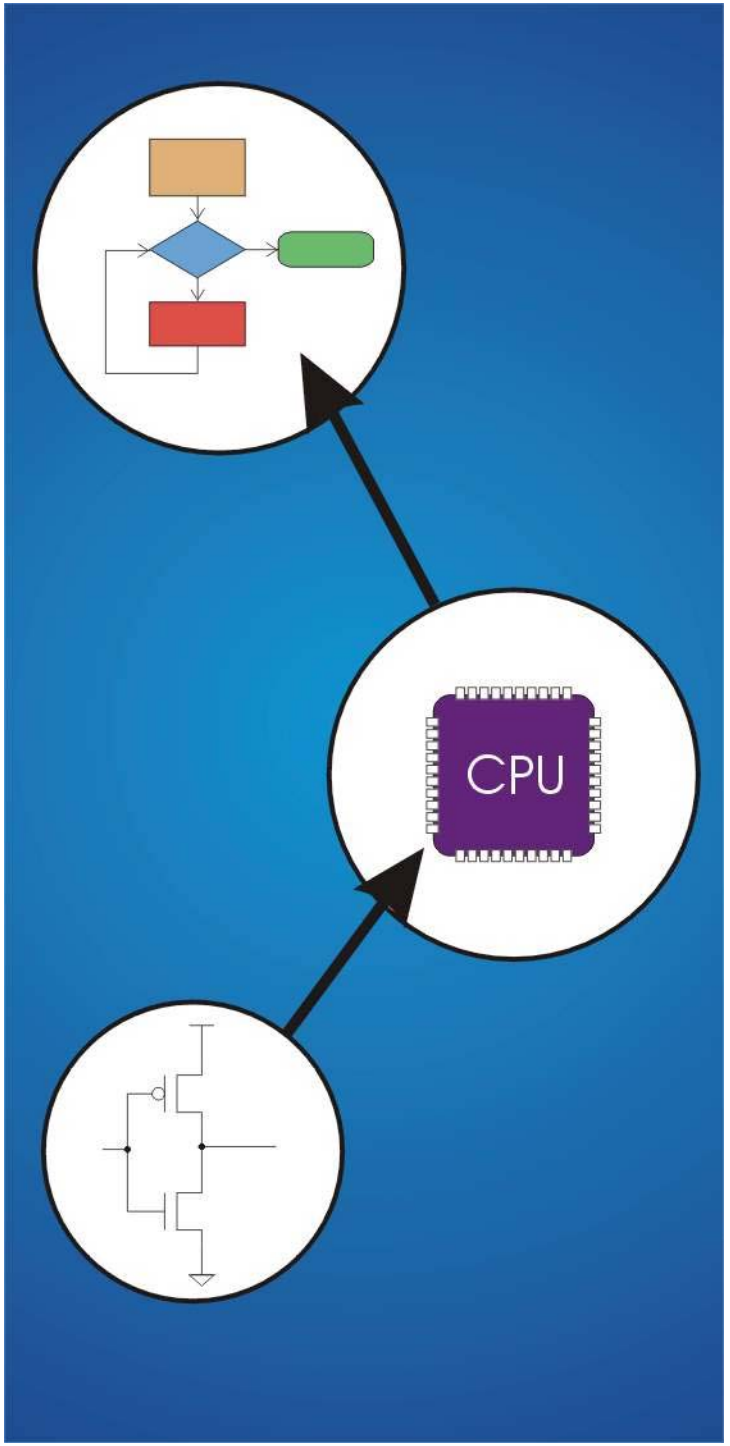
ECEN 324 – BYU-Idaho

Computer Architecture

Text: *Computer Systems: A Programmer's Perspective*

Randal E. Bryant and David O'Hallaron



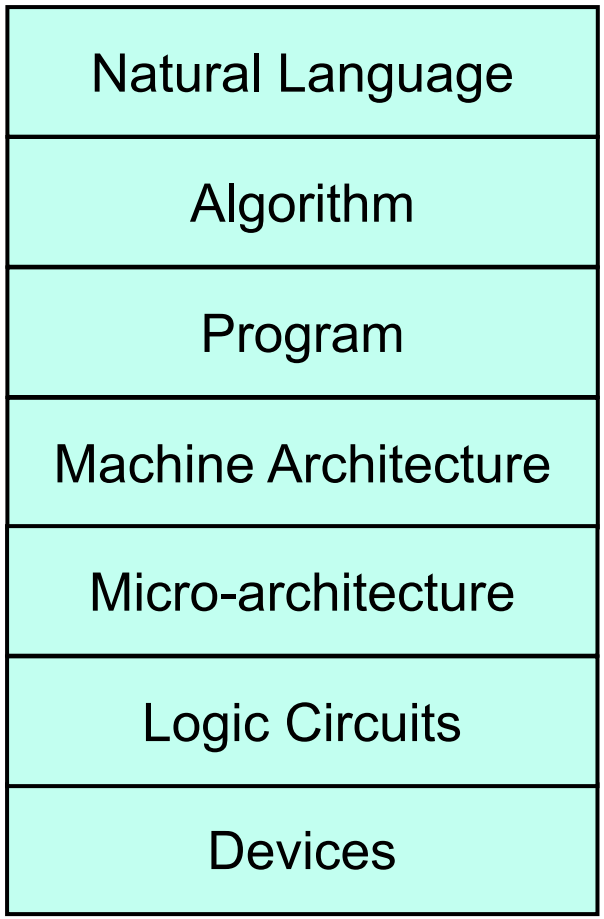
