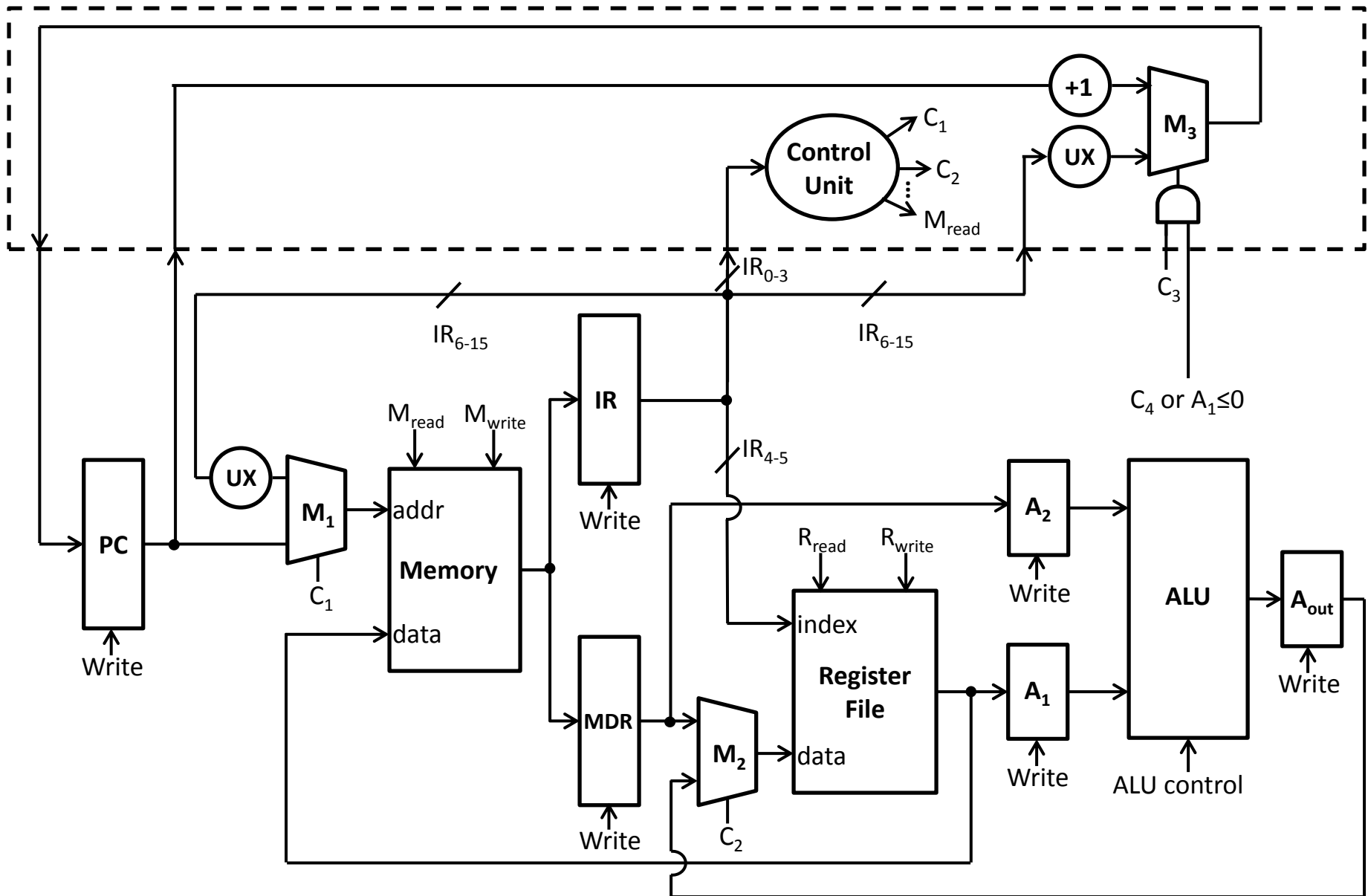


Understand CPU structure

- step by step
 - step 1: just main elements, no connections, no control
 - step 2: add control elements, e.g. multiplexers (mux)
 - step 3: derive control signals: this lecture
- Register Transfer Level (RTL) description: simple
 - 6 registers: PC, IR, A_1 , A_2 , A_{out} , MDR (memory data register)
 - 2 components respond in same cycle: UX, ALU, combinational
 - 2 components have Read and Write signals: Memory, Register file
- circuit diagram (see notes on *basic logic design*)
 - with or without control elements (e.g. mux)
 - with or without control unit

