step by step.

Execution Steps: Program Format



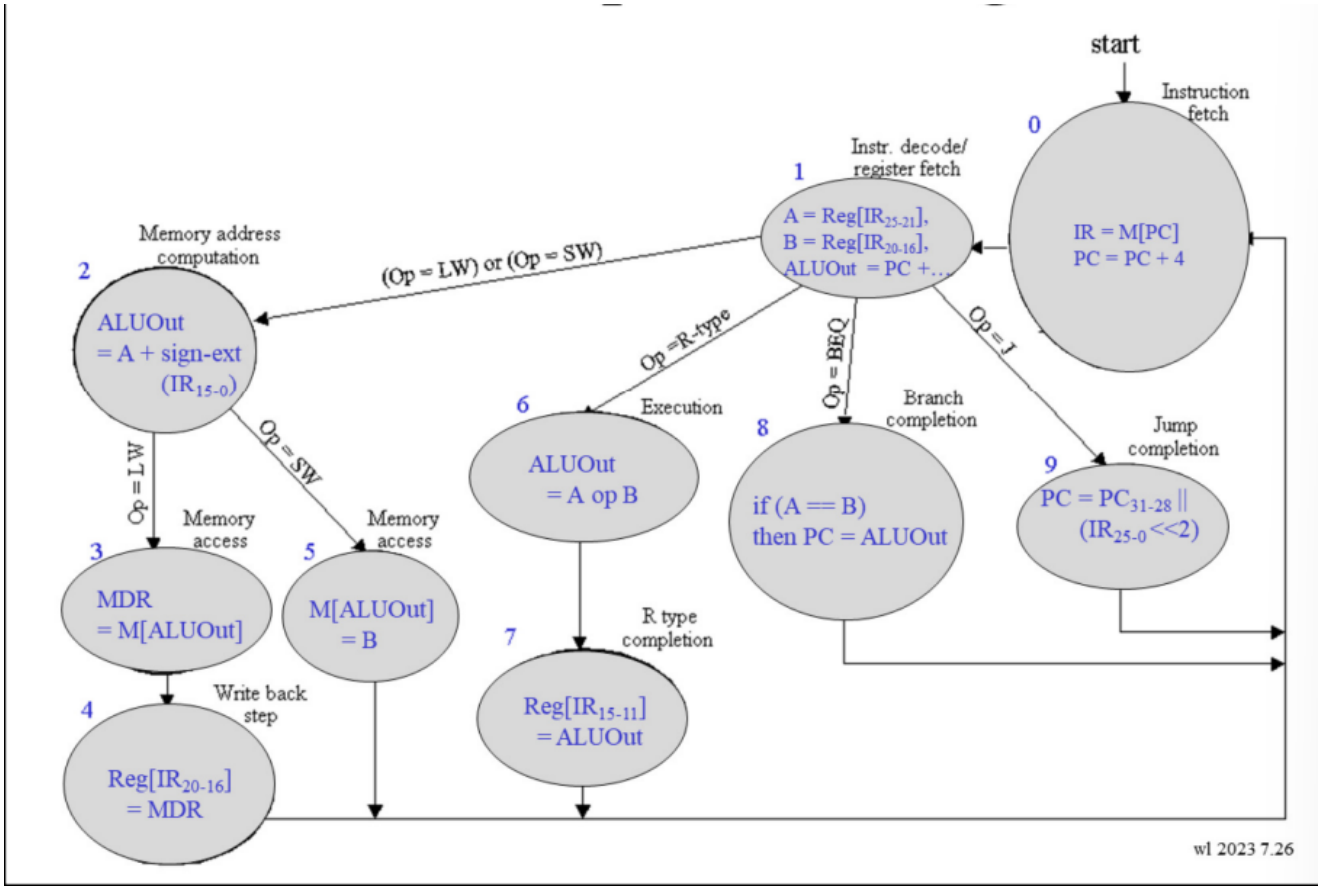
This is the program format describing the execution steps. It is clear, but does not make explicit the states where RTL assignments take place in each cycle.