Introduction to Computing Systems

–from bits & gates to C & beyond

Yale N. Patt and
Sanjay J. Patel

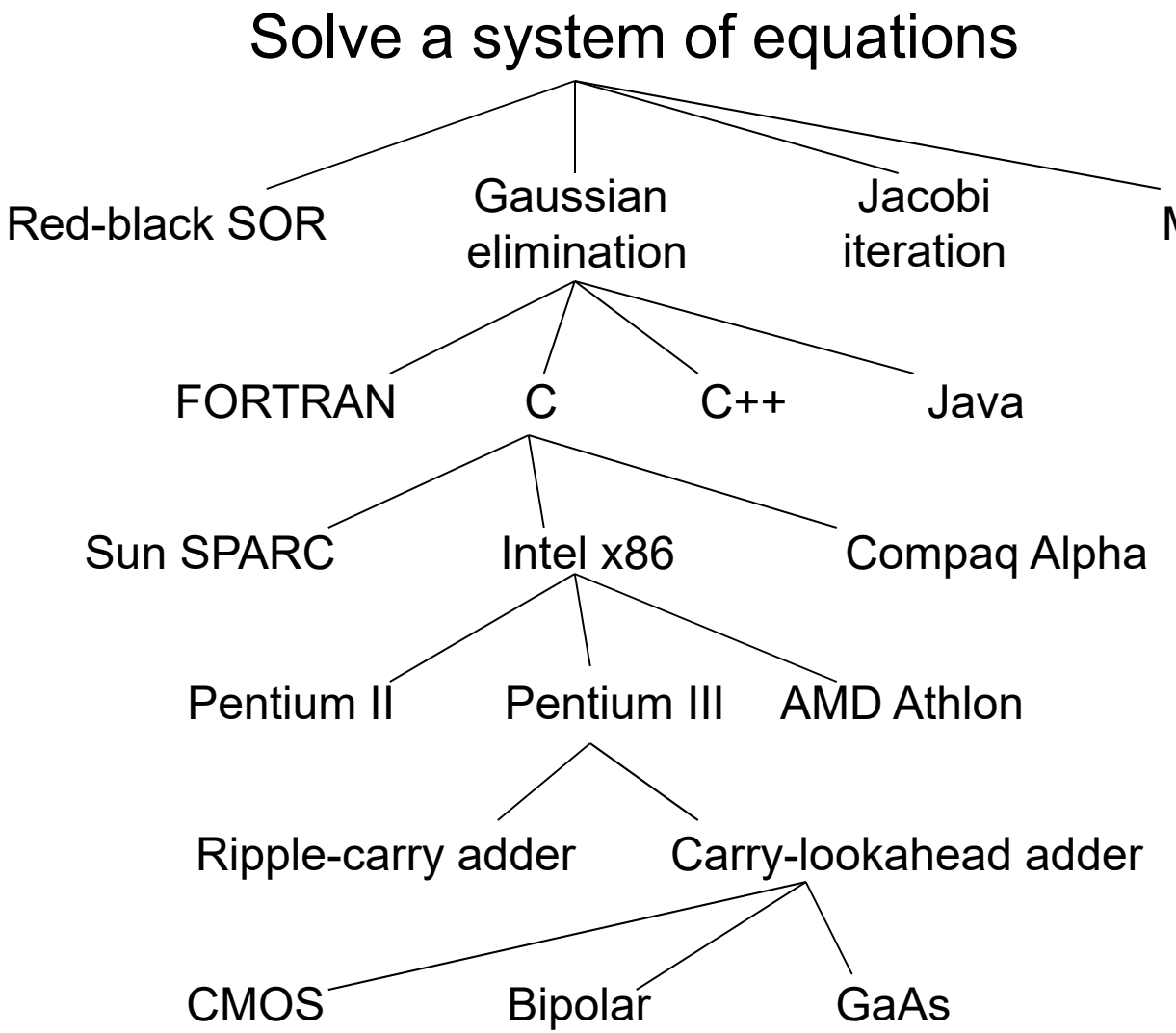
Problem Transformation
- finer breakdown

