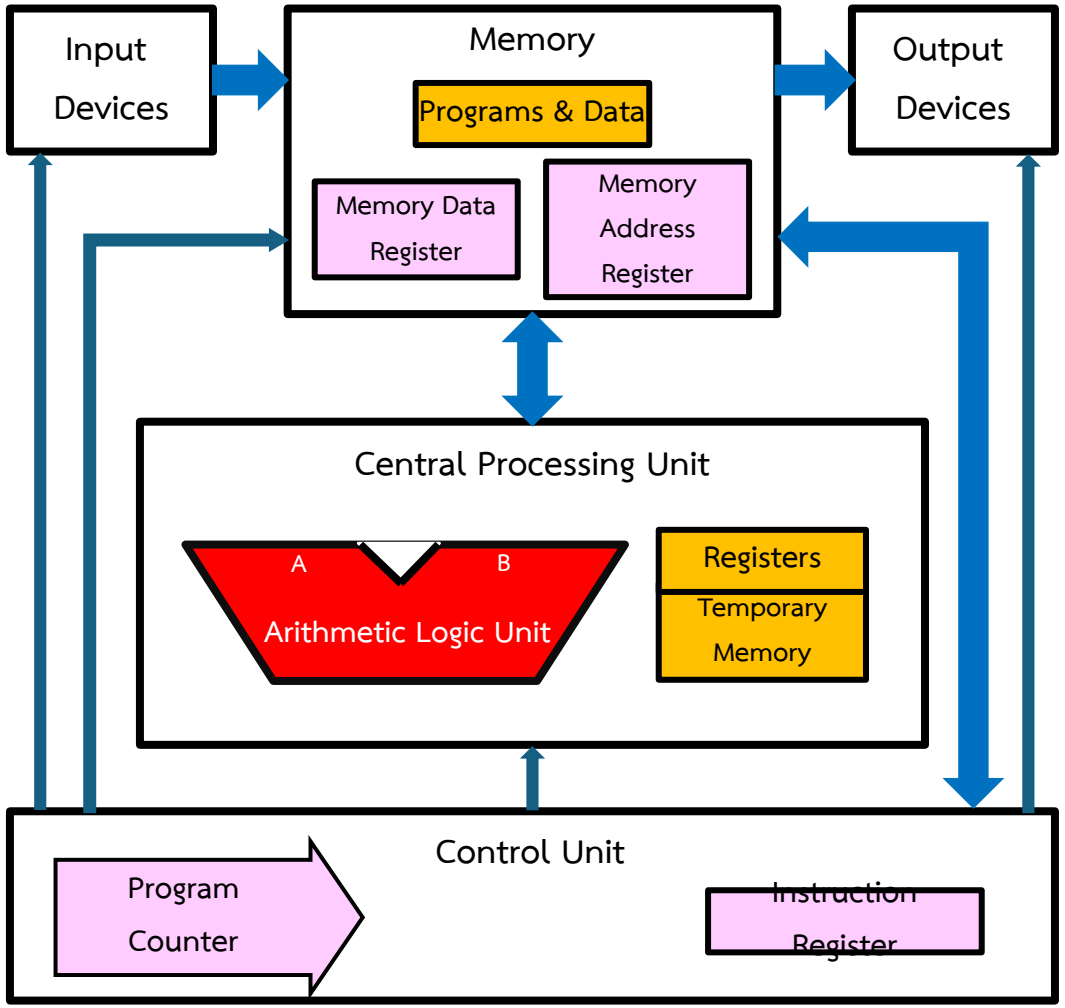
310-2202 โครงสร้างของระบบคอมพิวเตอร์ (Computer Organization)

Week 4: The von Neumann Model

ดำรงค์ฤทธิ์ เศรษฐ์ศิริโชค

Topic

- The von Neumann Model
- LC-3 instruction
- Instruction Processing
- Changing the Sequence of Instructions
- Control of the Instruction Cycle
- Single-Cycle Implementation



Review

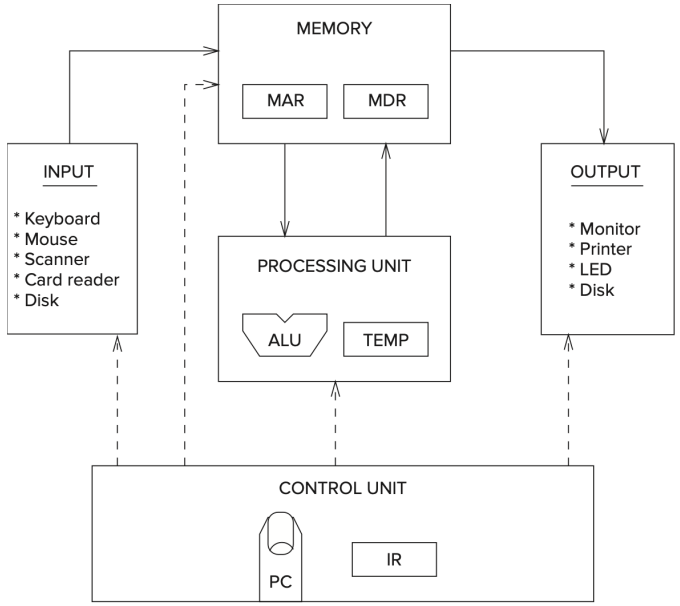
- MOS transistors are used as switches to implement logic functions.
  - N-type: connect to GND, turn on (with 1) to pull down to 0
  - P-type: connect to +2.9V, turn on (with 0) to pull up to 1
- Basic gates: NOT, NOR, NAND
  - Logic functions are usually expressed with AND, OR, and NOT
- Properties of logic gates
  - Completeness: can implement any truth table with AND, OR, NOT
  - DeMorgan's Law: convert AND to OR by inverting inputs and output

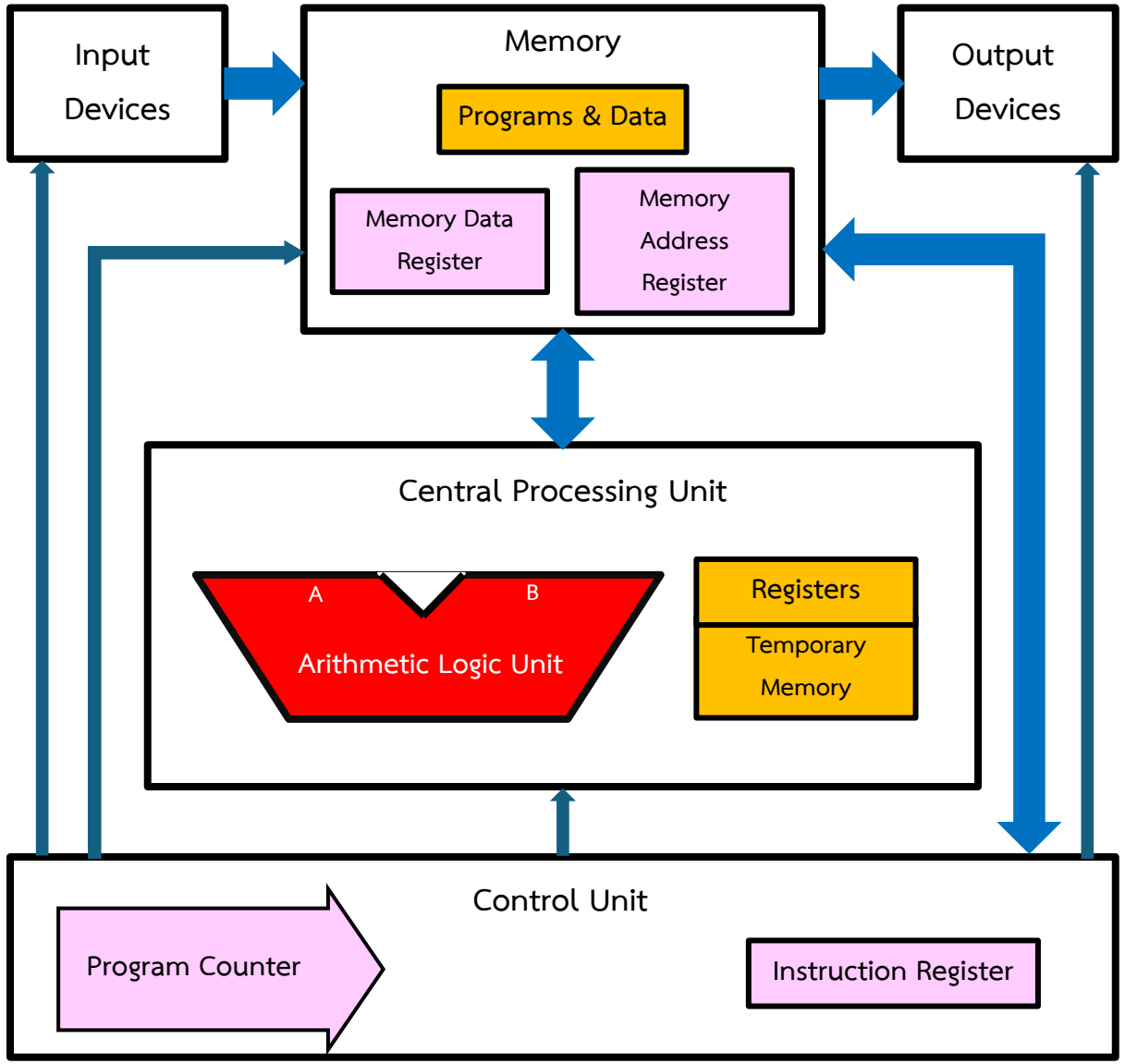
Review

- Basic digital logic
  - Transistors / Gates
  - Storage (latches, flip-flops, memory)
  - State machines
- Some simple circuits
  - Adder, subtracter, adder/subtracter, Incrementor
  - Traffic sign state machine
- **Next: Stored program computer (Von Neumann Model--A Machine Structure)**

The von Neumann Model

- Basic Components for a machine
- The LC-3: An Example of von Neumann Machine
- Instruction Processing





Two major inventions of the microprocessor chip

Stored program

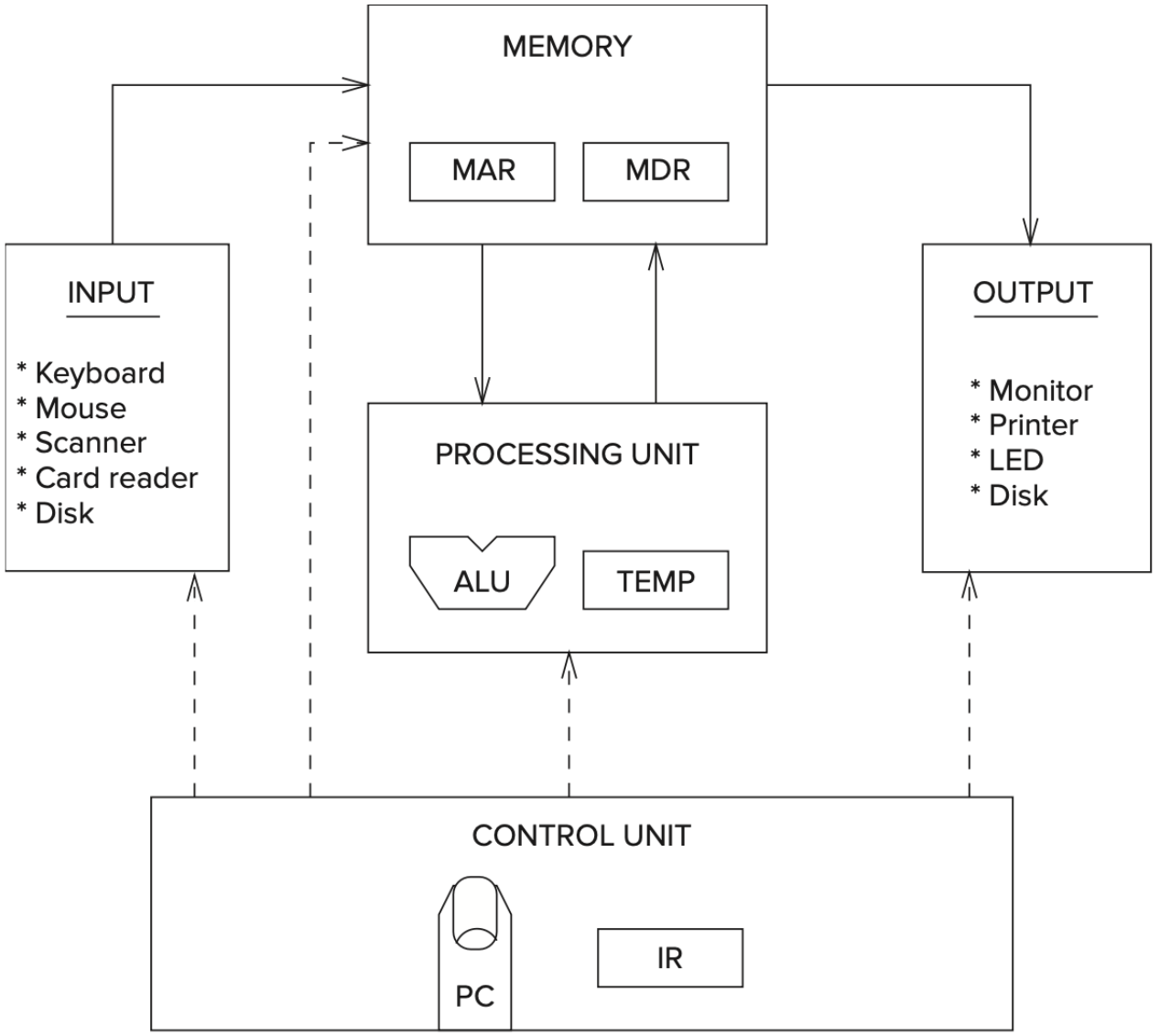
+

Transistor technology

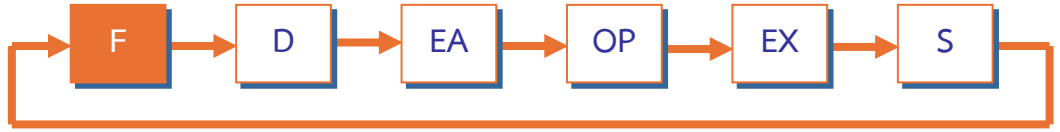
Change the program so that you can do all kinds of tasks on the same hardware

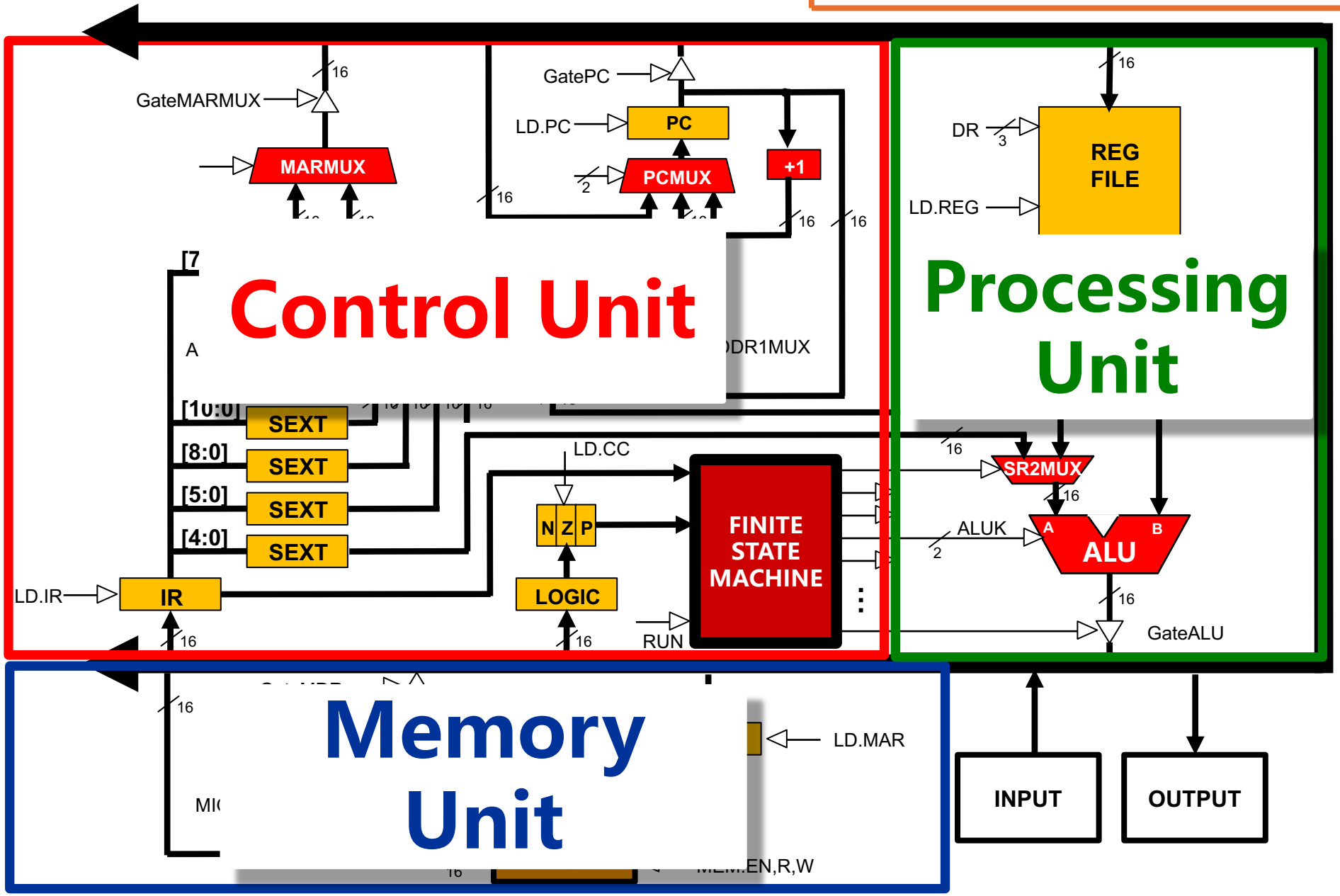
The device is smaller and faster than a vacuum tube

Basic Components



LC-3 Data Path





Memory

- A memory can be built – a logical ***k x m* array of stored bits**.