Execution Steps: State Diagram



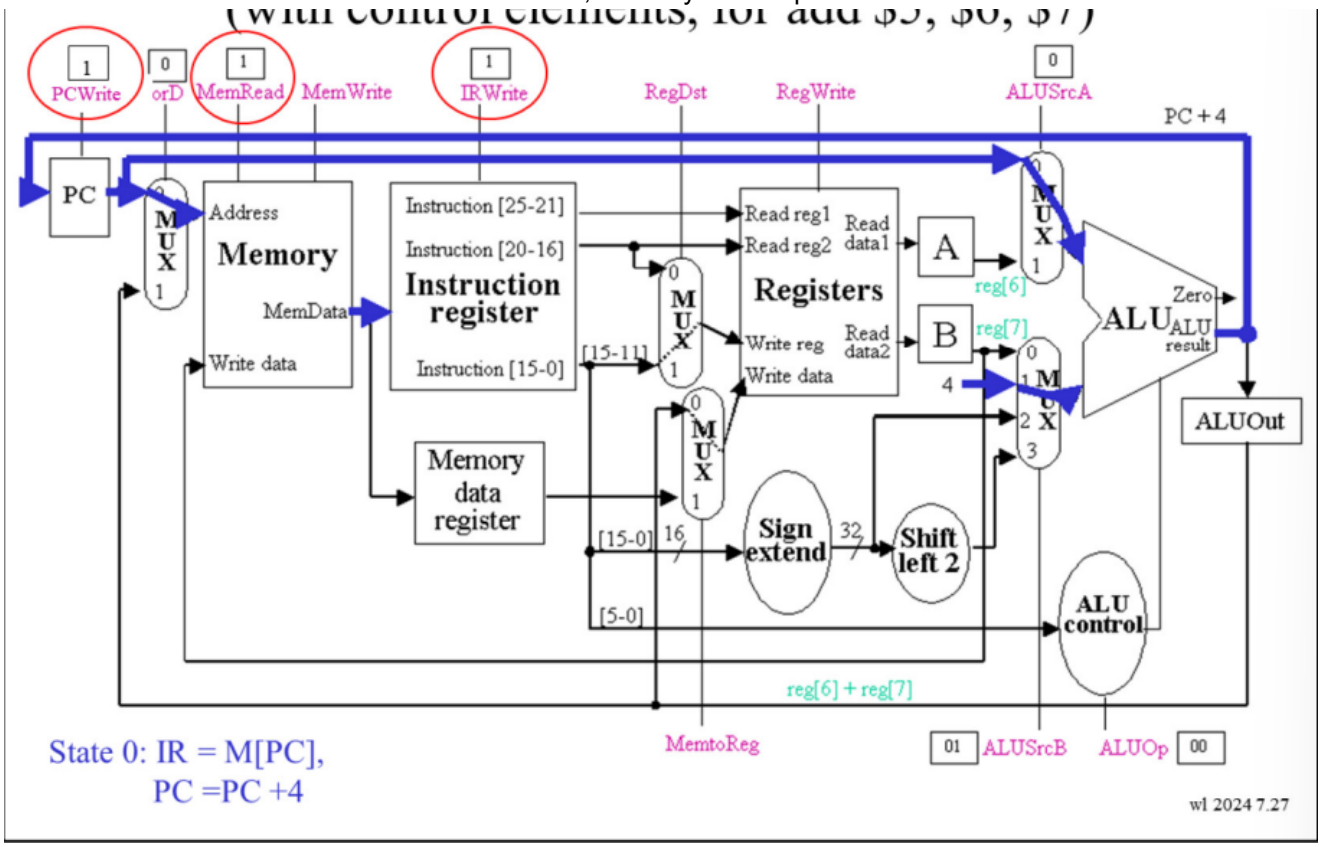
So we make explicit the RTL assignments in the same cycle by grouping them together to form a state, and add arrows to make explicit the flow of control as state transitions.