Desired Behavior: application



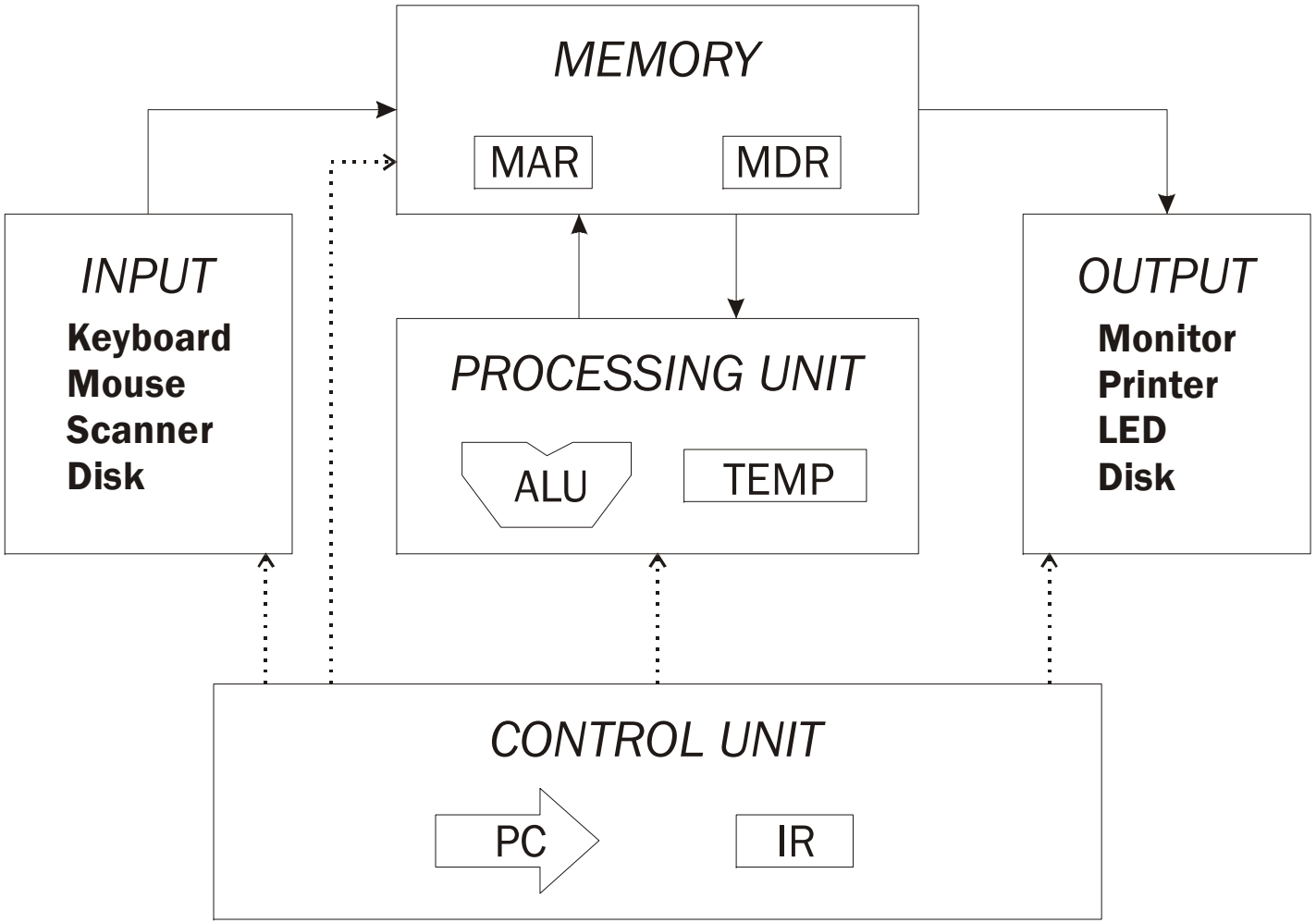
Raw Material: electronic devices