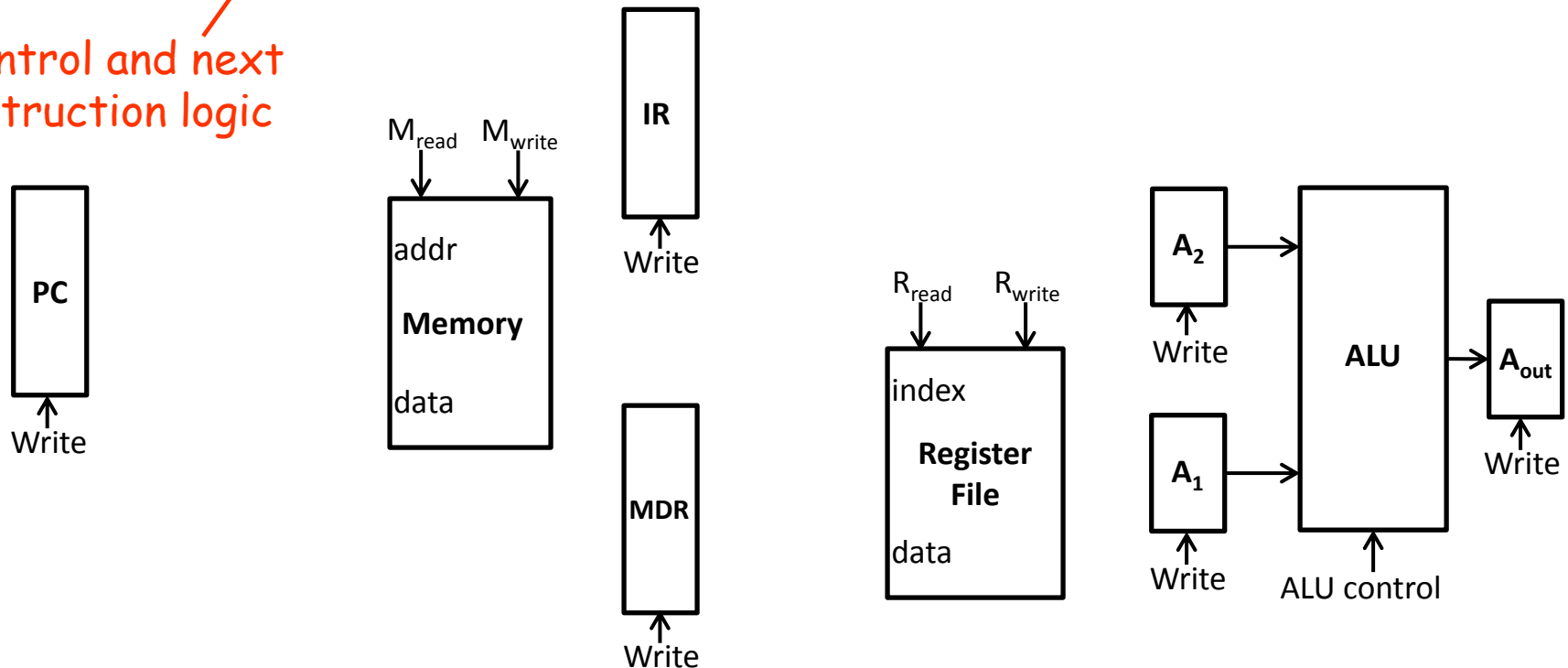


if ALU Control = 0 then $A_{out} = A_1 + A_2$
 if ALU Control = 1 then $A_{out} = A_1 - A_2$