Address

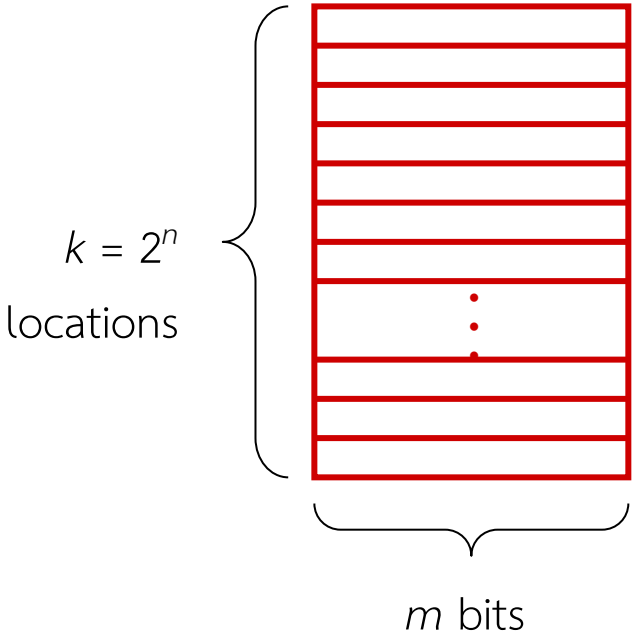
- unique (*n*-bit) identifier of location

Contents

- *m*-bit value stored in location

Basic Operations:

- LOAD: read a value from a memory location
- STORE: write a value to a memory location



000	
001	
010	
011	
100	00000110
101	
110	00000100
111	

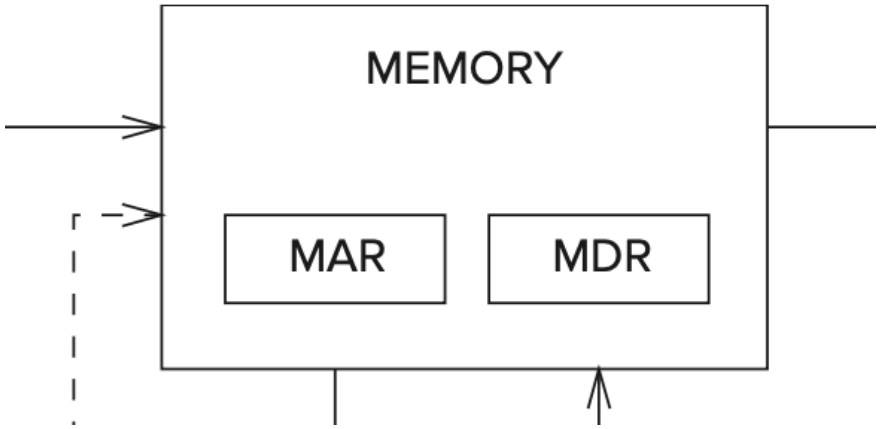
Location 6 contains the value 4; location 4 contains the value 6.

Interface to Memory

How does processing unit get data to/from memory?

MAR: Memory Address Register

MDR: Memory Data Register



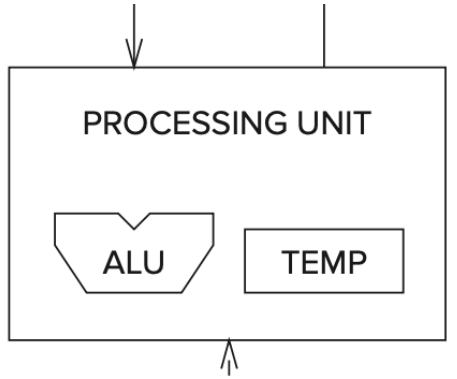
To read a location (A):

1. Write the address (A) into the MAR.
2. Send a "read" signal to the memory.
3. Read the data from MDR.

To write a value (X) to a location (A):

1. Write the data (X) to the MDR.
2. Write the address (A) into the MAR.
3. Send a "write" signal to the memory.

Processing Unit



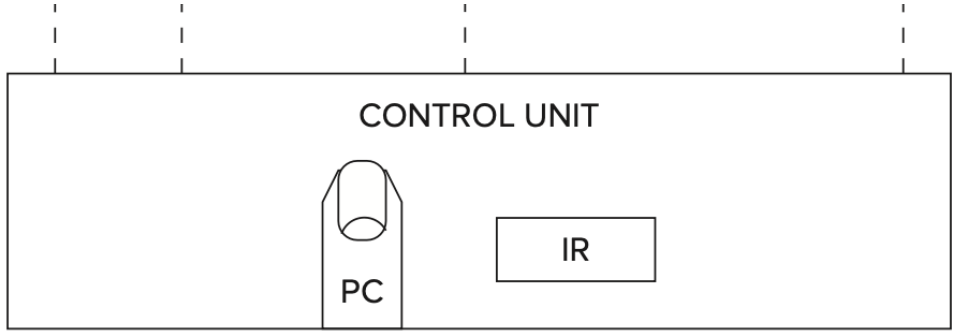
- Modern computer can consist of many complex functional units
  - some of them special-purpose
  - divide, square root, etc.
- The simplest processing unit: Arithmetic and Logic Unit (**ALU**)
  - LC-3 performs ADD, AND, NOT
- **Registers**
  - Small, temporary storage
  - Operands and results of functional units
  - **LC-3 has 8 register (R0, …, R7)**
- **Word Size**
  - number of bits normally processed by ALU in one instruction also width of registers
  - LC-3 is 16 bits

Input and Output

- Devices for getting data into and out of computer memory
- Each device has its own interface, usually a set of registers like the memory's MAR and MDR
  - LC-3 supports keyboard (input) and console (output)
  - keyboard: data register (KBDR) and status register (KBSR)
  - console: data register (CRTDR) and status register (CRTSR)
  - frame buffer: memory-mapped pixels
- Some devices provide both input and output
  - disk, network
- Program that controls access to a device is usually called a *driver*.

INPUT	*OUTPUT*
Keyboard	**Monitor**
Mouse	**Printer**
Scanner	**LED**
Disk	**Disk**

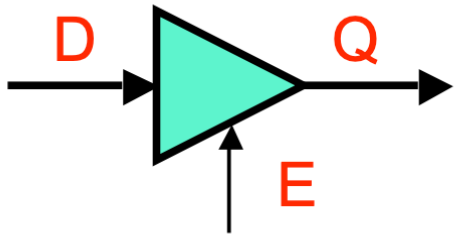
Control Unit



- Orchestrates execution of the program
- Instruction Register (IR) contains the ***current instruction***.
- Program Counter (PC) contains the ***address*** of the **next** instruction to be executed.
- Control unit:
  - Reads an instruction from memory the instruction's address is in the PC
  - Interprets the instruction, generating signals that tell the other components what to do an instruction may take many *machine cycles* to complete

One More Gate

- Tri-state buffer
  - NOT an inverter!

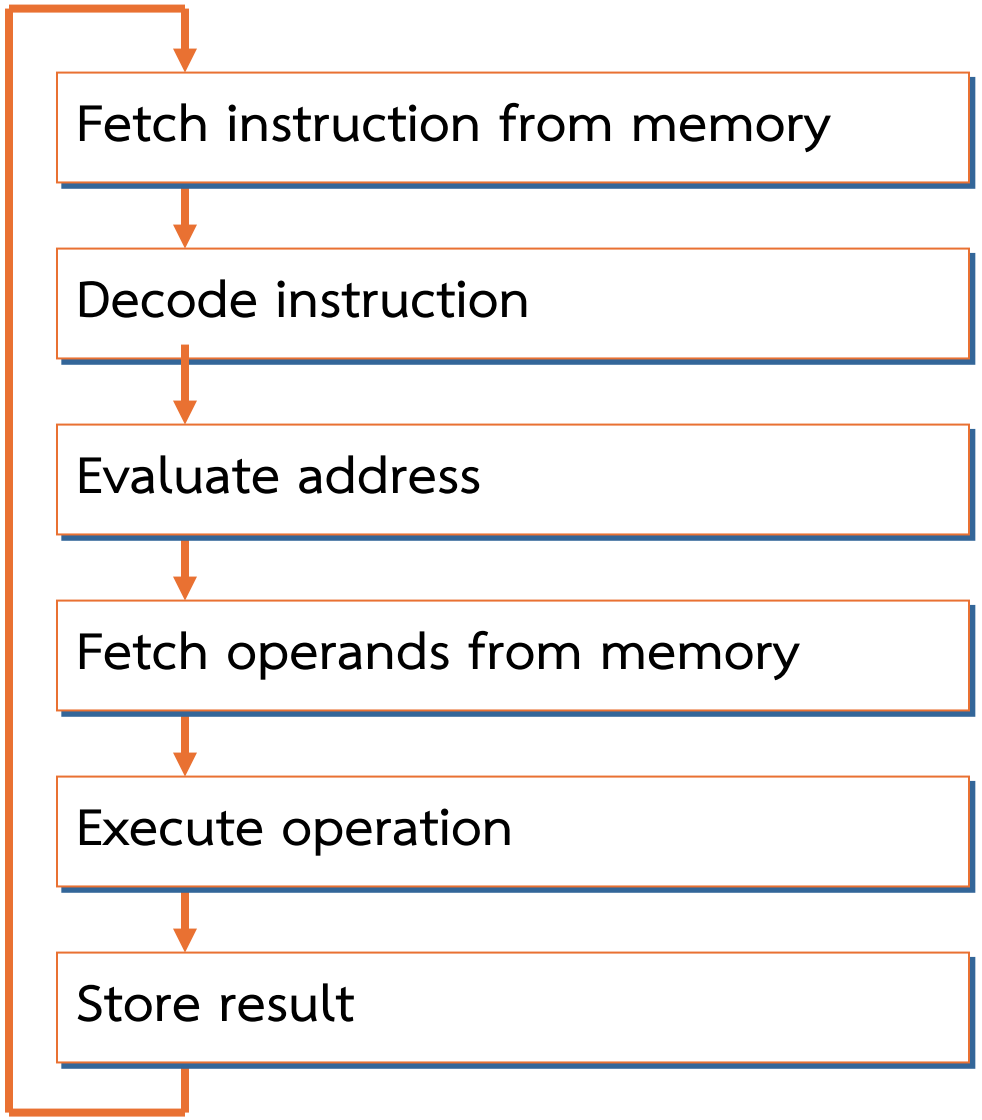


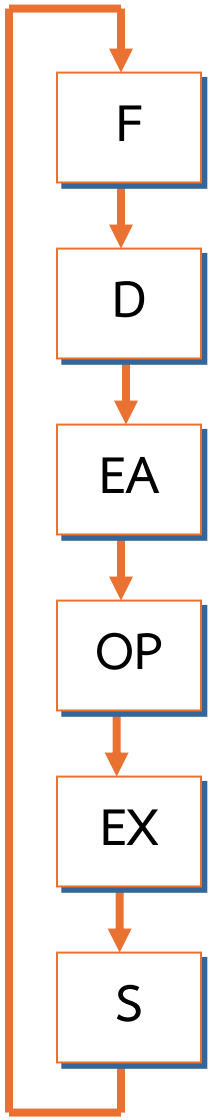
- Allows wires to be “shared”
  - Alternative to mux
  - Only one source may drive at a time

E	D	Q
1	0	0
1	1	1
0	0	Z
0	1	Z

Z = “High Impedance” State
(no current, i.e., no “pressure”)

Instruction Processing



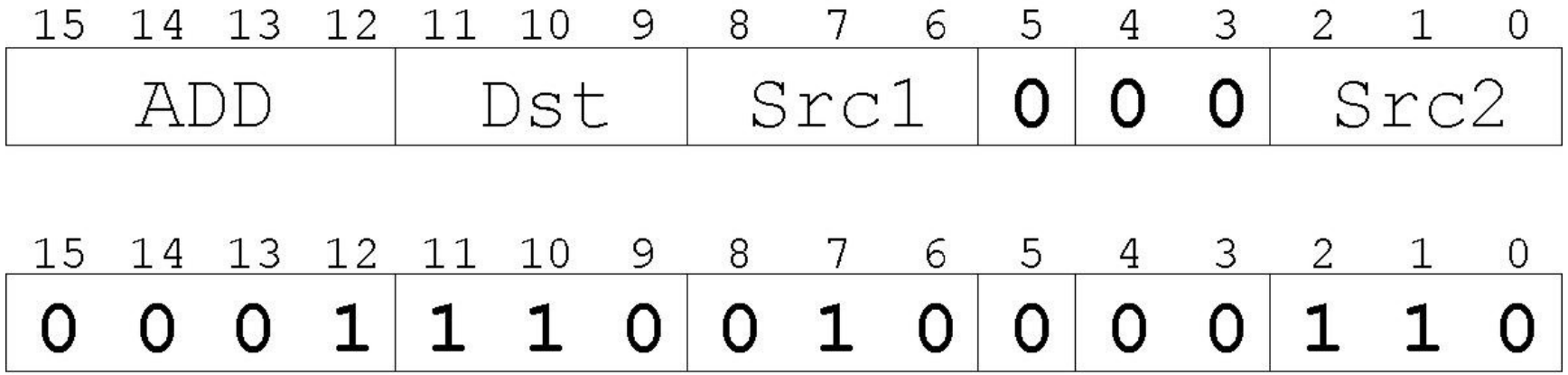


Instruction

- The most basic unit of computer processing
  - The sequence of bits, like Data
- Made up of 2 parts,
  - **Opcode** (what the instruction does)
  - **Operands** (who it does it to!)
- **3 kinds of instructions: operates, data movement, and control**
- A computer's instructions and their formats is known as its *Instruction Set Architecture (ISA)*