Many Choices at Each Level



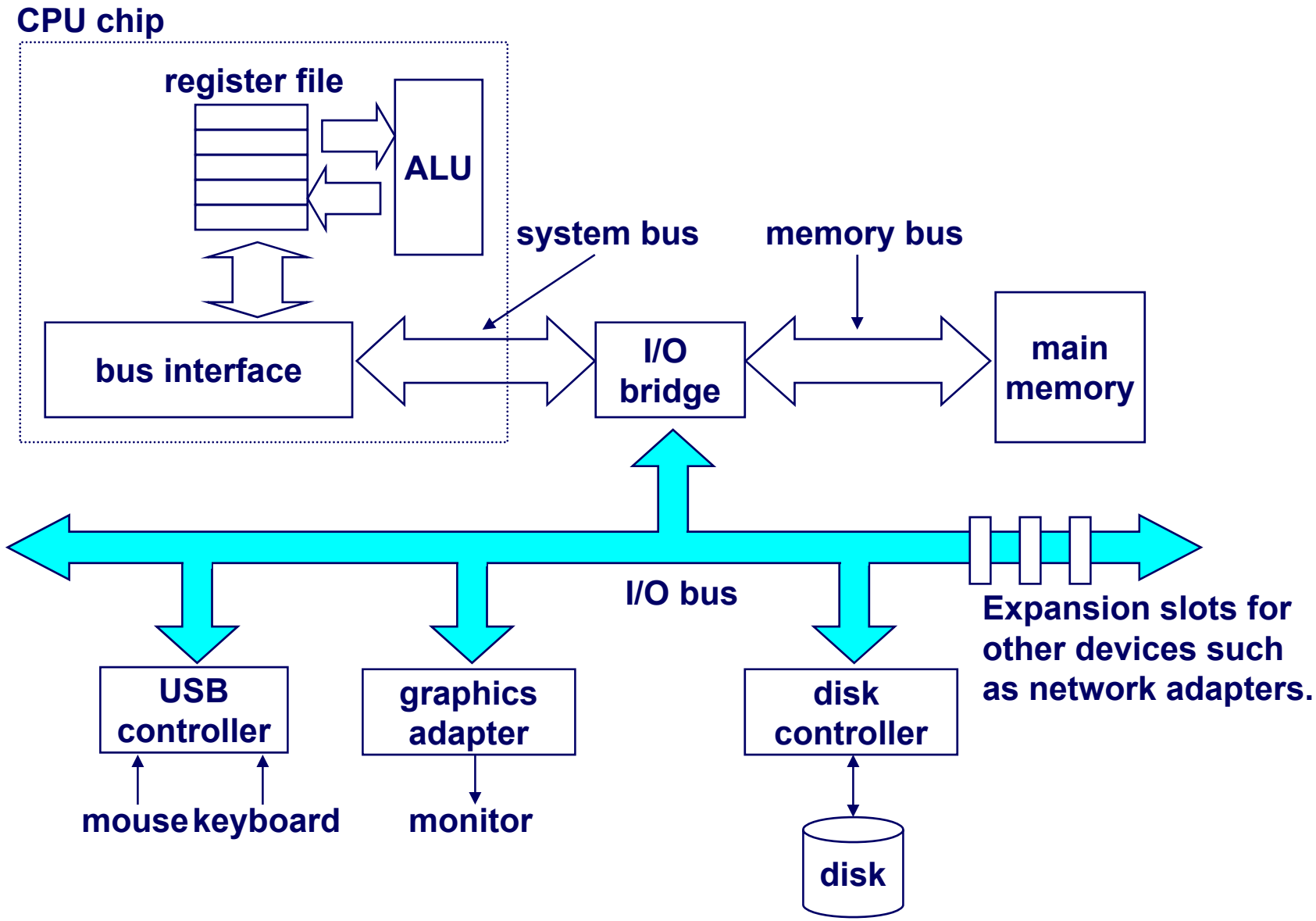
Tradeoffs:
cost
performance
functionality
power
(etc.)