10 read/write signals
 1 ALU Control signal
 4 multiplexer signals

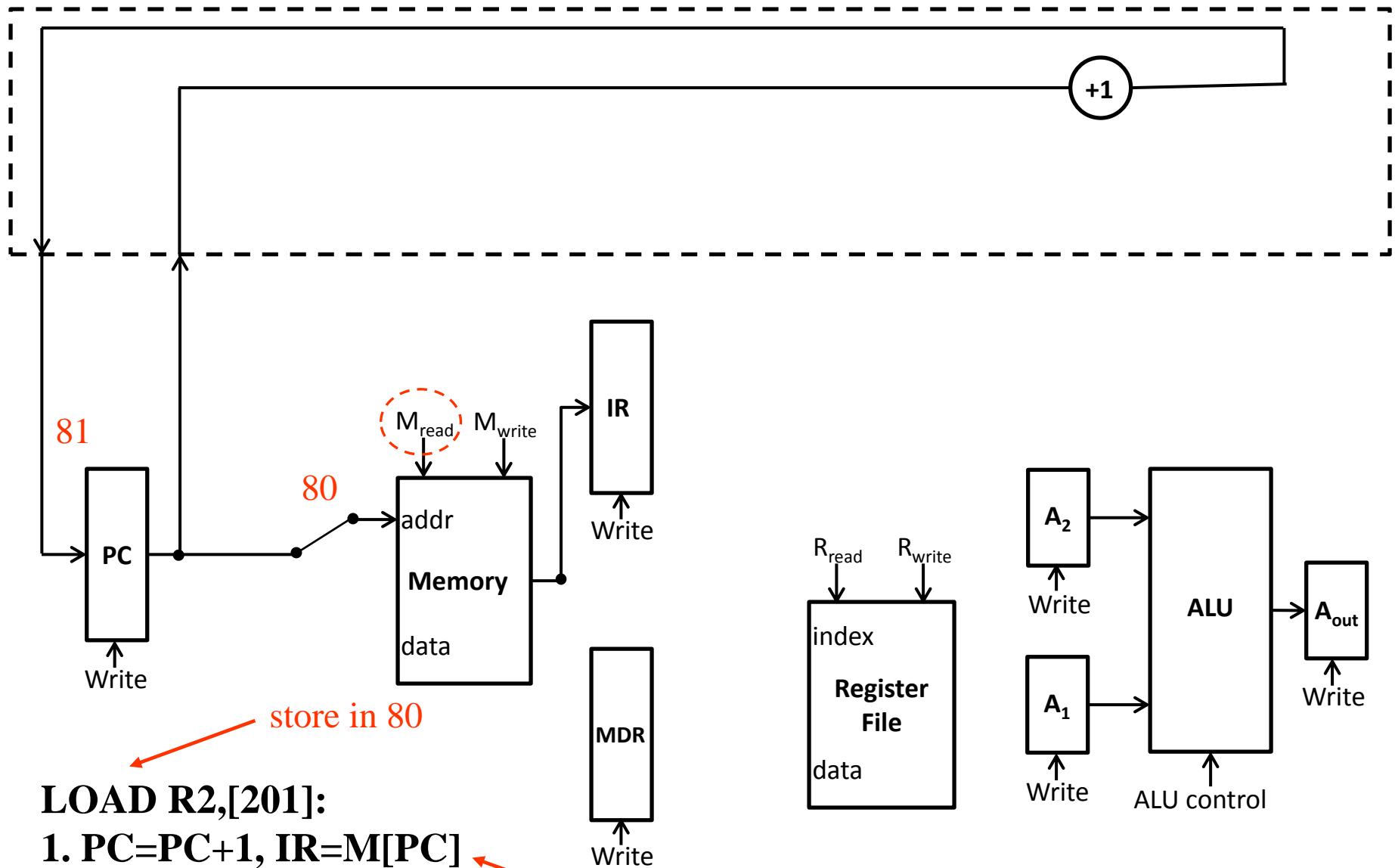
Abstraction: include the minimum

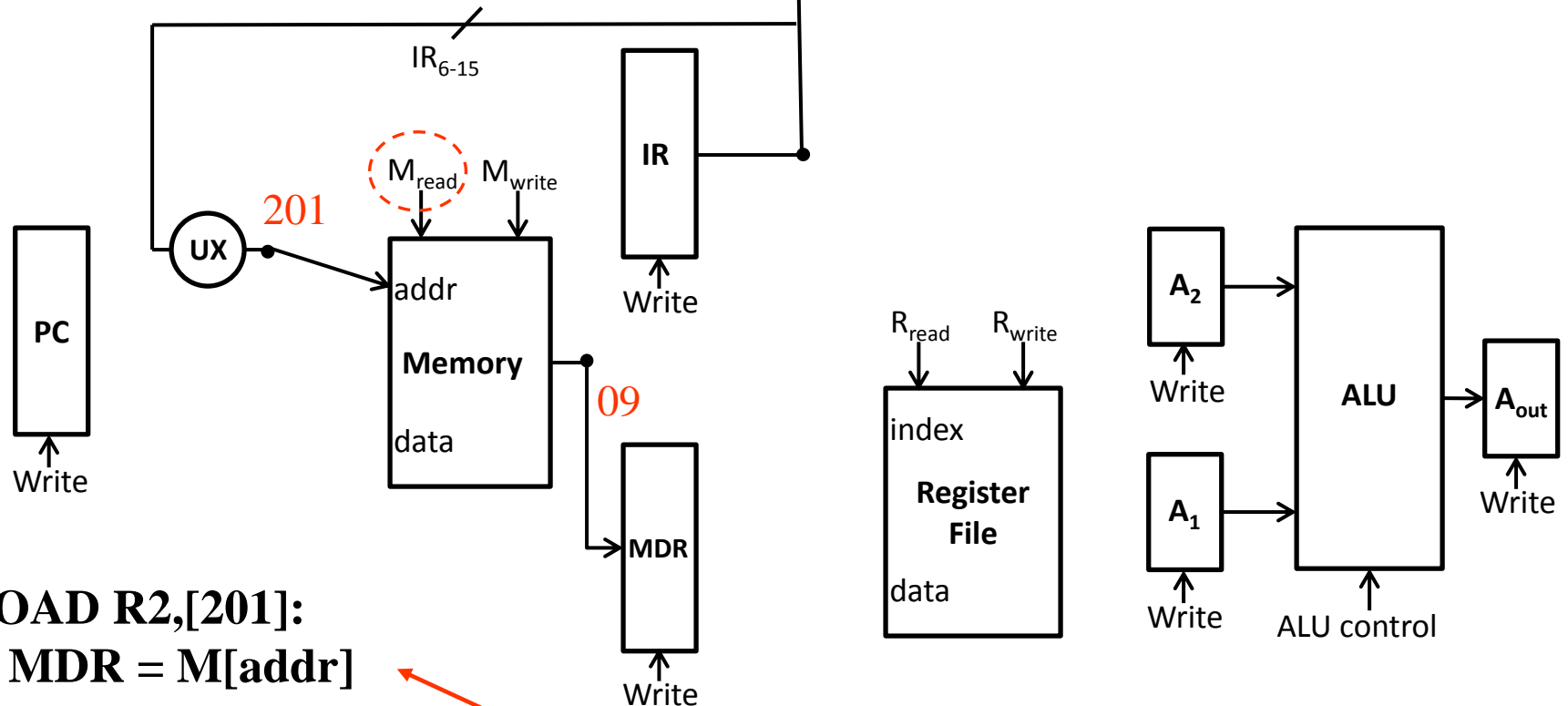
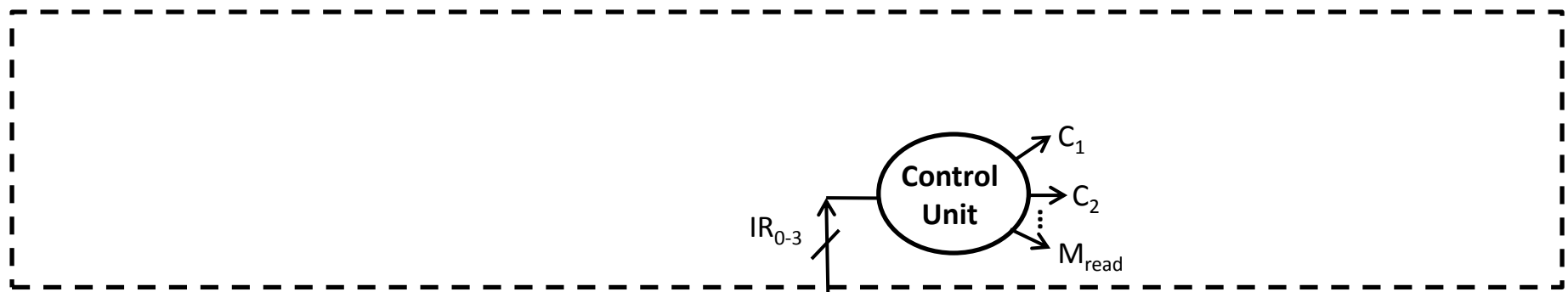
Control and next instruction logic



if ALU Control = 0 then $A_{out} = A_1 + A_2$
if ALU Control = 1 then $A_{out} = A_1 - A_2$