Execution Steps: State Diagram



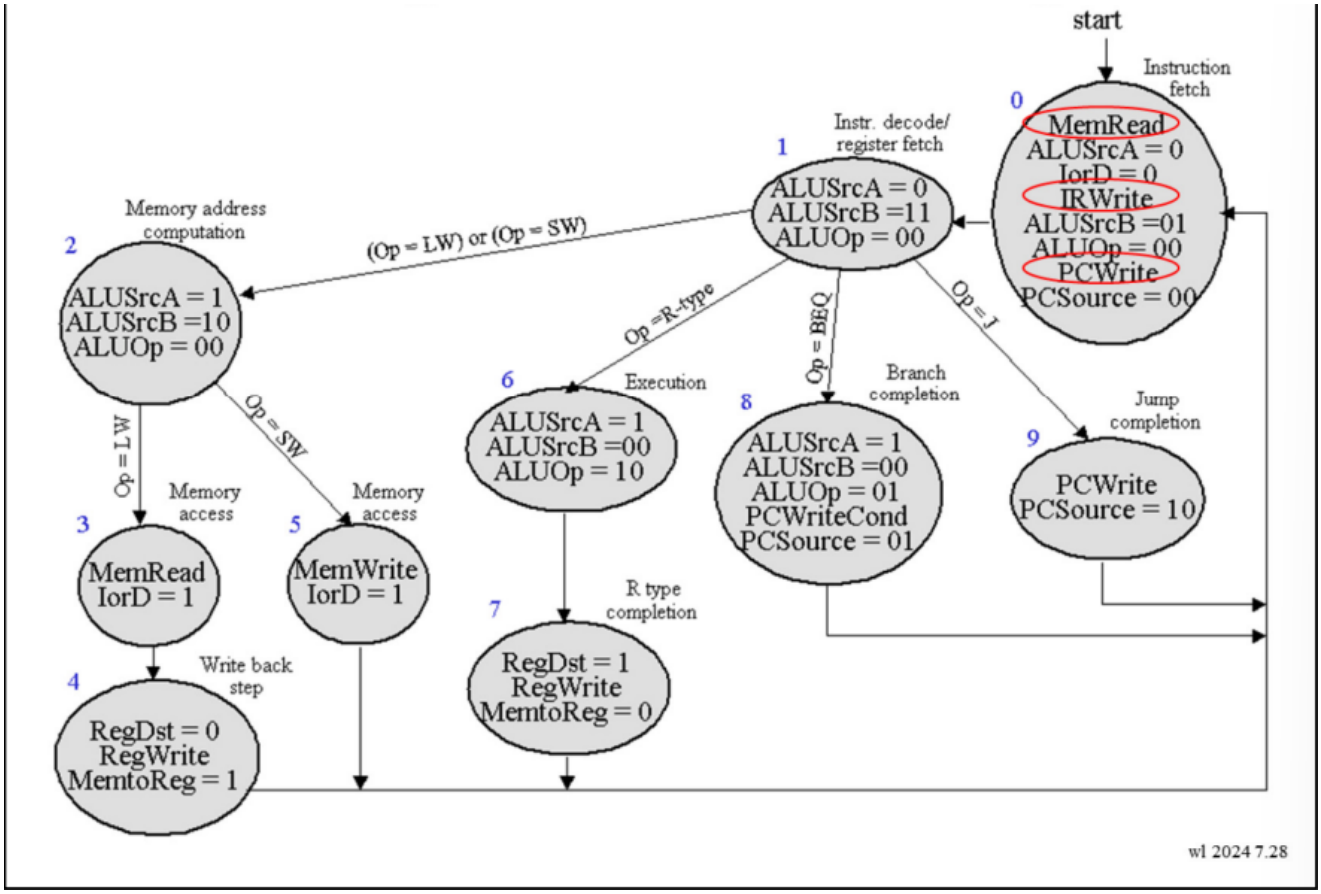
This presents a more conventional view of the same state diagram as shown previously, with the RTL assignments specified in each state. Next, we shall replace the RTL assignments by the corresponding control signal assignments.