Von Neumann Model



http://en.wikipedia.org/wiki/Von_Neumann_architecture

I/O Bus


CPU chip
register file
ALU
system bus
memory bus
bus interface
I/O bridge
main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
graphics adapter
disk controller
mouse
keyboard
monitor
disk

Memory

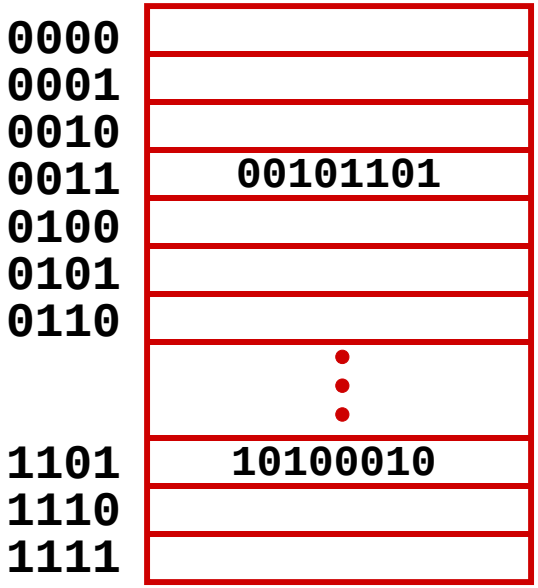
k x m array of stored bits (k is usually 2^n)

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**

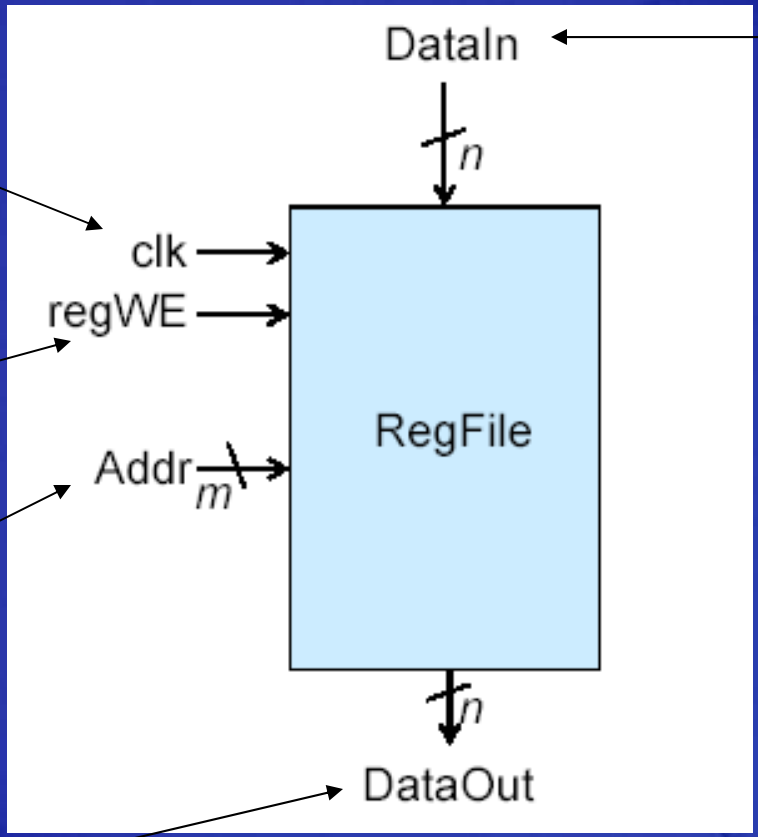
BRIGHAM YOUNG UNIVERSITY IDAHO

A Typical Register File (a type of memory device)

Writes occur on the clock edge (are synchronous), reads are asynchronous

Controls writing of locations (words)

Address that determines which location (word) to read and write; m bits wide



Data to be written into a location (word) in the register file; n bits per word

This register holds 2^m words (or has 2^m locations), each n bits wide

*The register capacity (total number of bits it can hold) is 2^m * n bits; or equivalently: number of words * width of a word*

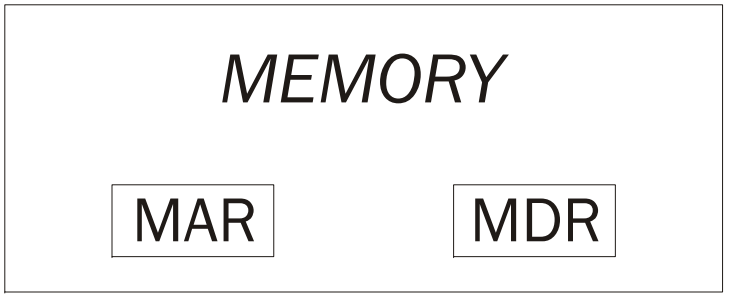
Data that is read from a register file location (word); n bits per word (n bits wide)

Interface to Memory

How does processing unit get data to/from memory?