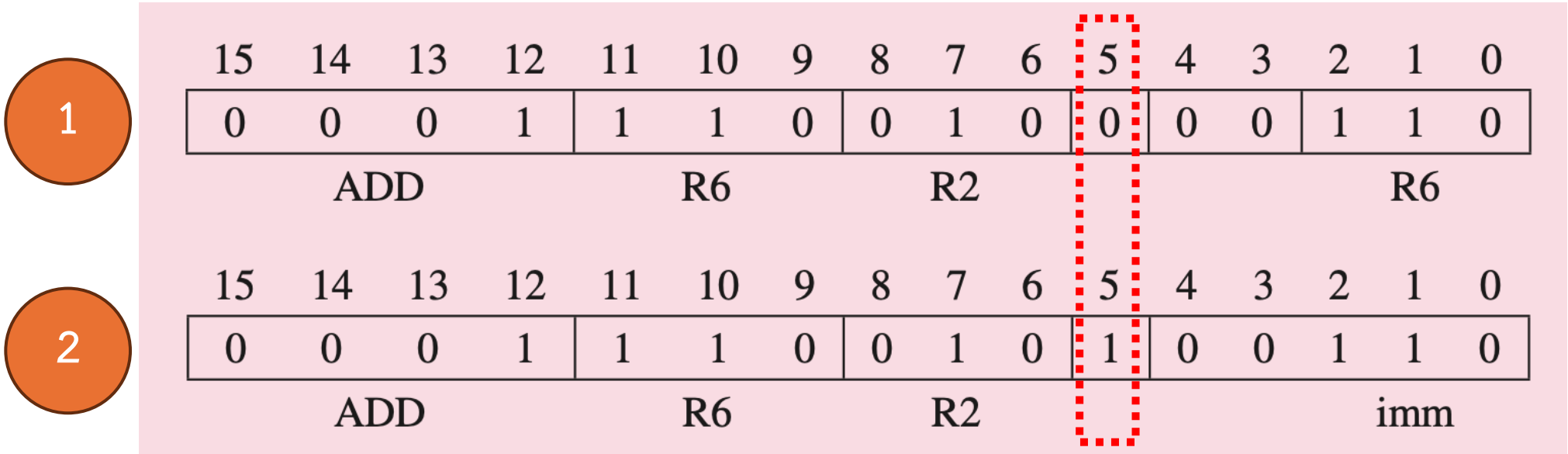
LC-3 instruction

- 16-bit instructions
- 4-bits Opcode ➔ 2^4 distinct opcodes
- 8 registers (R0-R7) for temporary storage, sources and destination of **ADD** are registers



"Add the contents of R2 to the contents of R6 and store the result in R6."

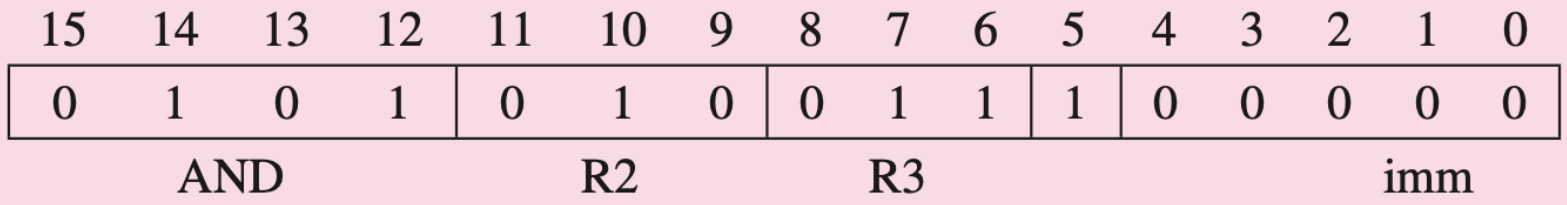
Example: LC-3 ADD Instruction



- The only difference in the two formats is the 1 or 0 stored in bit 5
  1. Add the contents of register 2 (R2) to the contents of register 6 (R6) and store the result back into register 6 (R6)
  2. Add the contents of register 2 (R2) to the ***positive integer 6*** and store the result into register 6

Example: LC-3 AND Instruction (Initializing Register to Zero)

The AND Instruction The AND instruction is also an operate instruction, and its behavior is essentially identical to the ADD instruction, except for one thing. Instead of ADDing the two source operands, the AND instruction performs a bit-wise AND of the corresponding bits of the two source operands. For example, the instruction shown below



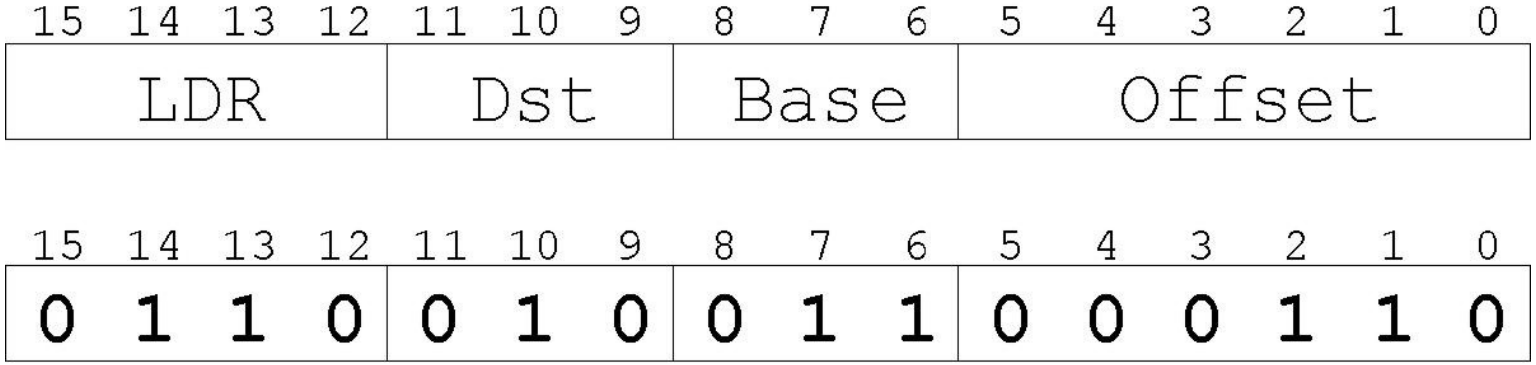
is an AND instruction since bits [15:12] = 0101. The two sources are R3 and the immediate value 0. The instruction loads R2 with the value 0 since the AND instruction performs a bit-wise AND where the bit of the second operand is always 0. As we shall see, this instruction is a convenient technique for making sure a particular register contains 0 at the start of processing. We refer to this technique as *initializing* R2 to 0.

Example: LC-3 LDR Instruction (Base + offset)

- Load instruction -- reads data from memory

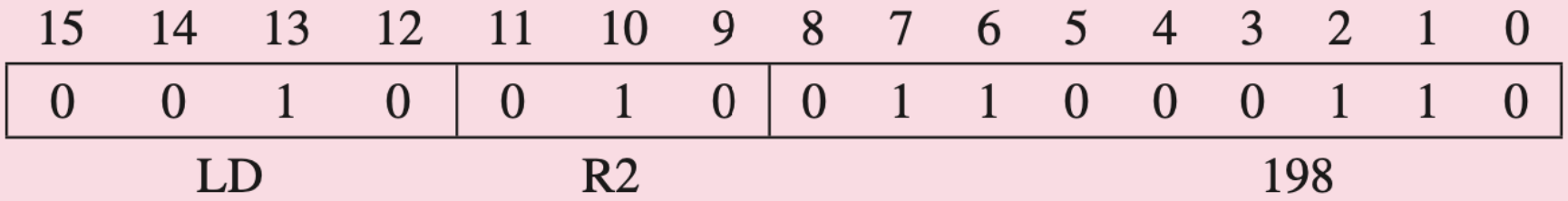
Base + offset mode:

  - Add offset to base register -- result is memory address
  - Load from memory address into destination register



Example: LC-3 LD Instruction (PC + offset)

The 16-bit LC-3 LD instruction has the following format:



Add 198 to the contents of the PC to form the address of a memory location and to load the contents of that memory location into R2

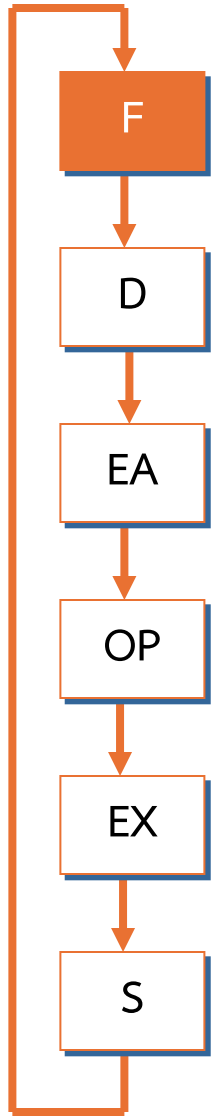
If bits [8:0] had been 111111001, the instruction would have been interpreted: "Add –7 to the contents of the PC to form the address of a memory location."

The Instruction Cycle (NOT the Clock Cycle!)

- The entire sequence of steps needed to process an instruction is called the *instruction cycle*
- 6 sequential phases, each phase requiring **zero** or more steps
  - FETCH
  - DECODE
  - EVALUATE ADDRESS FETCH OPERANDS
  - EXECUTE
  - STORE RESULT

Instruction Processing: FETCH

- Load next instruction (at address stored in PC) from memory into Instruction Register (IR)
  1. Load contents of PC into MAR
  2. Send “read” signal to memory
  3. Read contents of MDR, store in IR
- Then increment PC, so that it points to the next instruction in sequence
  - PC becomes PC+1