RTL Assignments (with control elements, for `add $5, $6, $7`)



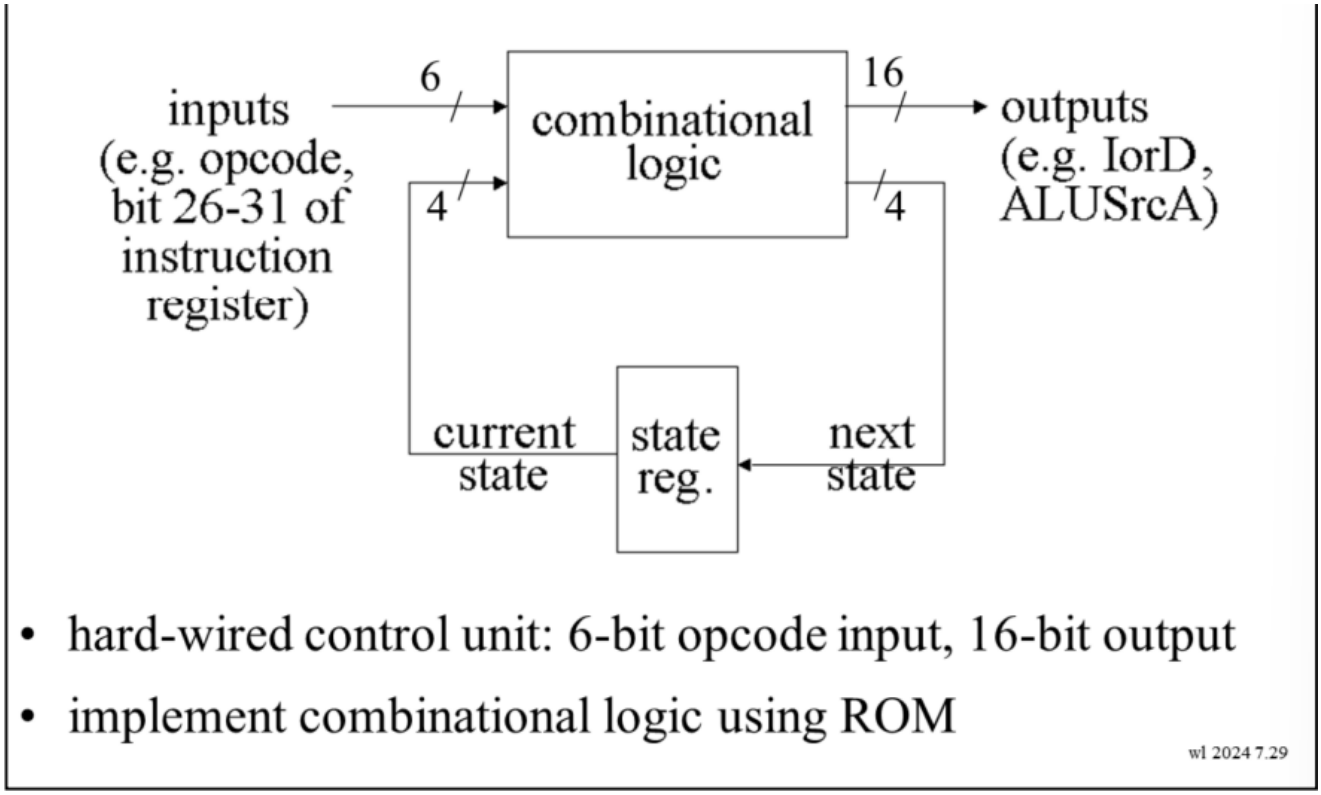
Lets study which control signals are used in supporting the RTL assignments in each state. In State 0, the relevant RTL assignments are implemented by asserting PCWrite, IRWrite, MemRead and so on.