10 read/write signals
1 ALU Control signal





LOAD R2,[201]:

2. MDR = M[addr]

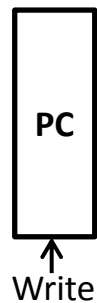
addr = UX(IR_{6..15})

RTL (or RT) description

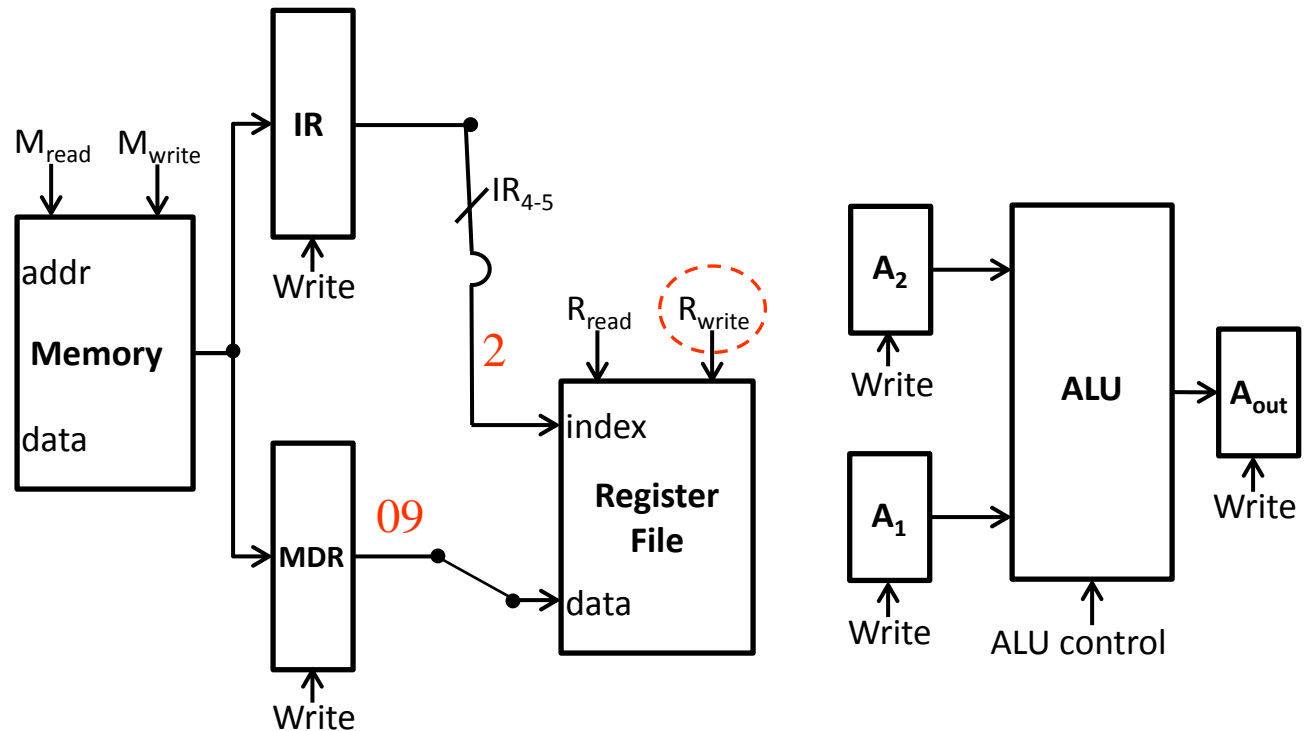
**UX: Unsign eXtension
from 10 to 16 bits**

if ALU Control = 0 then $A_{out} = A_1 + A_2$
if ALU Control = 1 then $A_{out} = A_1 - A_2$