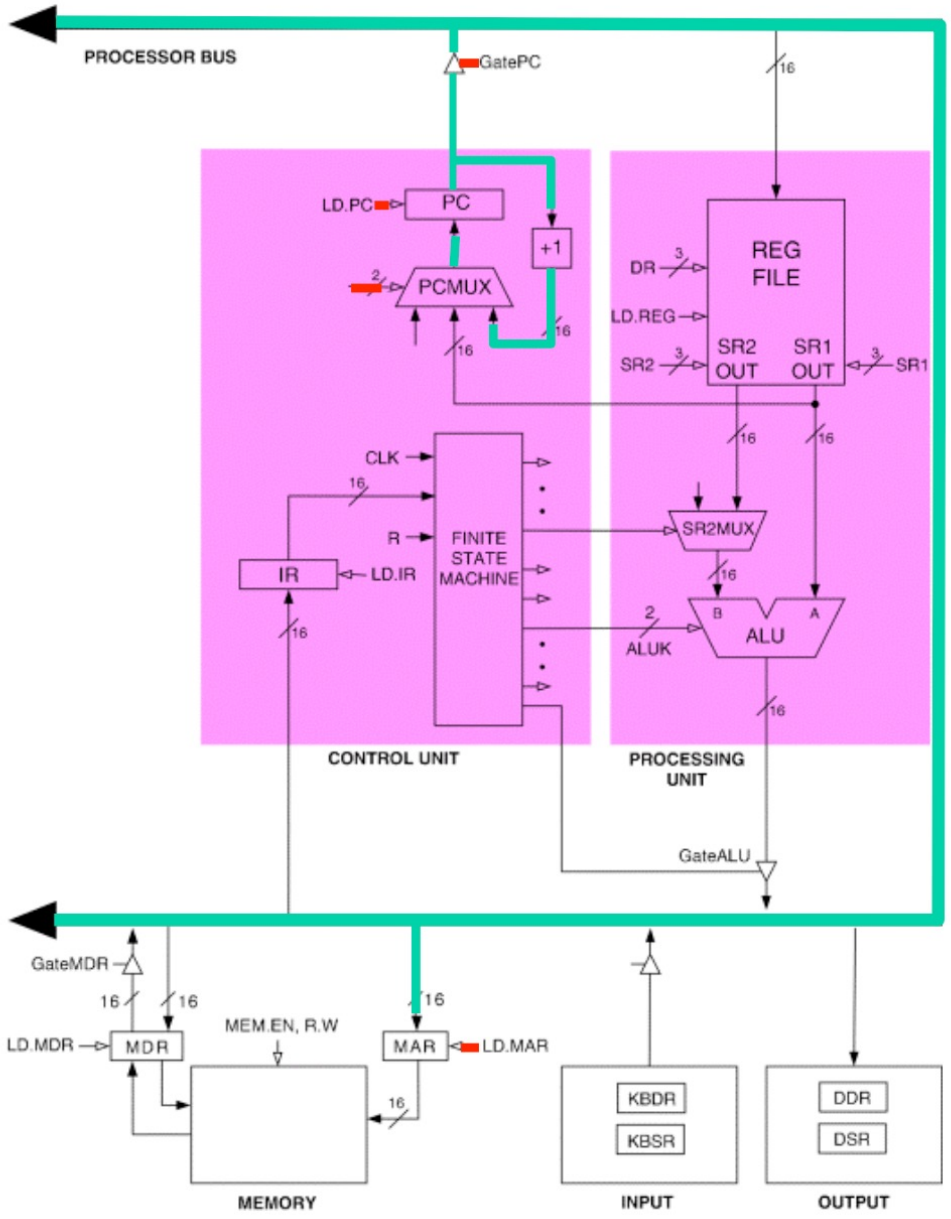


Instruction Processing: FETCH

1. Load contents of PC into MAR
2. Send “read” signal to memory
3. Read contents of MDR, store in IR

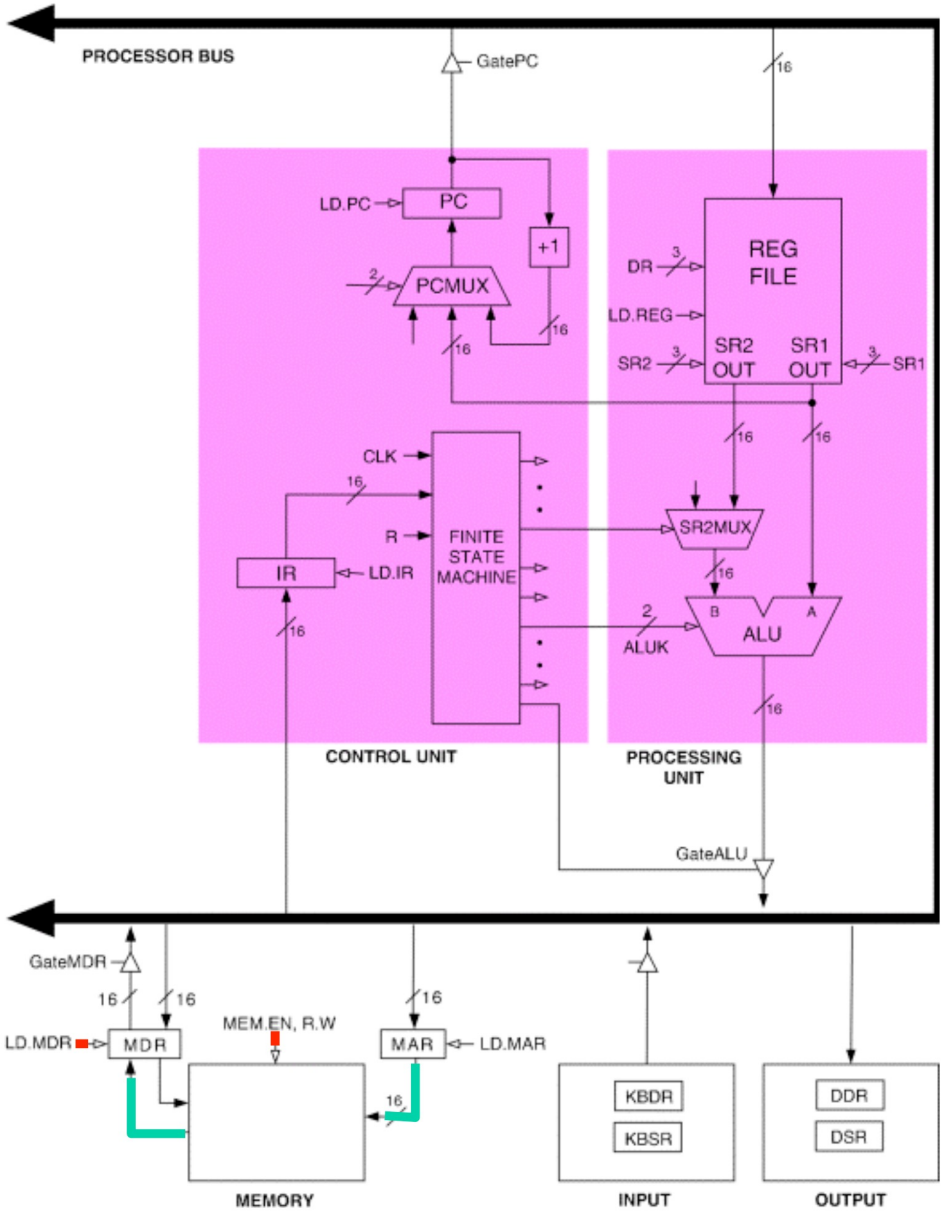
→ Control

— Data



Instruction Processing: FETCH

1. Load contents of PC into MAR
2. Send “read” signal to memory
3. Read contents of MDR, store in IR

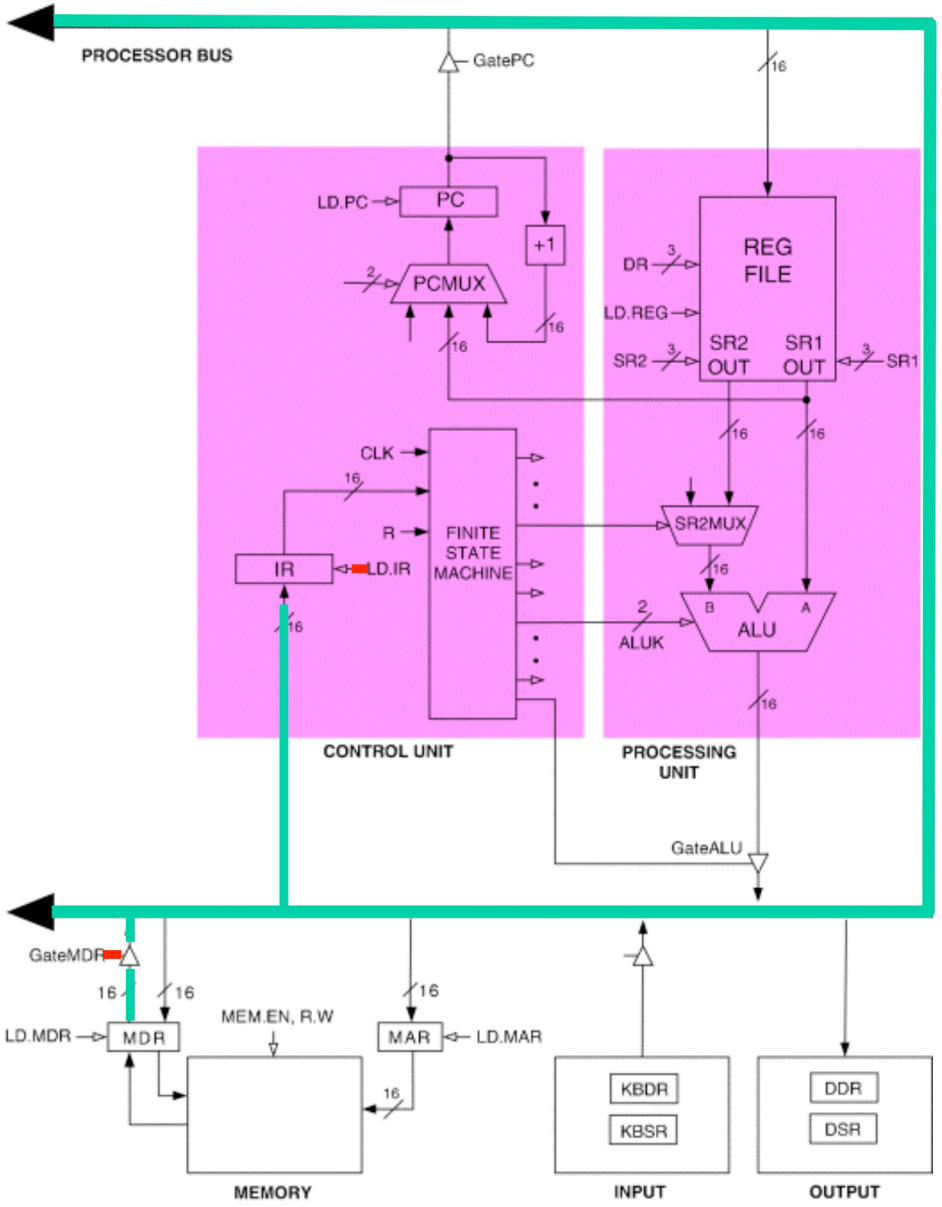


→ Control

— Data

Instruction Processing: FETCH

1. Load contents of PC into MAR
2. Send “read” signal to memory
3. Read contents of MDR, store in IR

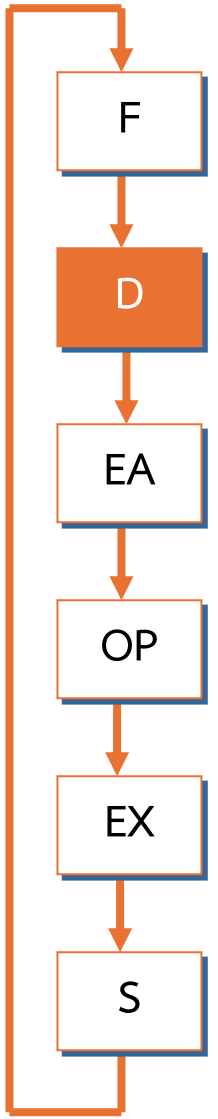


→ Control

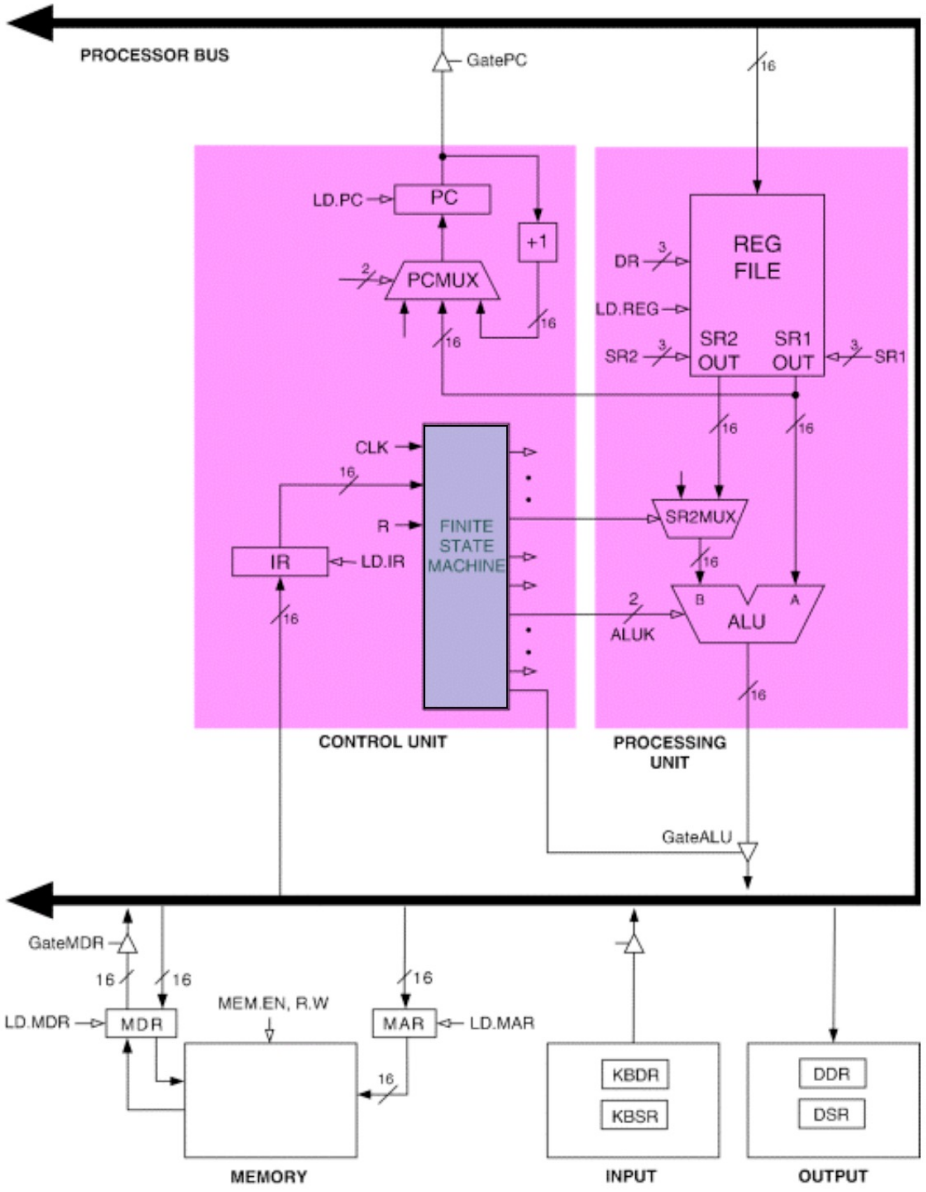
— Data

Instruction Processing: DECODE

- First identify the opcode
  - In LC-3, this is always the first **4 bits** of instruction.
  - A **4-to-16 decoder** asserts a control line corresponding to the opcode
- Depending on opcode, identify other operands from the **remaining bits (12 bit)**
  - Example:
    - for ADD, last 3 bits is source operand
    - for LDR, last 6 bits is offset

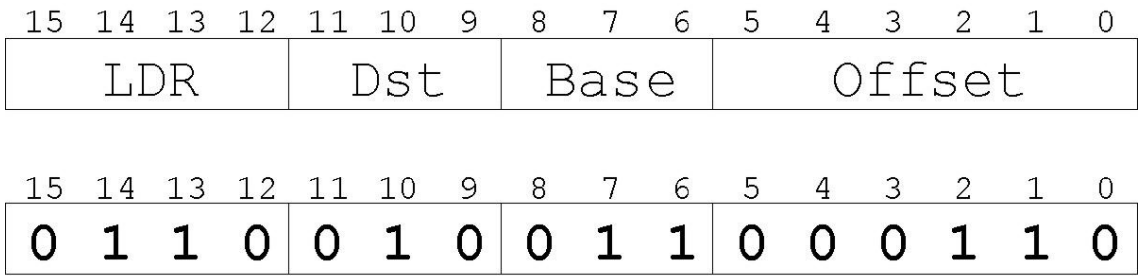


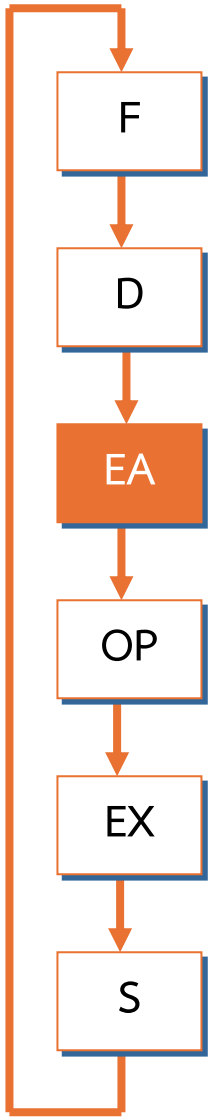
Instruction Processing: DECODE



Instruction Processing: EVALUATE ADDRESS

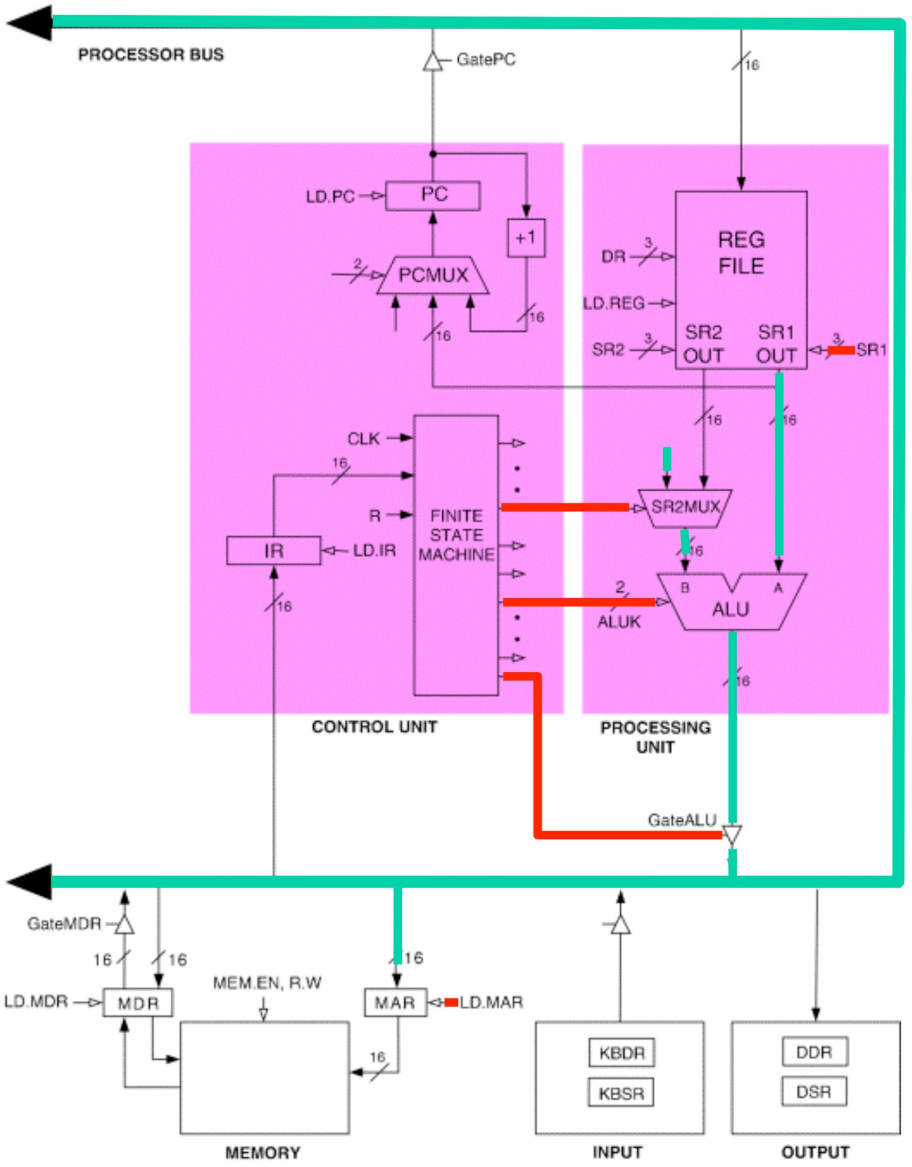
- For instructions that require memory access, compute address used for access.
- Examples:
  - `add offset to base register (as in LDR)`
  - `add offset to PC (or to part of PC)`
  - `add offset to zero`