MAR: Memory Address Register

MDR: Memory Data Register



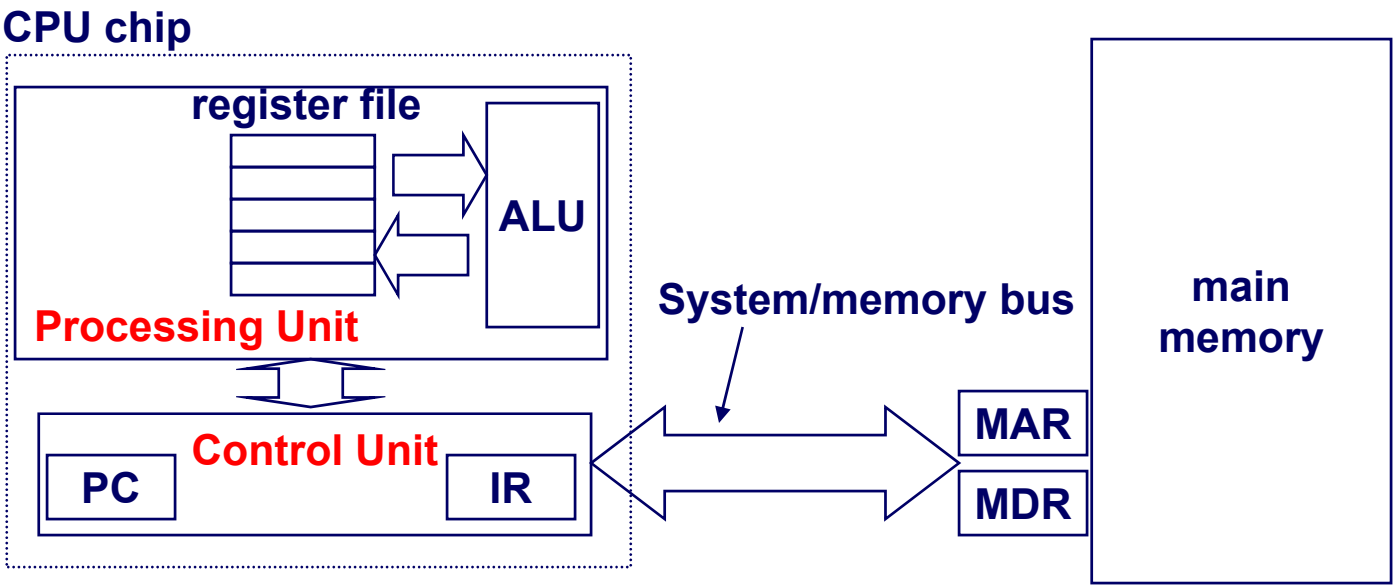
To read a location (A):

1. **Write the address (A) into the MAR.**
2. **Send a "read" signal to the memory.**
3. **The memory subsystem reads the data it has stored at the address location specified and puts the data into the MDR**
4. **Read the data from MDR.**

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

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

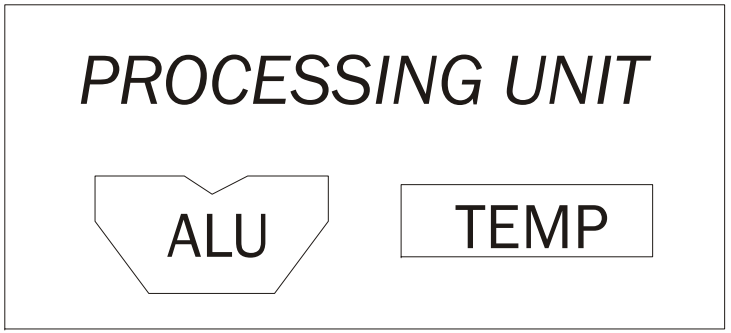
CPU and Memory



Processing Unit

Functional Units

- ALU = Arithmetic and Logic Unit
- could have many functional units. some of them special-purpose (multiply, square root, …)
- LC-2 performs ADD, AND, NOT



Registers

- Small, fast, temporary storage that is not part of “main memory”
- Operands and results of functional units
- LC-2 has eight register (R0, …, R7)

Word Size

- number of bits normally processed by ALU in one instruction
- also width of registers
- LC-2 is 16 bits

https://en.wikipedia.org/wiki/Comparison_of_instruction_set_architectures

Input and Output

Devices for getting data into and out of computer memory

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

Each device has its own interface, usually a set of registers like the memory's MAR and MDR

- **LC-2 supports keyboard (input) and console (output)**
- **keyboard: data register (KBDR) and status register (KBSR)**
- **console: data register (CRTDR) and status register (CRTSR)**