10 read/write signals
1 ALU Control signal

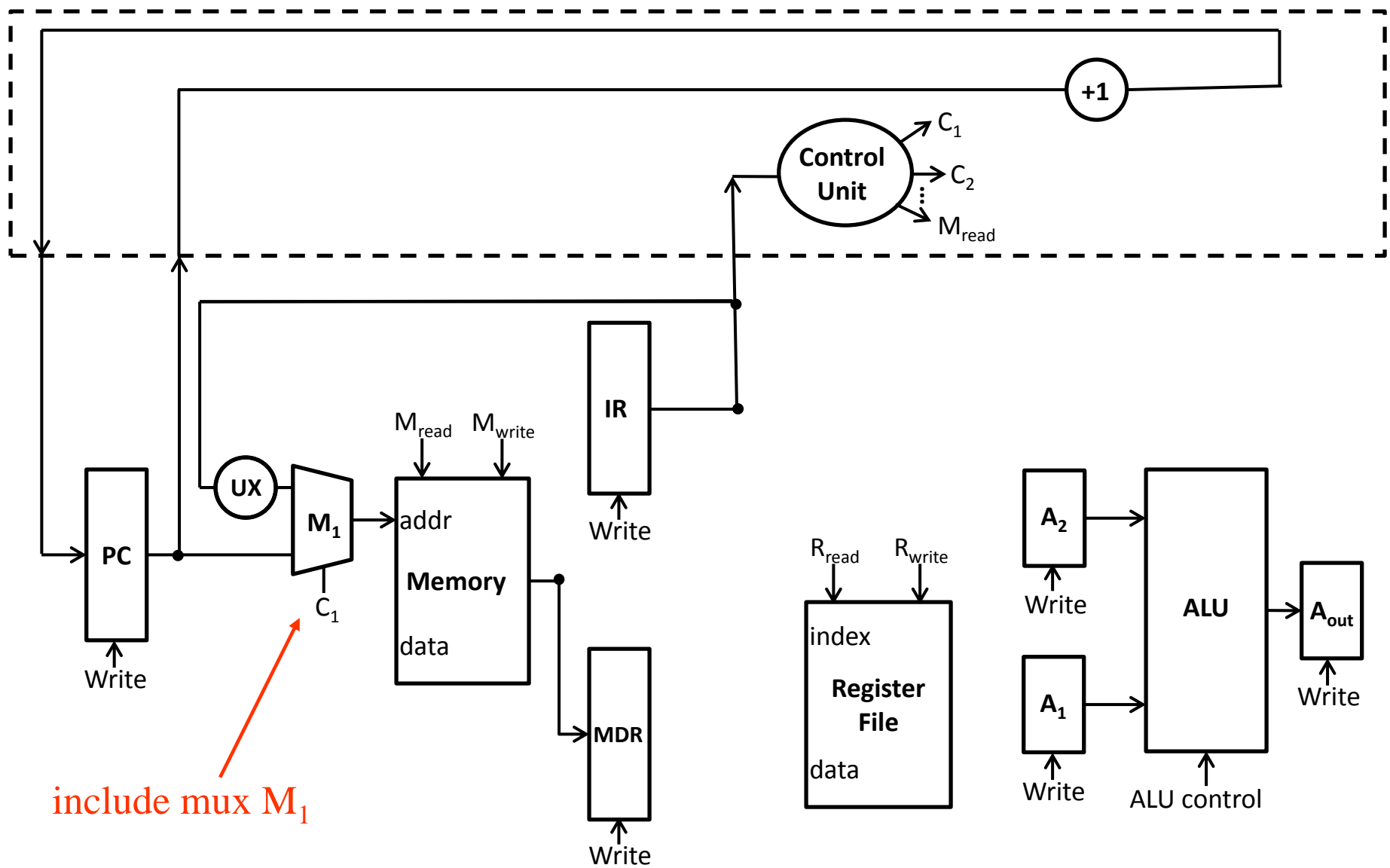


LOAD R2,[201]:
3. $R[IR_{4..5}] = MDR$

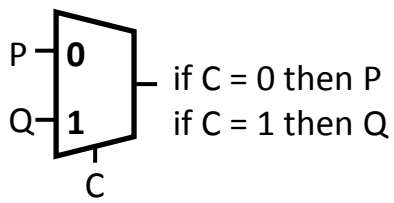


if ALU Control = 0 then $A_{out} = A_1 + A_2$
 if ALU Control = 1 then $A_{out} = A_1 - A_2$

10 read/write signals
 1 ALU Control signal



include mux M_1







if ALU Control = 0 then $A_{out} = A_1 + A_2$
 if ALU Control = 1 then $A_{out} = A_1 - A_2$