Instruction Processing: EVALUATE ADDRESS

Load / Store

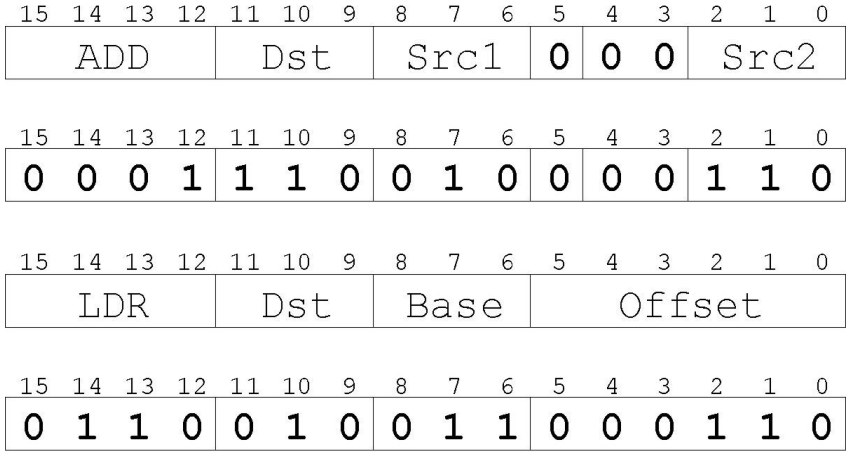


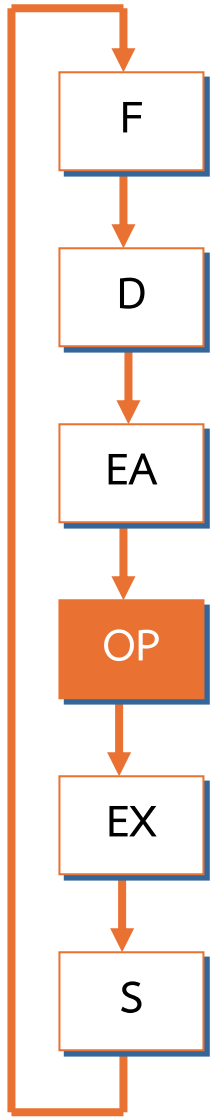
→ Control

— Data

Instruction Processing: FETCH OPERANDS

- Obtain source operands needed to perform operation.
- Examples:
  - `read data from register file (ADD)`
  - `load data from memory (LDR)`



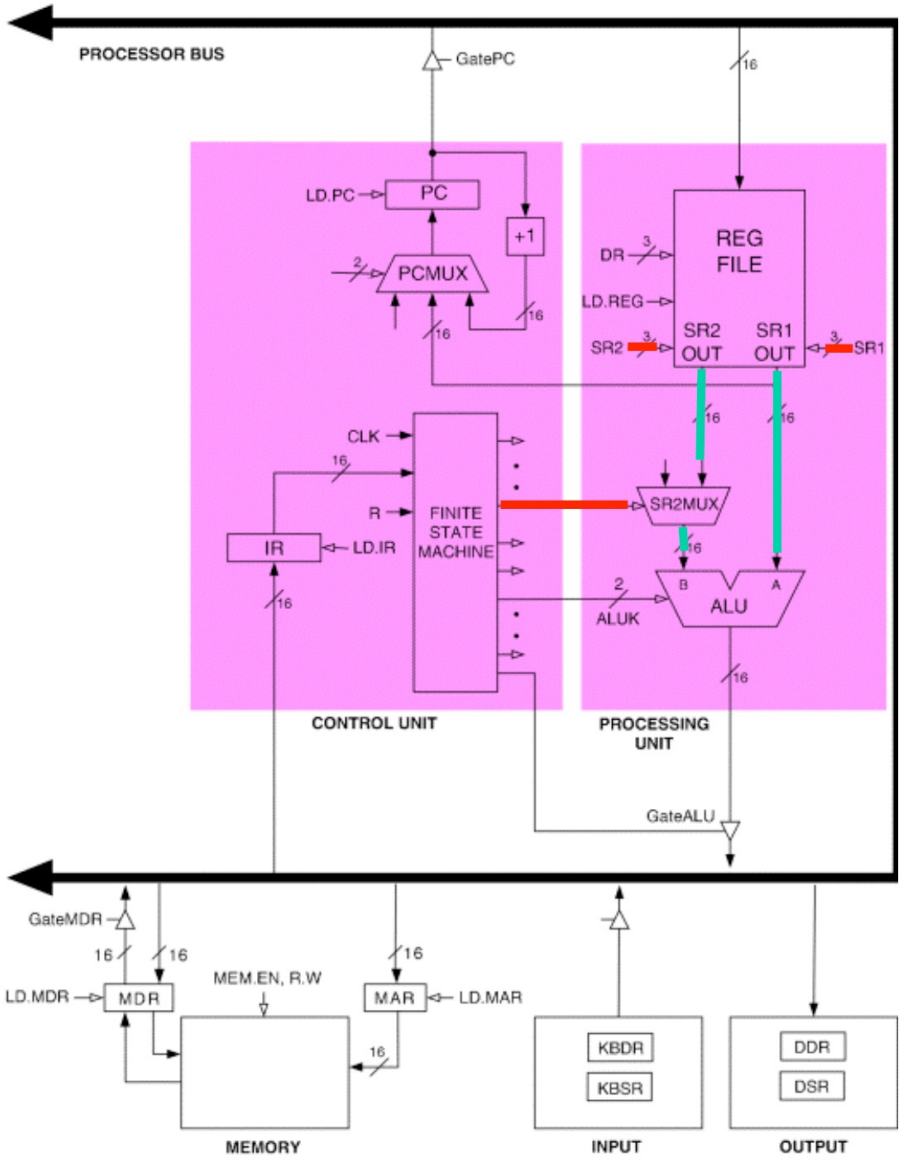


Instruction Processing: FETCH OPERANDS

ADD

→ Control

— Data

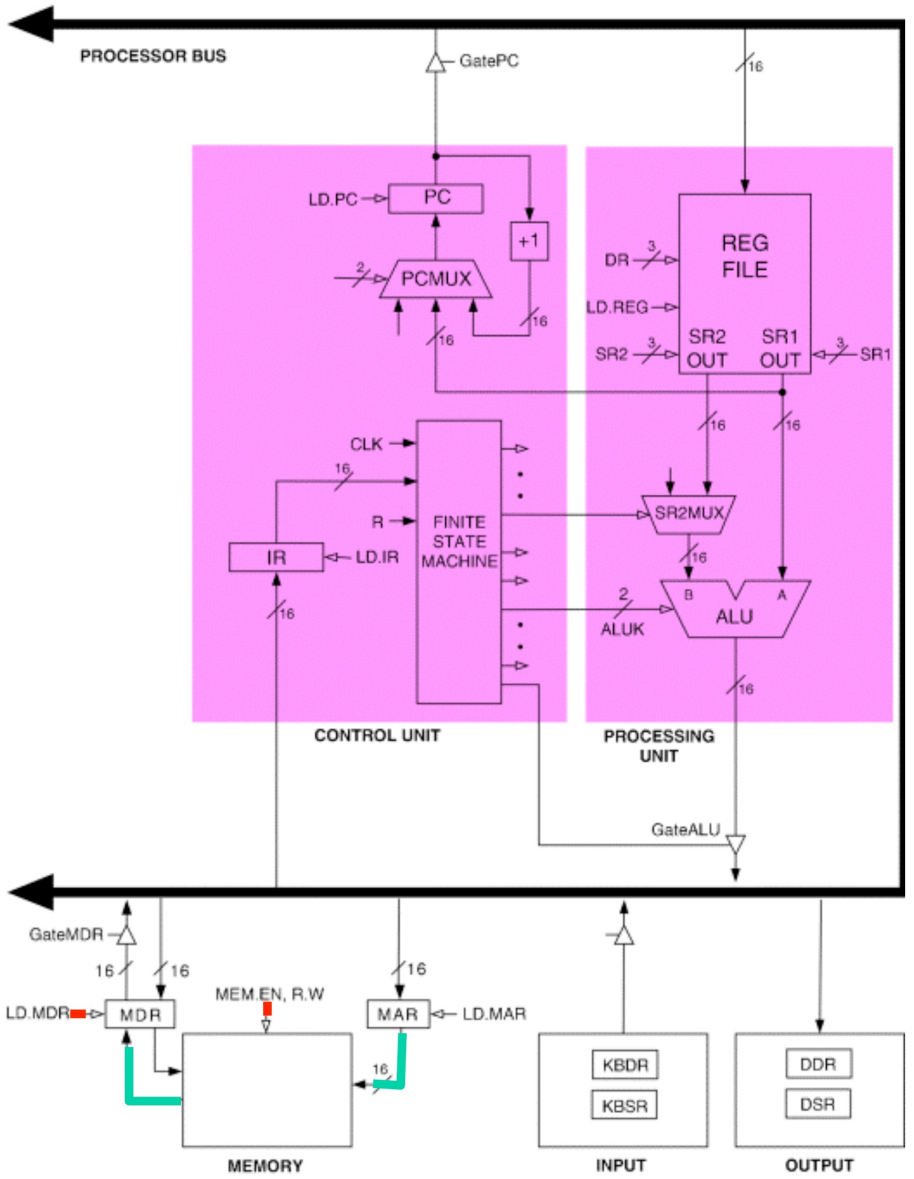


Instruction Processing: FETCH OPERANDS

LDR

→ Control

— Data

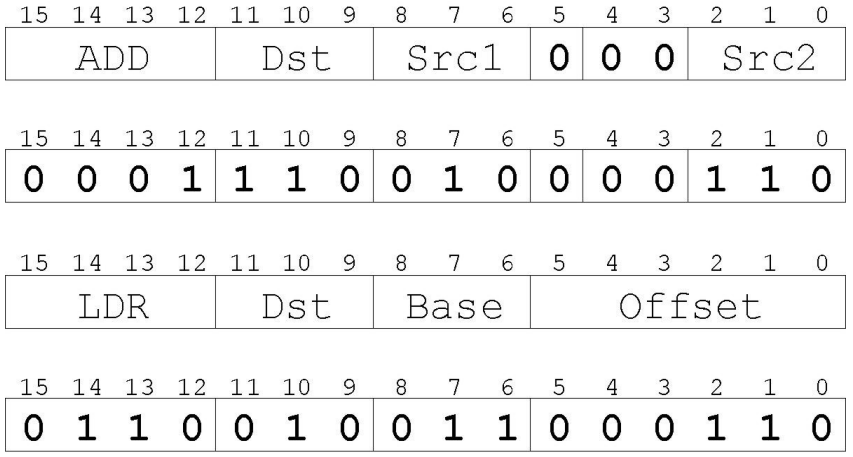


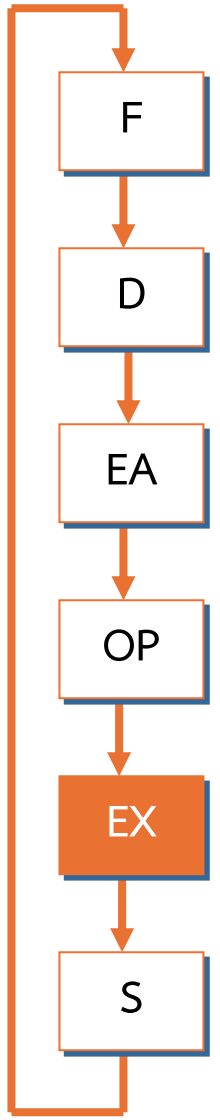
Instruction Processing: EXECUTE

Perform the operation, using the source operands.

Examples:

- send operands to ALU and assert ADD signal
- do nothing (e.g., for loads and stores)