State Diagram With Control Signals



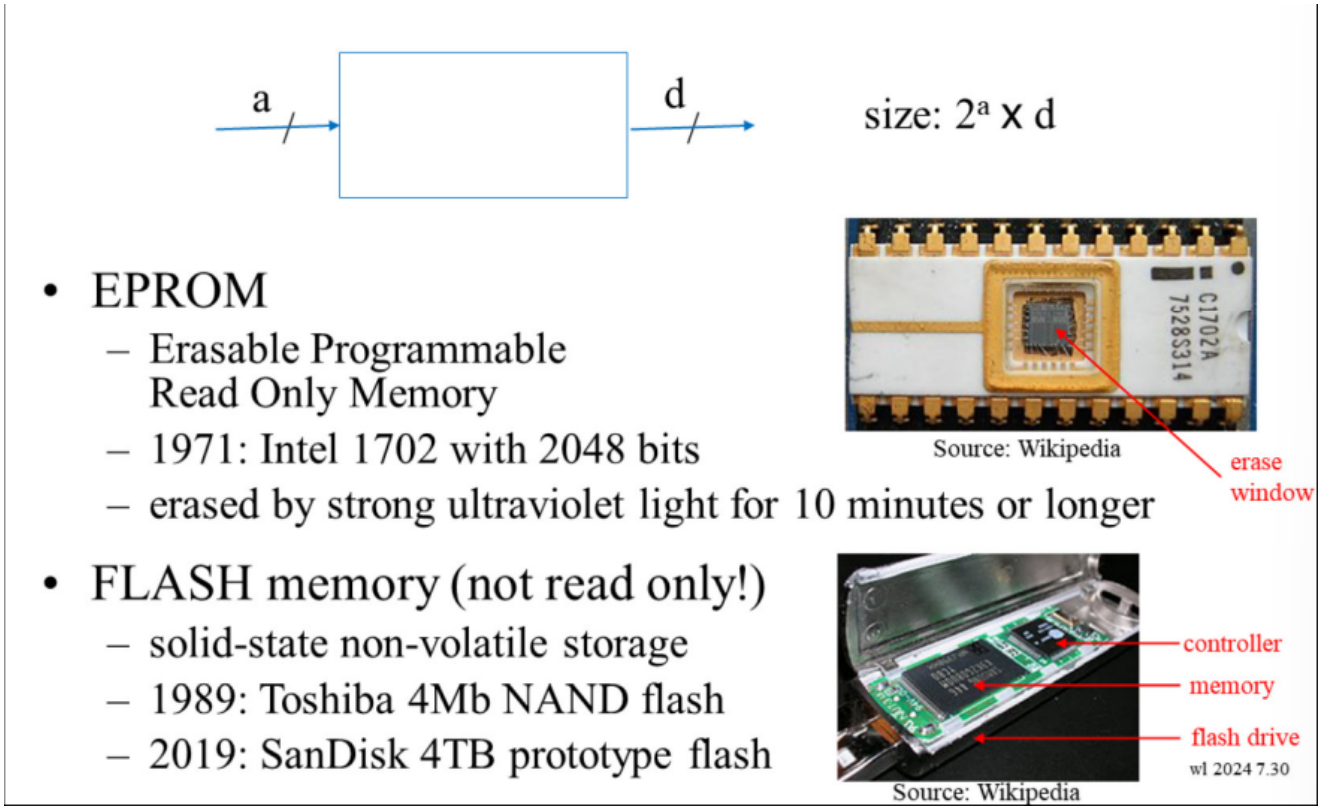
So instead of specifying the RTL assignments in each state, we specify the control signal assignments.

Finite-State Machine (Mealy)



How are the control signals generated? Use a finite state machine as the hard-wired control unit for the multi-cycle datapath. Since there are 10 states, we need 4 bits for the state register. The input to the control unit – this finite state machine – is the 6 bit opcode, the same as the input to the control unit for the single-cycle datapath. The control unit has 16 control signals as outputs – the multi-cycle datapath has more control signals than the single-cycle datapath.

ROM: Read Only Memory



The combinational logic in the control unit can be implemented by a ROM (Read Only Memory). This can be made from a library unit during the manufacture of the processor. You may have come across variants of ROMs as EPROM or FLASH. However, using a finite state machine with a ROM may not be the most efficient way of implementing the control unit. We will look at an alternative way in the next lecture