10 read/write signals
 1 ALU Control signal
 1 multiplexer signal

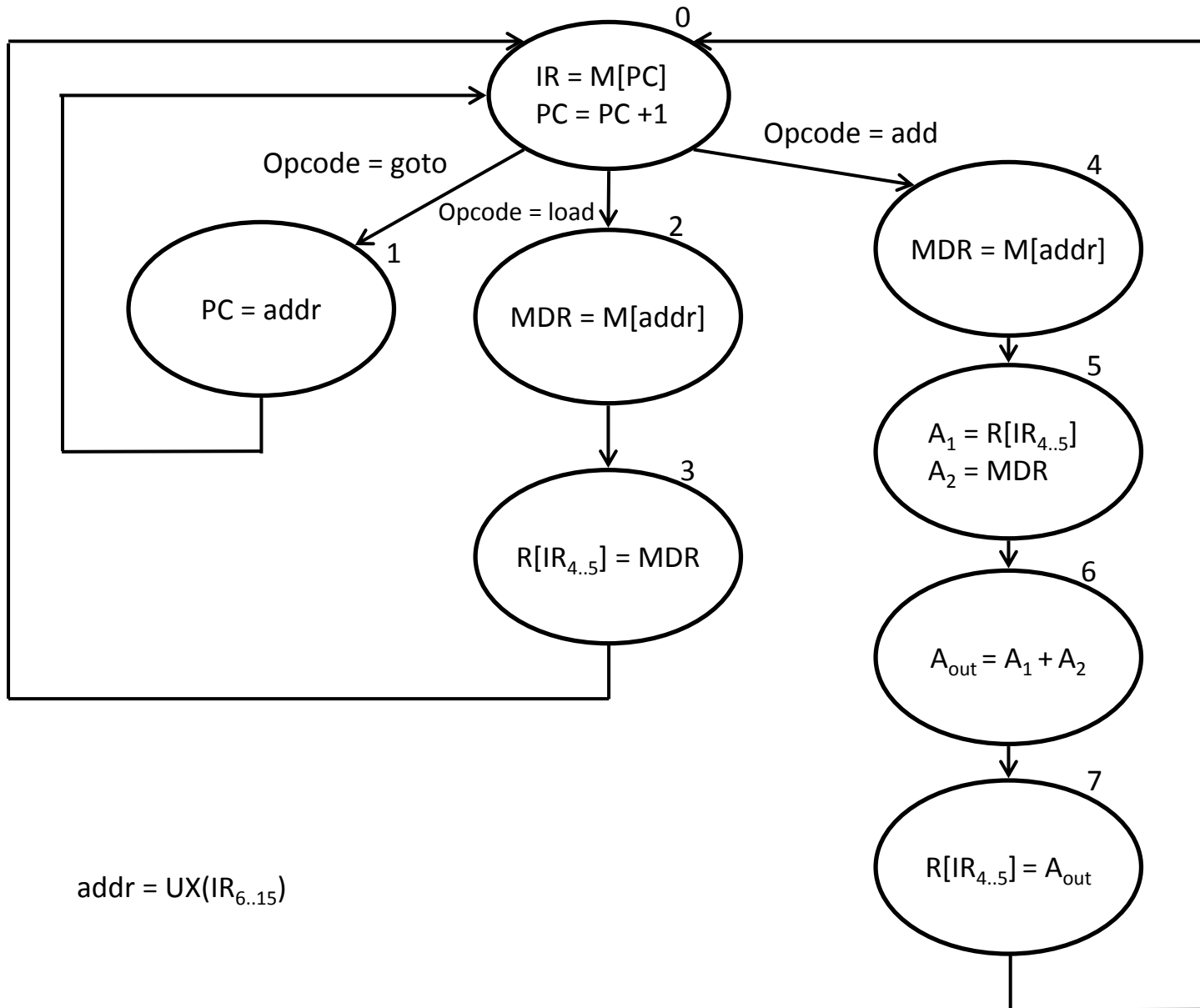
LOAD Rn, [Addr]: execution cycles

- effect: $R_n = M[Addr]$
 $= M[UX(Addr)]$
 $n = IR_{4..5}$ (register index)
 $Addr = IR_{6..15}$ (memory address)
- break down operation into small steps
 - 1 small step (e.g. add or memory access) each cycle
 - need temporary variables: A_1, A_2, A_{out}, MDR etc
- cycle 1: $PC = PC + 1, IR = M[PC]$ ← for all instructions
cycle 2: $MDR = M[UX(Addr)]$
cycle 3: $R[IR_{4..5}] = MDR$
- since $R[IR_{4..5}] = R_n = MDR = M[UX(Addr)] = M[Addr]$,
the instruction has the desired effect

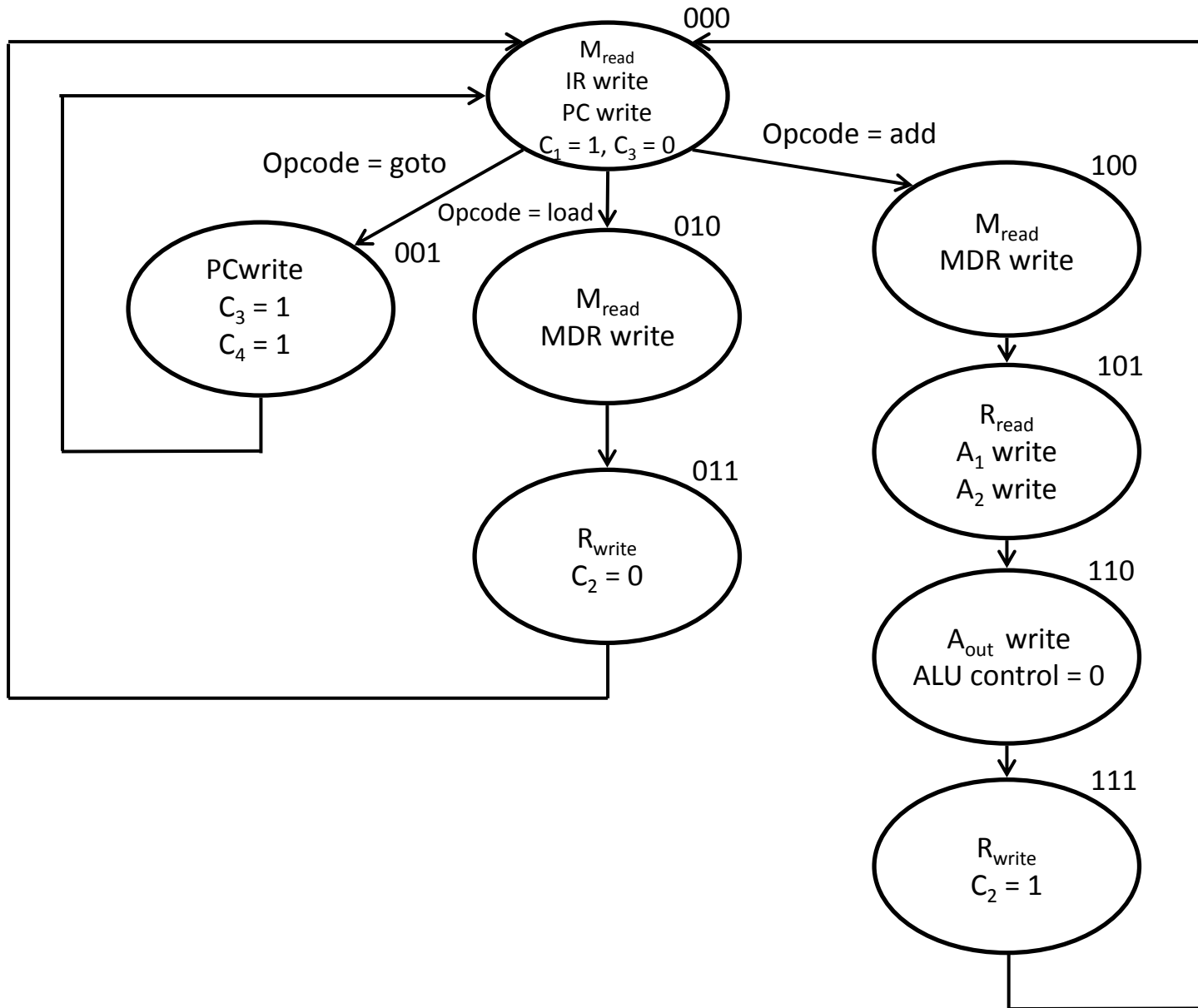
Micro-steps: program format

- $PC = PC + 1, IR = M[PC];$  PC, IR assignment in parallel
- if opcode == LOAD Rn, [Addr]
MDR = M[UX (IR_{6..15})];
R[IR_{4..5}] = MDR ↻  memory address of data to be loaded
 goto beginning
- if opcode == ADD Rn, [Addr]
MDR = M[UX (IR_{6..15})];
A₁ = R[IR_{4..5}], A₂ = MDR;
A_{out} = A₁ + A₂;
R[IR_{4..5}] = A_{out} ↻
- if opcode == GOTO Addr  memory address of instruction to be loaded
PC = UX (IR_{6..15}) ↻