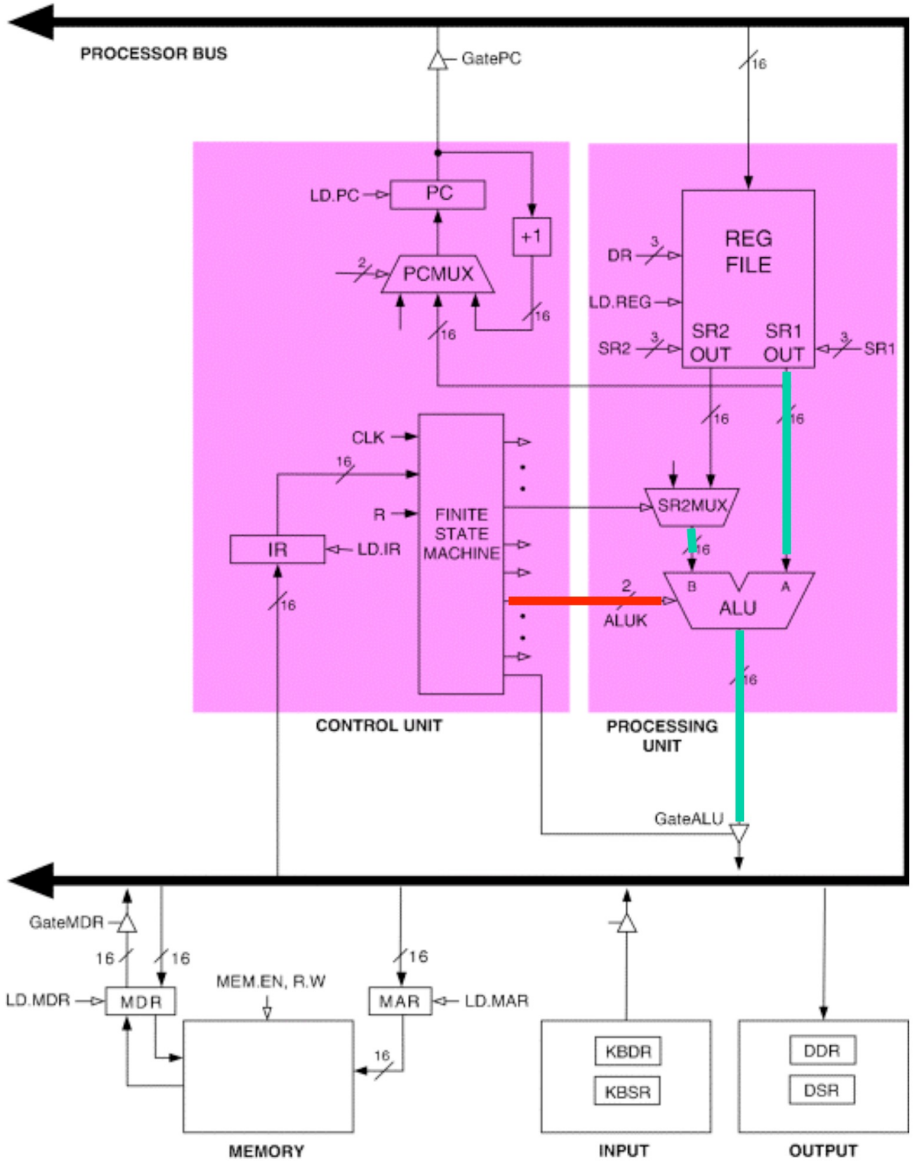


Instruction Processing: EXECUTE

ADD

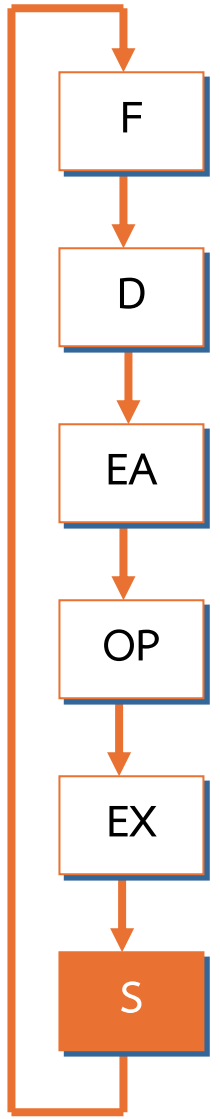
→ Control

→ Data



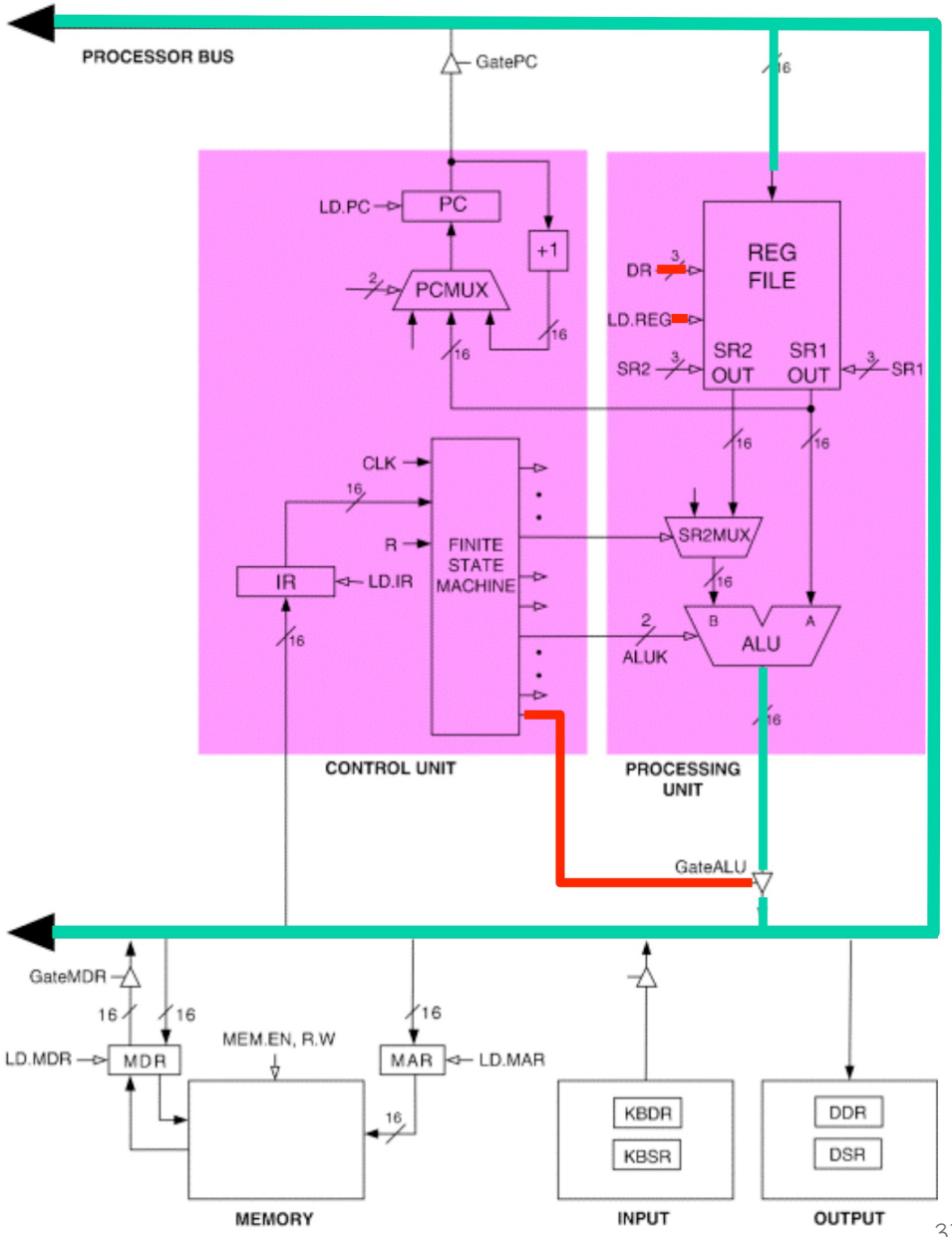
Instruction Processing: STORE

- Write results to destination. (register or memory)
- Examples:
  - result of ADD is placed in destination register
  - result of memory load is placed in destination register
  - for store instruction, data is stored to memory
    - write address to MAR, data to MDR
    - assert WRITE signal to memory



Instruction Processing: STORE

ADD



→ Control

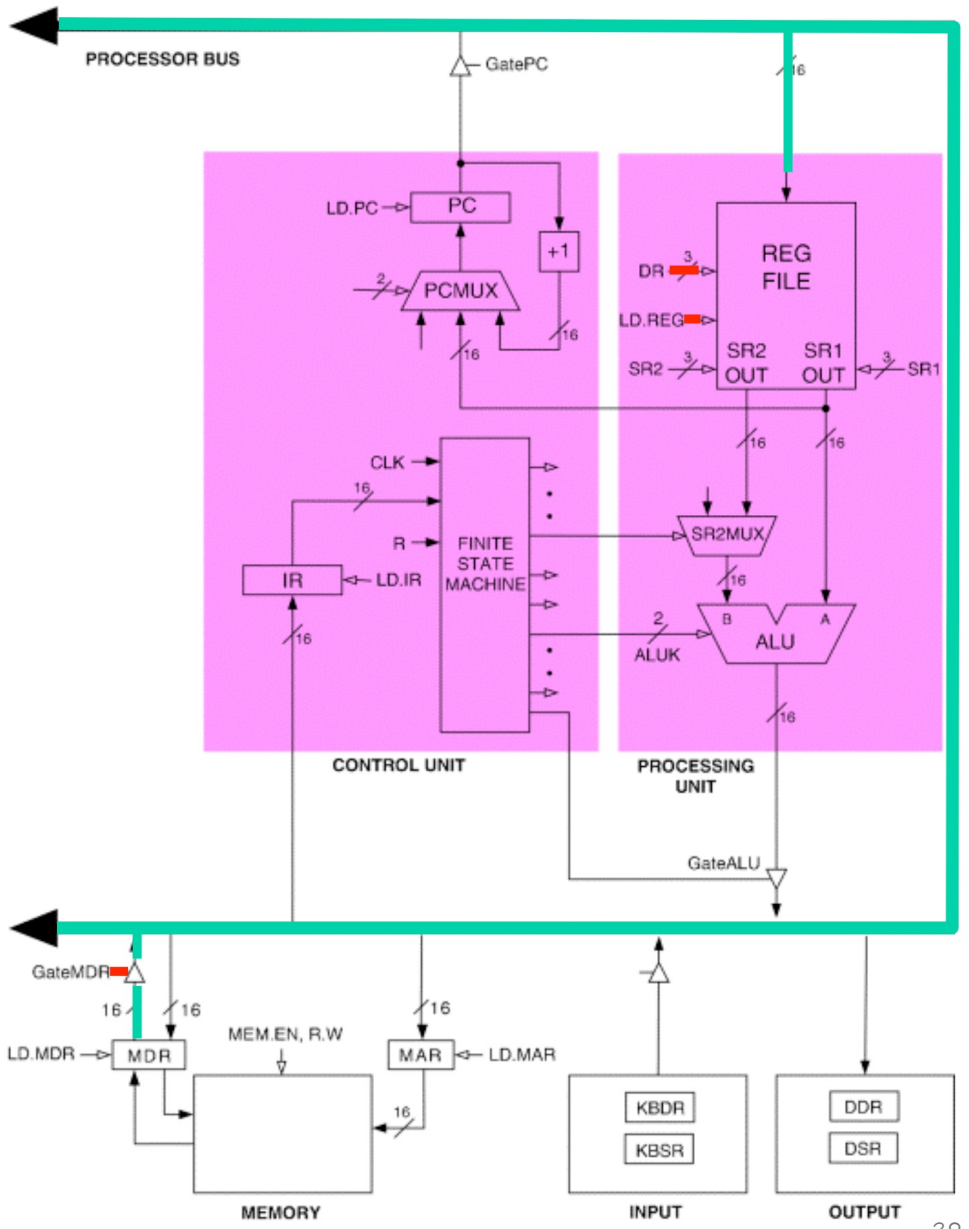
→ Data

Instruction Processing: STORE

LDR

→ Control

→ Data



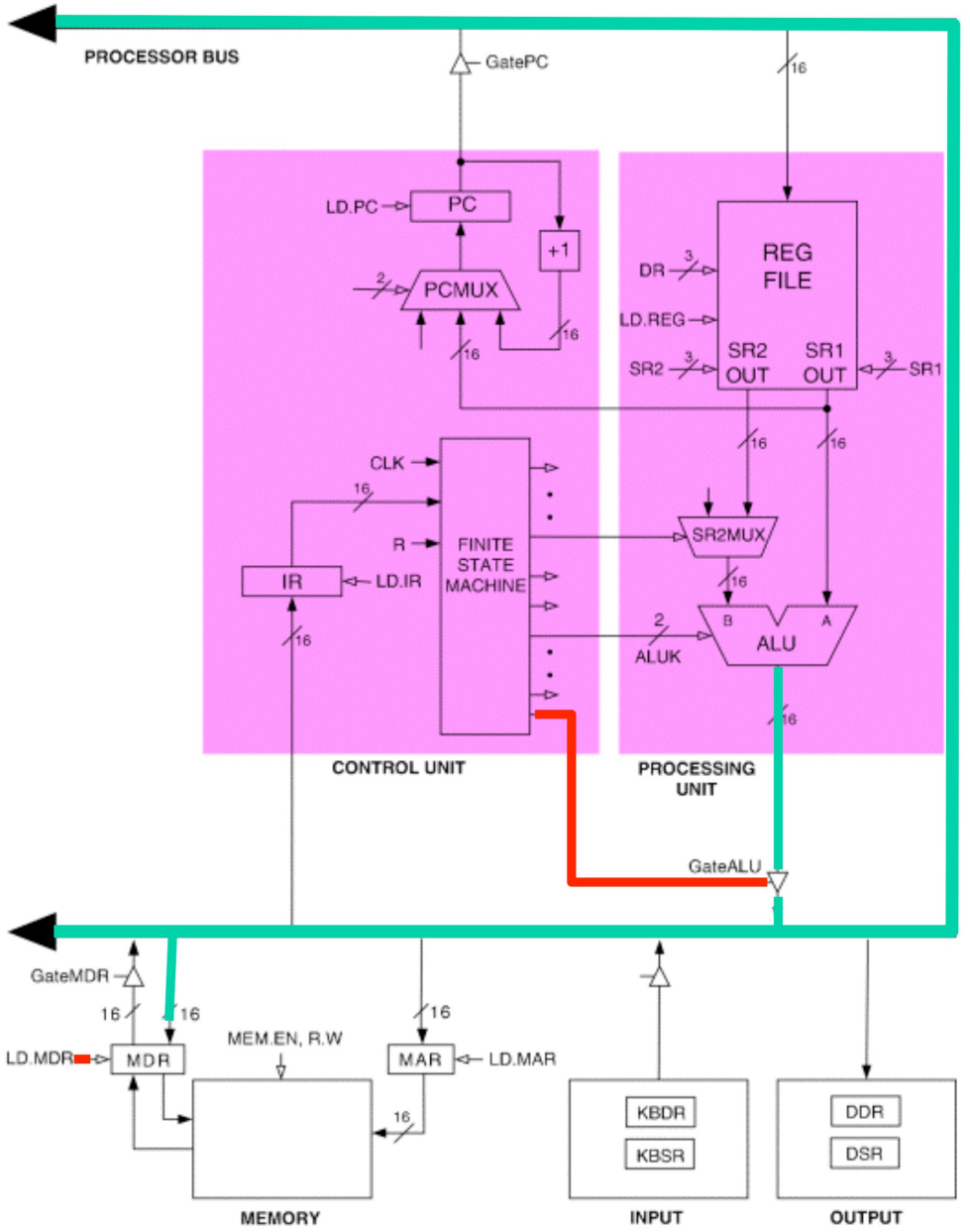
Instruction Processing: STORE

For store instruction, data is stored to memory

```
write address to MAR, data to MDR
assert WRITE signal to memory
```

→ Control

→ Data

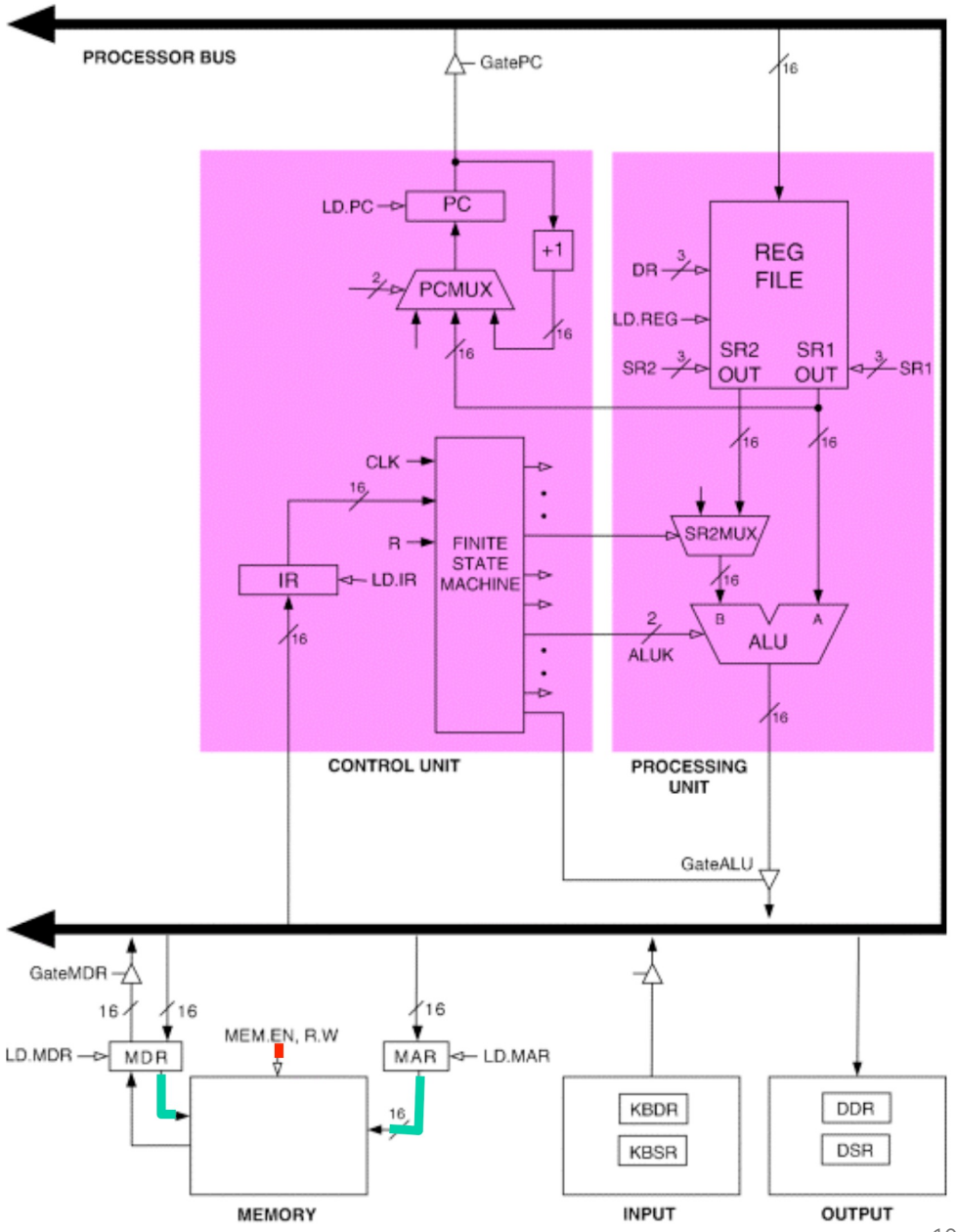


Instruction Processing: STORE

For store instruction, data is stored to memory

write address to MAR, data to MDR

assert WRITE signal to memory



→ Control

→ Data

Example: Intel x86 instruction

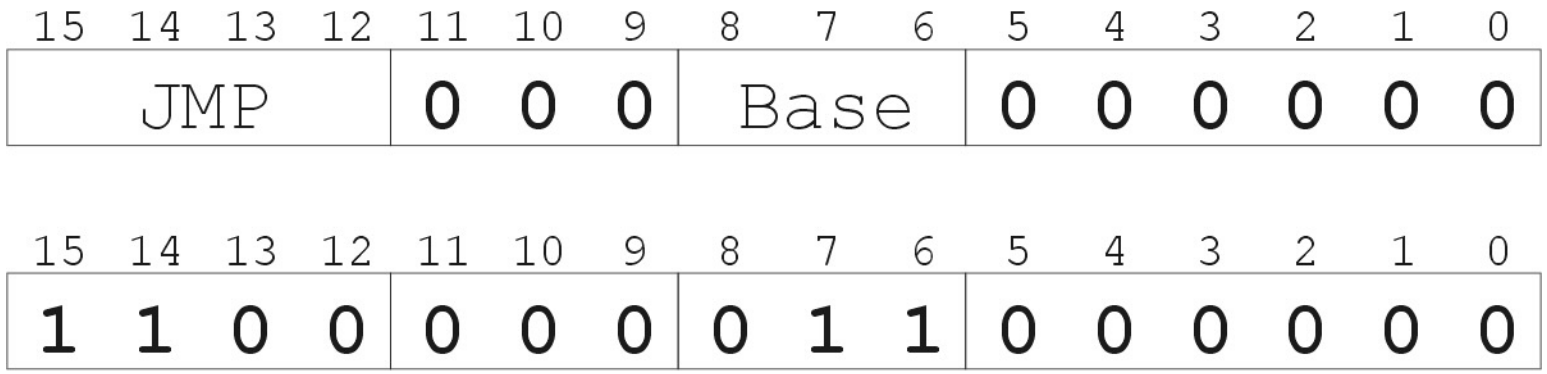
ADD [eax], edx This is an example of an Intel x86 instruction that requires all six phases of the instruction cycle. All instructions require the first two phases, FETCH and DECODE. This instruction uses the eax register to calculate the address of a memory location (EVALUATE ADDRESS). The contents of that memory location is then read (FETCH OPERAND), added to the contents of the edx register (EXECUTE), and the result written into the memory location that originally contained the first source operand (STORE RESULT).