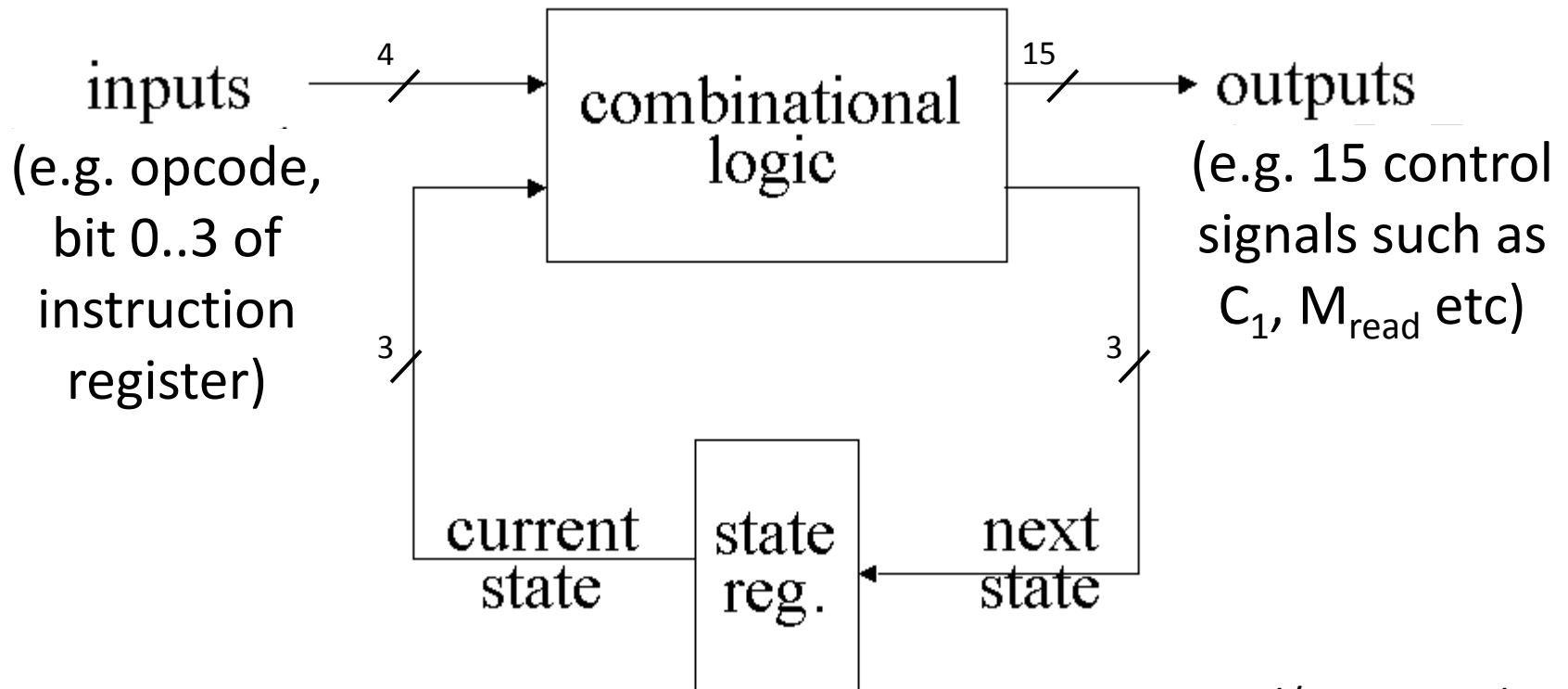
Micro-steps: RTL flow chart format



Micro-steps: state diagram format



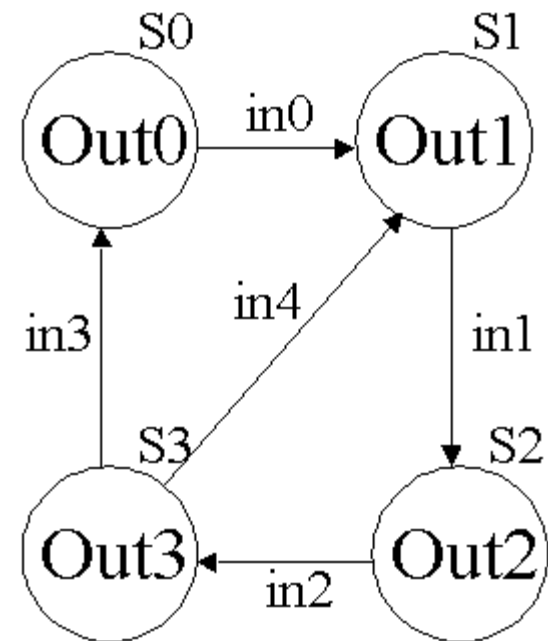
Finite-state machine (Mealy)



10 read/write signals
1 ALU Control signal
4 multiplexer signals

State table and state diagram

Input	Current state	Output	Next state
in0	S0	out0	S1
in1	S1	out1	S2
in2	S2	out2	S3
in3	S3	out3	S0
in4	S3	out3	S1