Changing the Sequence of Instructions

- In the **FETCH** phase, we incremented the **PC by 1**.
- What if we don't want to always execute the instruction that follows this one?
  - Examples: loop, if-then-else, function call
- **Need special instructions that change the contents of the PC**
- Called *jumps* and *branches*
  - `jumps are unconditional -- they always change the PC`
  - `branches are conditional -- they change the PC only if some condition is true (e.g., the contents of a register is zero)`

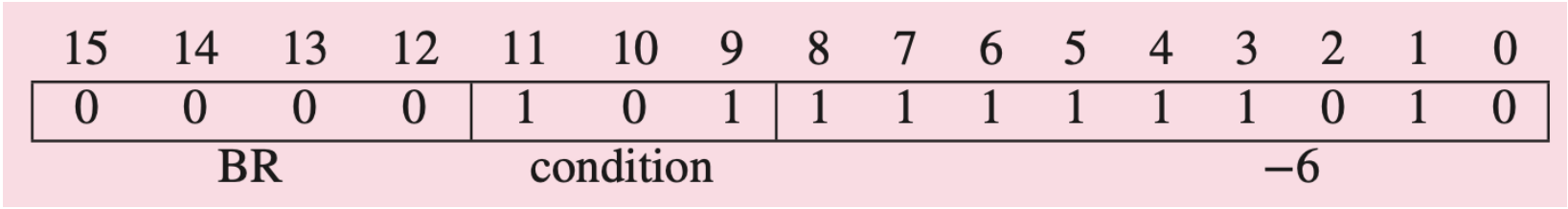
Example: LC-3 JMP Instruction

- Set the PC to the value contained in a register. This becomes the address of the next instruction to fetch



Load the contents of R3 into the PC

Example: LC-3 BR Instruction



- Opcode 0000 → the instruction as a conditional branch
- Condition `101` mean "Is the result not zero?"
- Assume the **previous instruction** executed (in memory location **x36C8**) was an ADD instruction (now PC = **x36C9**)
  - if the result of the ADD is 0:

    test "not-zero" failed → BR instruction would do nothing, next instruction **x36CA**
  - if the result of the ADD is **not 0**:

    test succeeds → BR instruction to load PC with **x36C4** (PC + (-6))

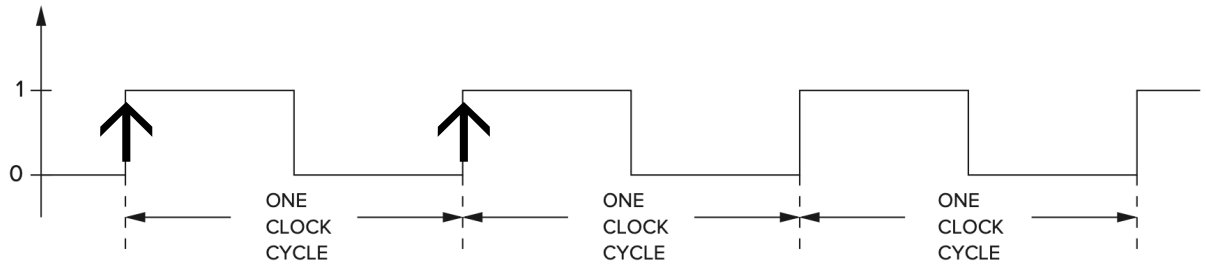
Control of the Instruction Cycle

The clock is a signal that keeps the control unit moving.

- At each clock “tick” control unit moves to the next machine cycle -- may be next instruction or next phase of current instruction.

Clock generator circuit:

- Based on crystal oscillator
- Generates regular sequence of “0” and “1” logic levels
- Clock cycle (or machine cycle) -- **rising edge** to **rising edge**

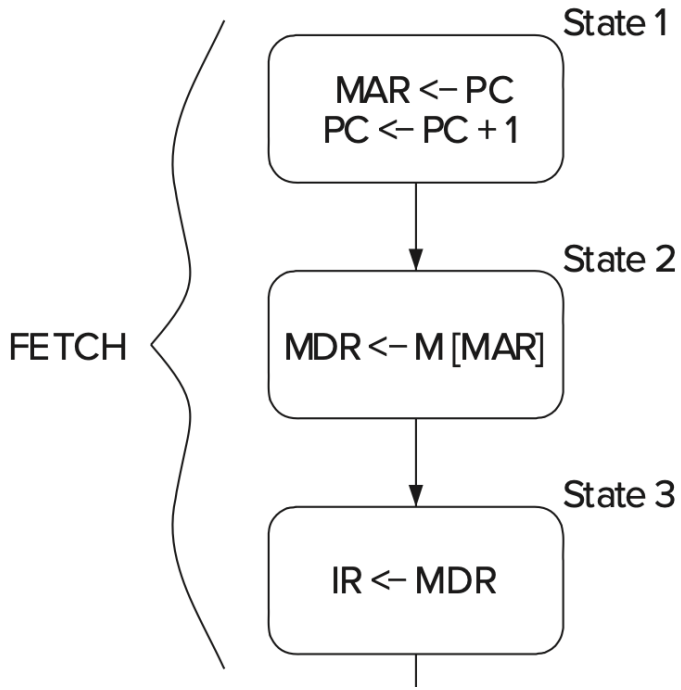


Processing Instruction

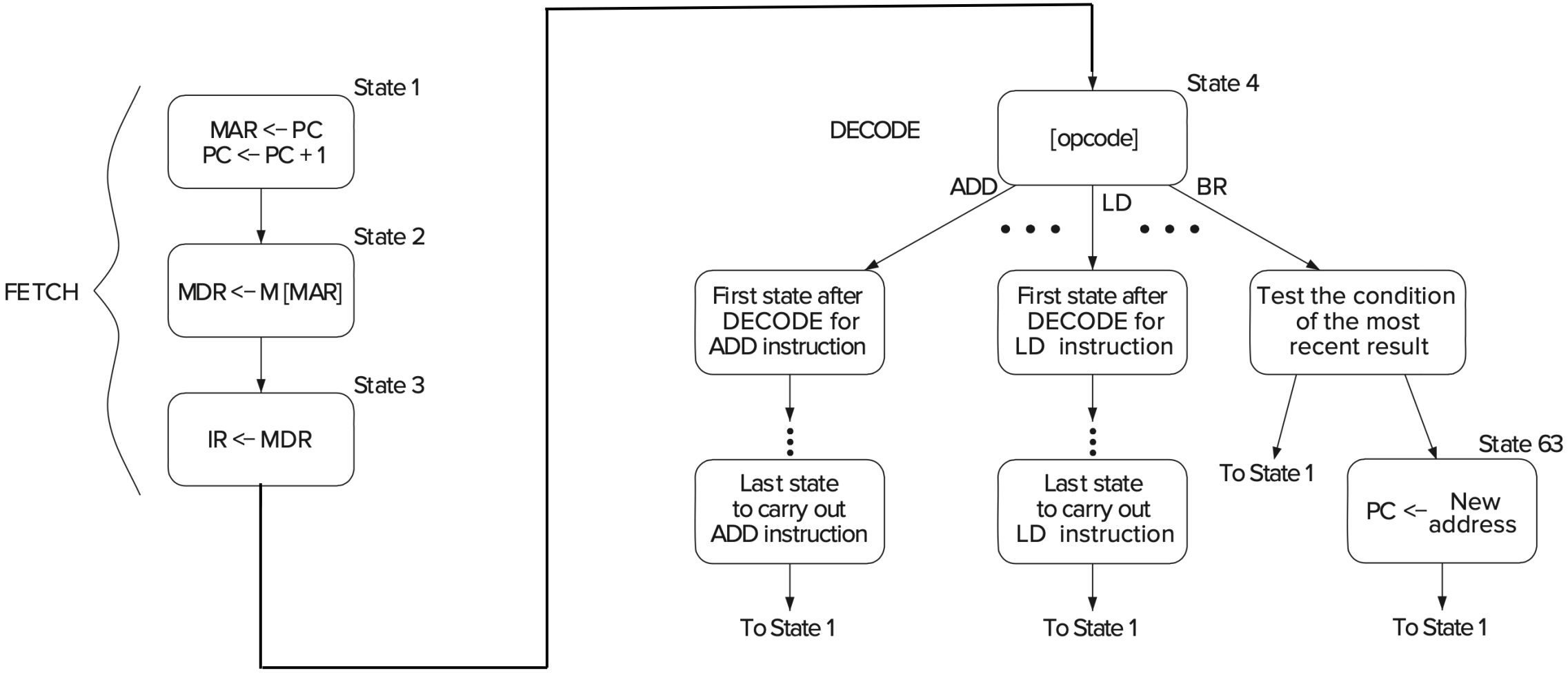
- The instruction cycle is controlled by a synchronous finite state machine
- State diagram
- Example:

  The FETCH phase may take three clock cycles

  The DECODE phase takes one clock cycle



Control Unit State Diagram



LC-3 35 states (Fig. C.2)

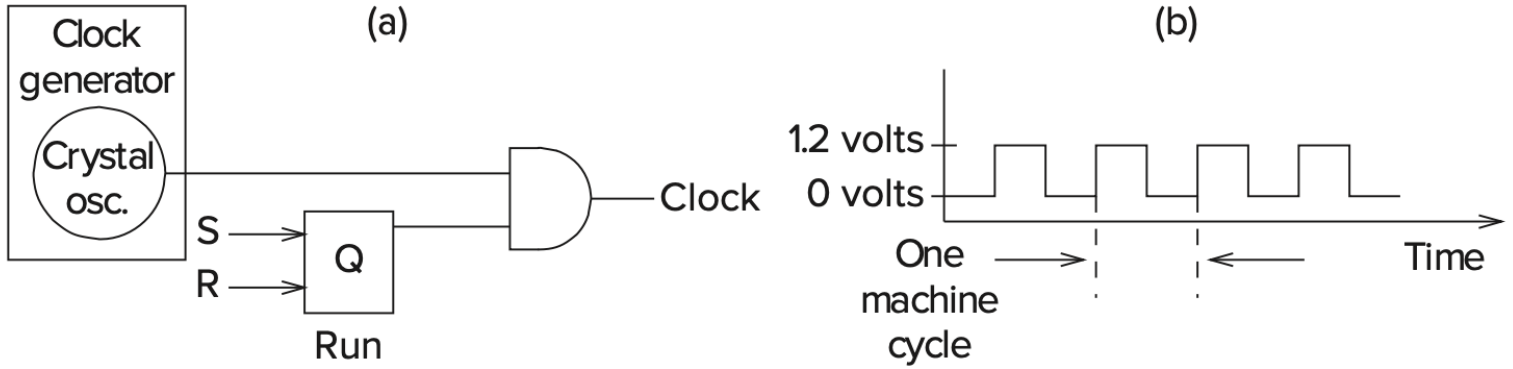
The clock circuit and its control

Control unit will repeat instruction processing sequence as long as clock is running.

- If not processing instructions from your application, then it is processing instructions from the Operating System (OS).
- The OS is a special program that manages processor and other resources.

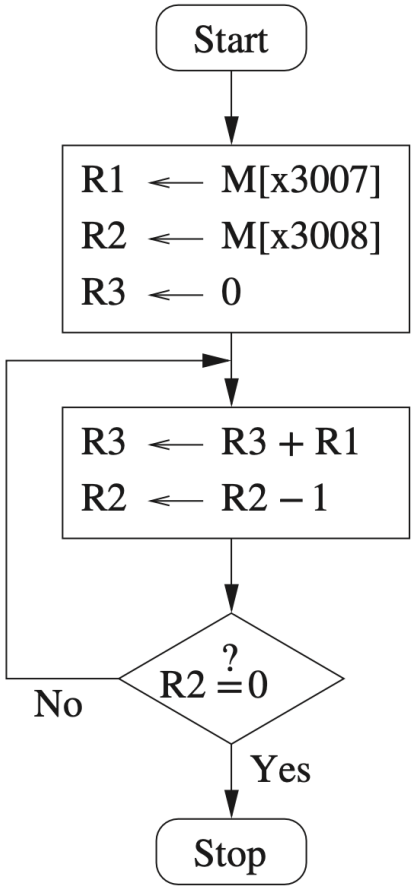
Halting the Computer (the TRAP Instruction, opcode = 1111):

- AND the clock generator signal with ZERO
- when control unit stops seeing the CLOCK signal, it stops processing