Direct FSM implementation: output function

- inputs: opcode from IR, 4 bits
- register: records state, 8 states \Rightarrow 3 bits (S2, S1, S0)
 - simplified state diagram: load, add, goto instructions
- derive state table from state diagram
 - e.g. PCwrite = 1 in state 0 and state 1:

input	current state	PCwrite output
-	000	1
-	001	1

$$\text{so PCwrite} = \overline{S2} \cdot \overline{S1} \cdot \overline{S0} + \overline{S2} \cdot \overline{S1} \cdot S0$$

and

or

negate

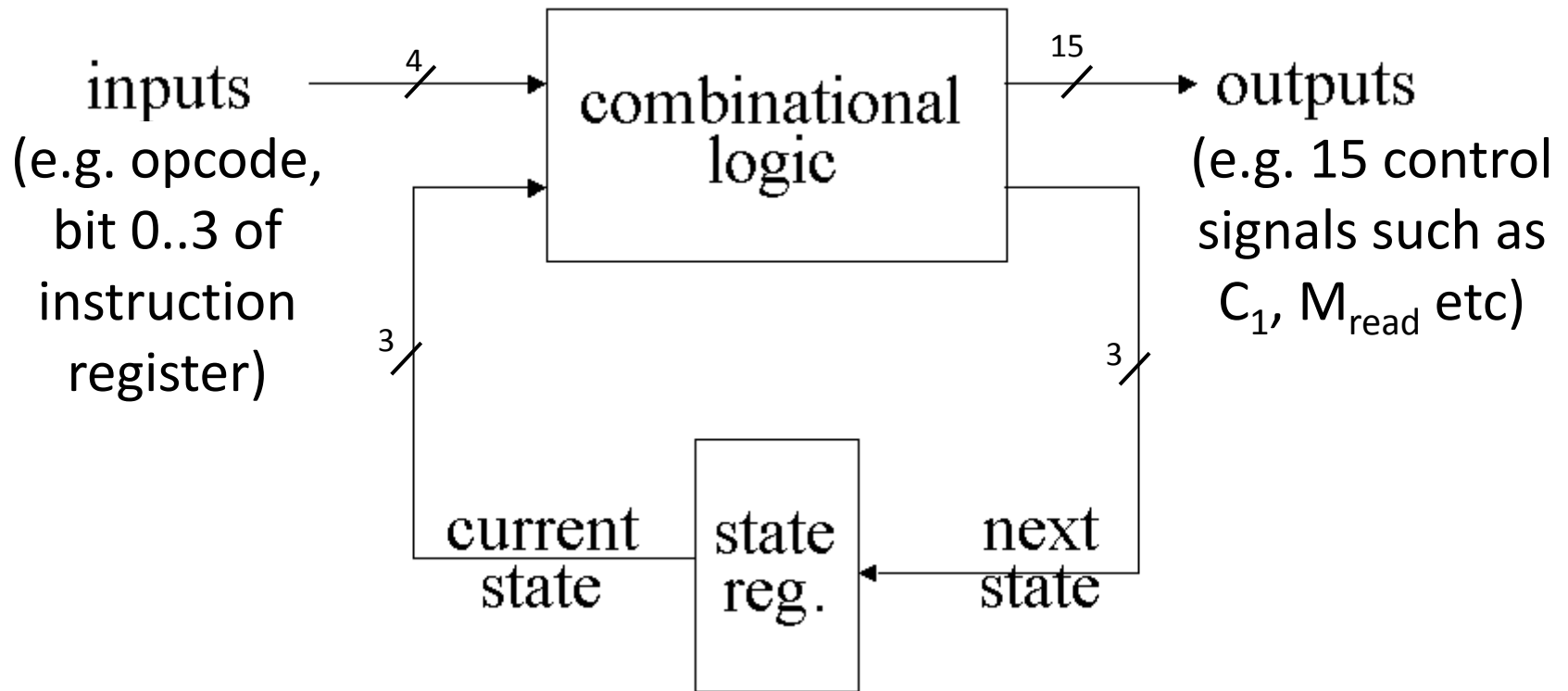
Direct FSM implementation: next state function

- control outputs only depend on current state
(not directly on external opcode input)
- next state outputs, ns2, ns1, ns0, depend on both
current state and external input
 - e.g. next state bit 0: ns0 = 1 for state 1, 3, 5, and 7

opcode input	current state	ns0
goto: 5 0101	000	1 (state 1)
load: 1 0001	010	1 (state 3)
add : 3 0011	100	1 (state 5)
add : 3 0011	110	1 (state 7)

$$\begin{aligned}
 \text{so ns0} = & \overline{i_3} \cdot \overline{i_2} \cdot \overline{i_1} \cdot i_0 \cdot \overline{S_2} \cdot \overline{S_1} \cdot \overline{S_0} + \\
 & \overline{i_3} \cdot \overline{i_2} \cdot \overline{i_1} \cdot i_0 \cdot \overline{S_2} \cdot S_1 \cdot \overline{S_0} + \\
 & \overline{i_3} \cdot \overline{i_2} \cdot i_1 \cdot i_0 \cdot \overline{S_2} \cdot \overline{S_1} \cdot \overline{S_0} + \\
 & \overline{i_3} \cdot \overline{i_2} \cdot i_1 \cdot i_0 \cdot S_2 \cdot S_1 \cdot \overline{S_0}
 \end{aligned}$$

Finite-state machine: ROM implementation



- ROM size: 7 bit (4-bit opcode, 3-bit state)
x 18 (15-bit control, 3-bit state)
- total size: $2^7 \times 18 = 2.3\text{K}$ bits