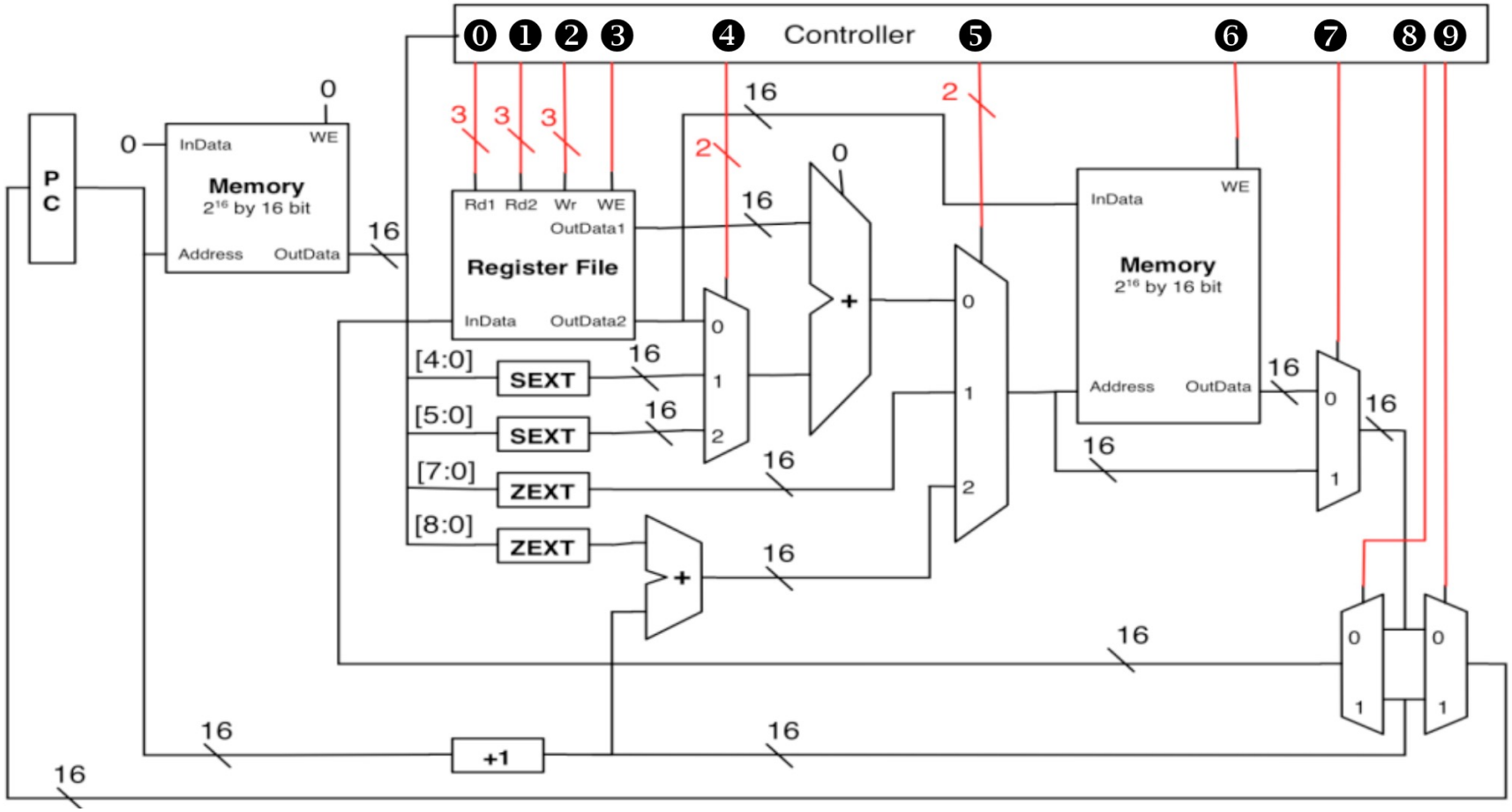
Our First Program: A Multiplication Algorithm

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x3000	0	0	1	0	0	0	1	0	0	0	0	0	0	1	1	0	`R1 <- M[x3007]`
x3001	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	`R2 <- M[x3008]`
x3002	0	1	0	1	0	1	1	0	1	1	1	0	0	0	0	0	`R3 <- 0`
x3003	0	0	0	1	0	1	1	0	1	1	0	0	0	0	0	1	`R3 <- R3+R1`
x3004	0	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	`R2 <- R2-1`
x3005	0	0	0	0	1	0	1	1	1	1	1	1	1	1	0	1	`BR not-zero M[x3003]`
x3006	1	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	`HALT`
x3007	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	`The value 5`
x3008	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	`The value 4`

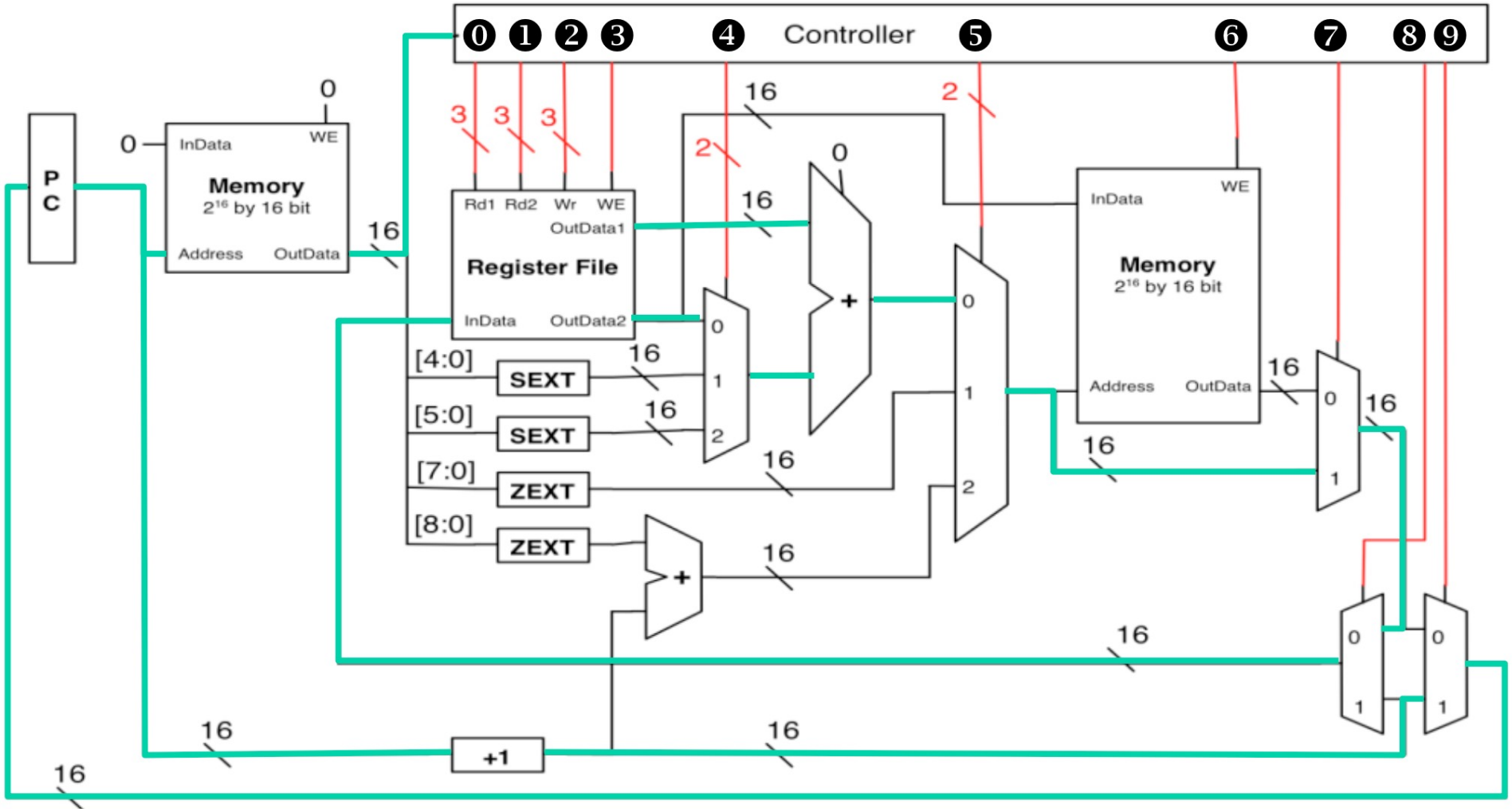


Alternate Implementation

- Execute Each Instruction in Single Cycle
  - Much simpler
  - All phases happen in one cycle

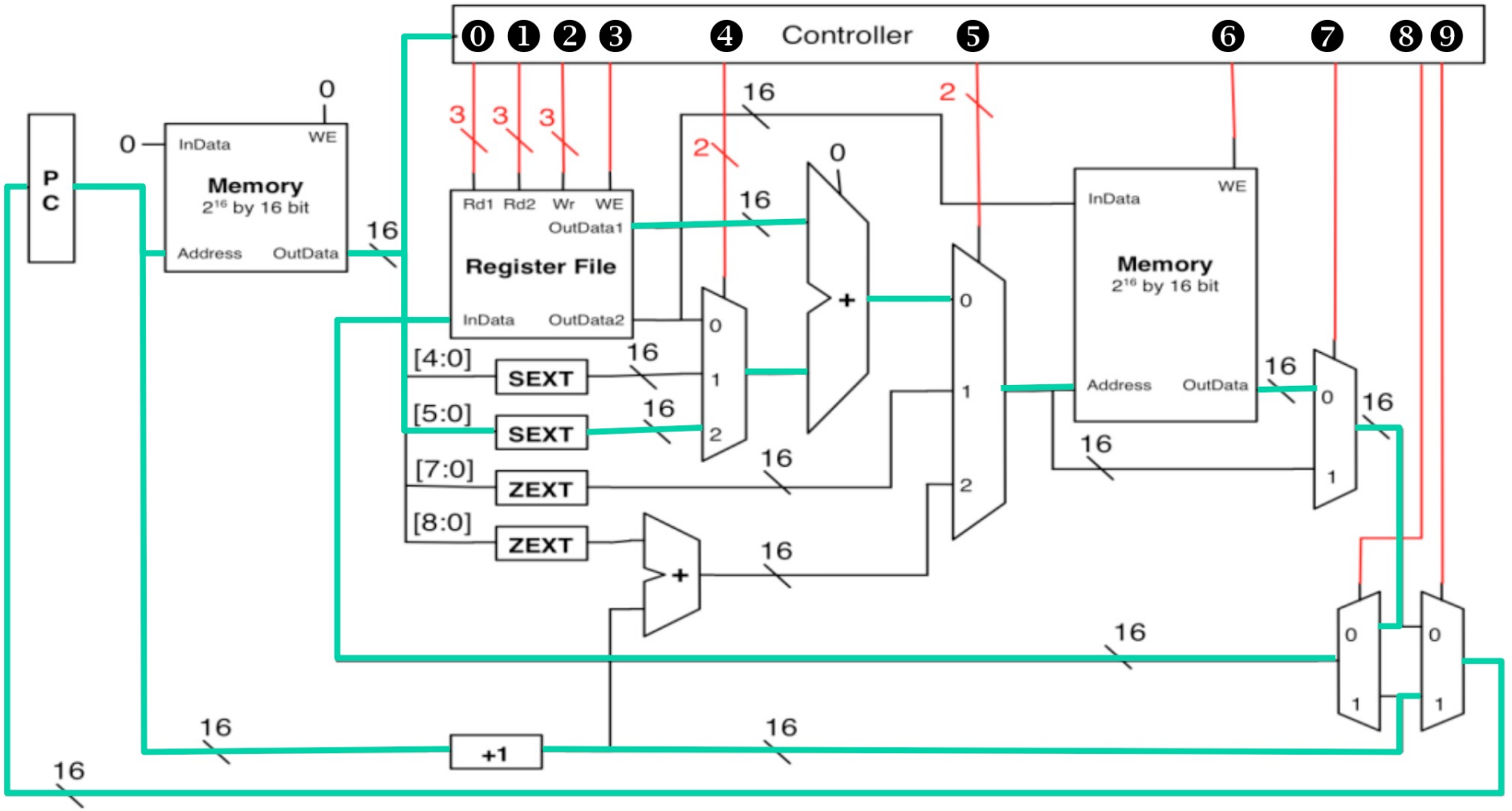


Single-Cycle Implementation (ADD)



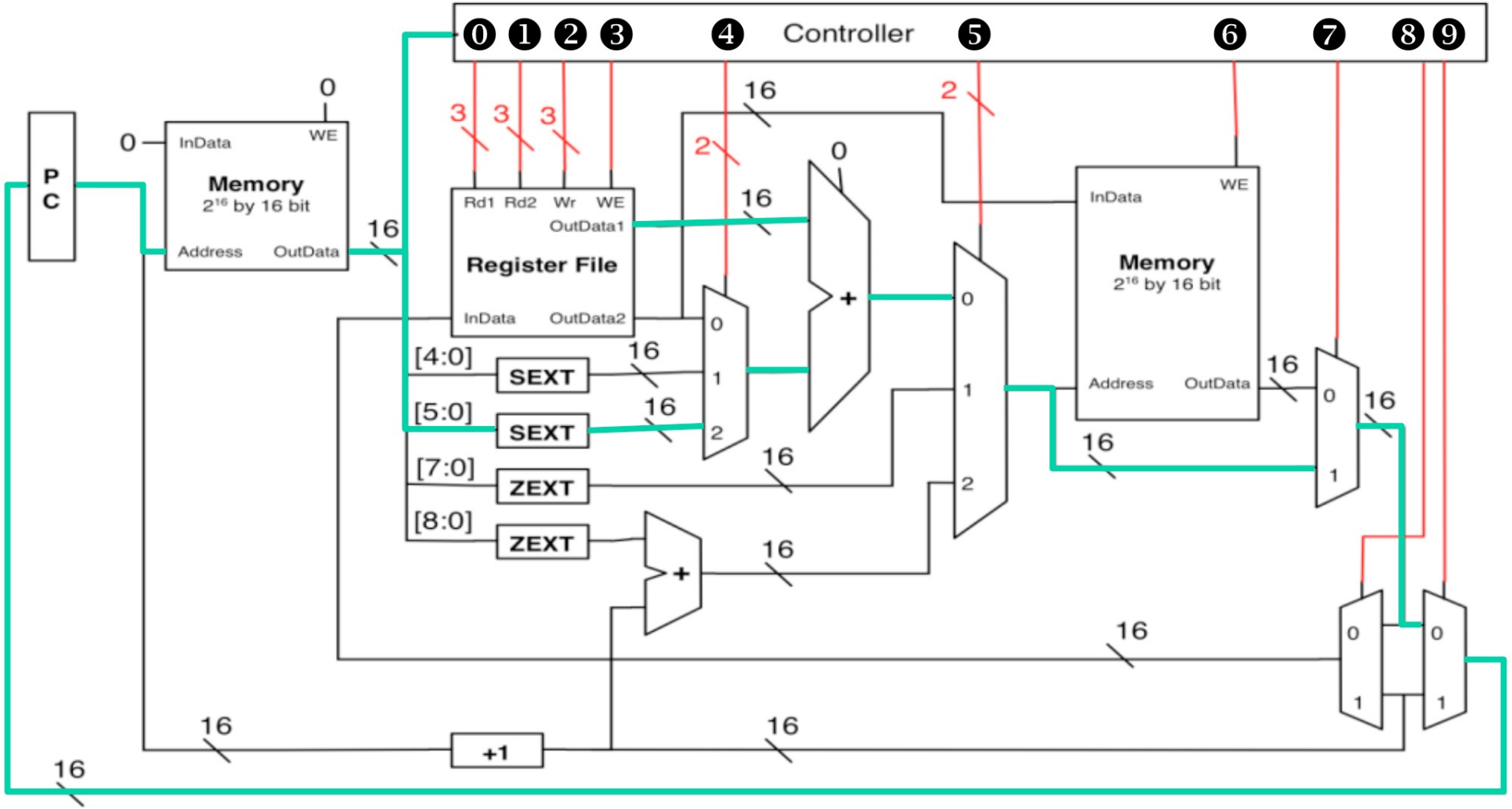
	Opcode		Control									
Instr	I[15:12]	I[5]	0	1	2	3	4	5	6	7	8	9
ADD	0001	0	I[8:6]	I[2:0]	I[11:9]	1	00	00	0	1	0	1

Single-Cycle Implementation (LDR)



	Opcode		Control									
Instr	I[15:12]	I[5]	❶	❶	❷	❸	❹	❺	❻	❼	❽	❾
LDR	0110	-	I[8:6]	-	I[11:9]	1	10	00	0	0	0	1

Single-Cycle Implementation (JMP)



	Opcode		Control									
Instr	I[15:12]	I[5]	❶	❷	❸	❹	❺	❻	❼	❽	❾	
JMP	1100	-	I[8:6]	-	-	0	10	00	0	1	x	0

Single-Cycle Implementation Limitations

- All instructions given same time to execute
  - Cycle time must be time required for longest instruction
  - JMP vs ADD vs LDR
- Requires multi-ported memory
  - Separate buses between each CPU and each memory module
  - Allow multiple processors to access memory simultaneously
  - Ex: two read ports could allow fetching the instruction and reading the operand in the same cycle

Homework

- Exercises:

  4.1, 4.5, 4.7, 4.12, 4.14, 4.19