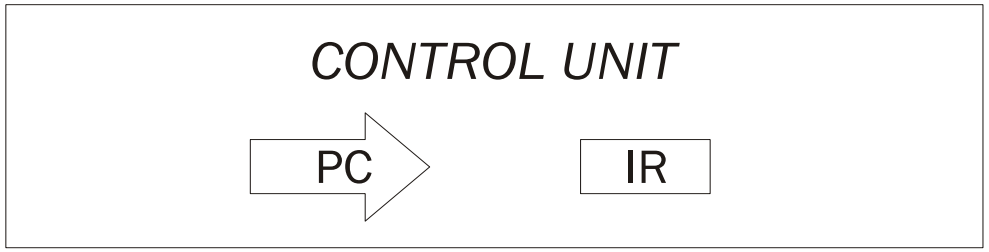
Some devices provide both input and output

- **disk, network**

Program that controls access to a device is usually called a *driver*.

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 (fetches an instruction)**
  - **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**

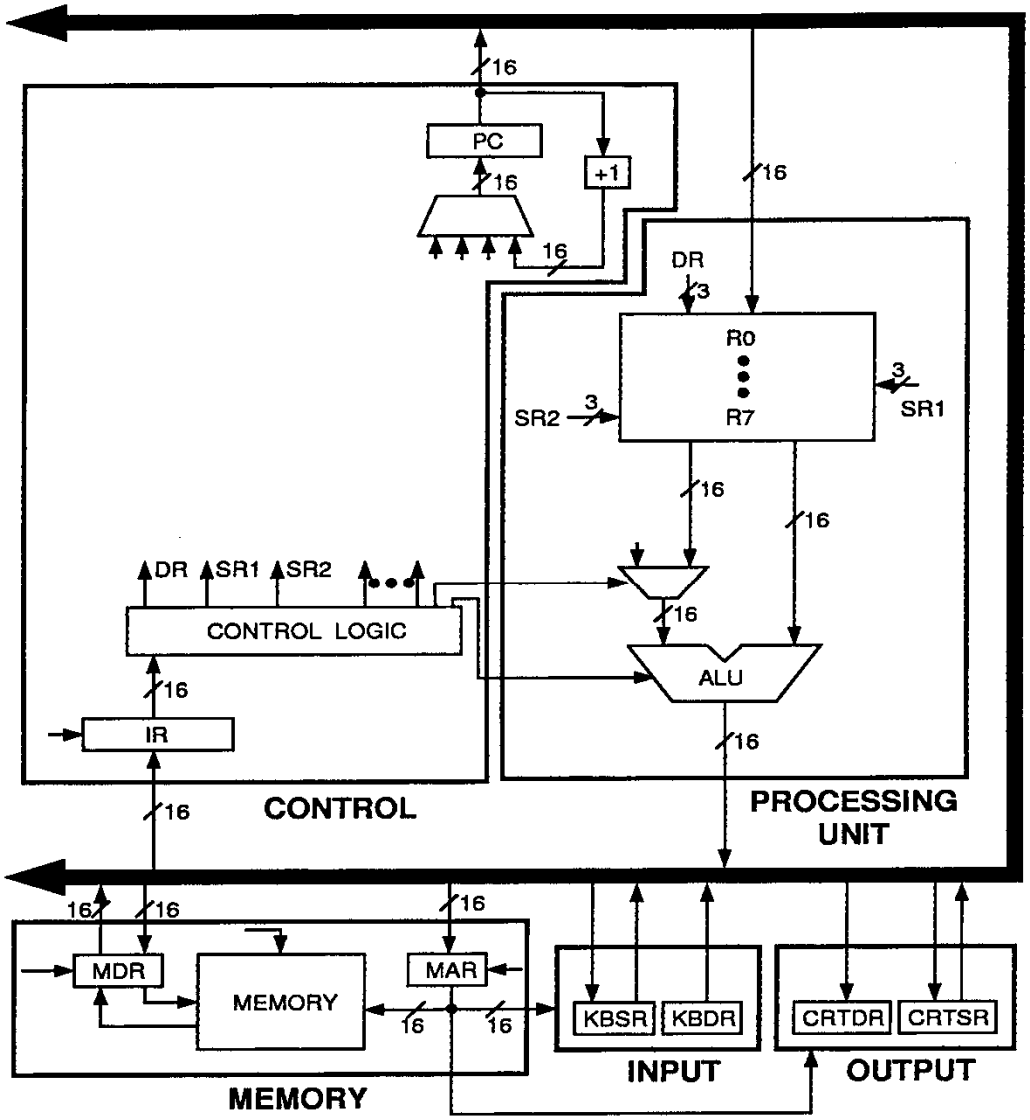
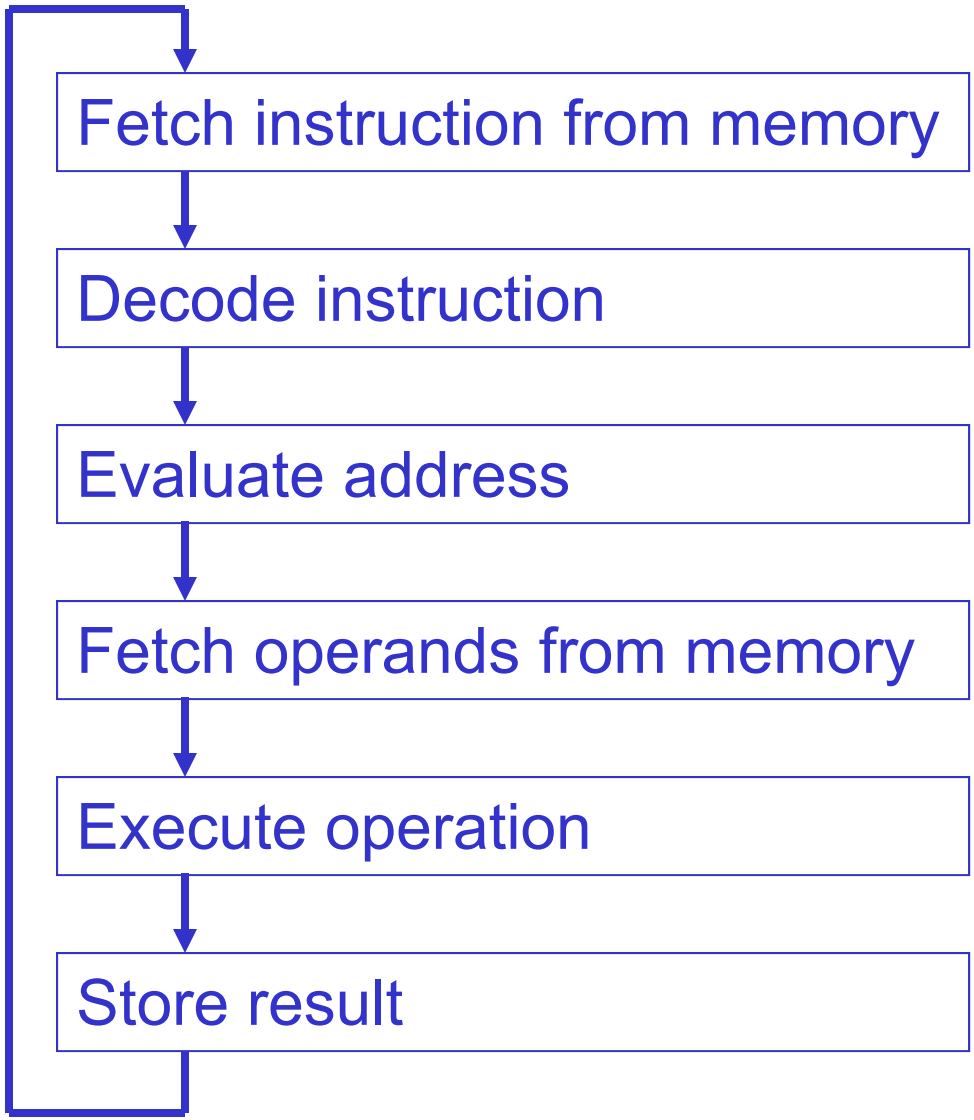


Figure 4.3: The LC-2 as an Example of the Von Neumann Model.

Instruction Processing



Instruction

The instruction is the fundamental unit of work.

Specifies two things:

- ***opcode*: operation to be performed**
- ***operands*: data/locations to be used for operation**

An instruction is encoded as a sequence of bits.
(Just like data!)

- Often, but not always, instructions have a fixed length, such as 16 or 32 bits.
- Control unit interprets instruction: generates sequence of control signals to carry out operation.
- Operation is either executed completely, or not at all.

A CPU's instructions and their formats is known as its *Instruction Set Architecture (ISA)***.**

Operands

Operands may be …

- **The name of a register**
  - **Eight registers in LC-2 named 0 to 7 (000_2 to 111_2)**
- **A memory address**
- **A literal or immediate value**
- **The name of a register whose contents will point us to where in memory the data we want is currently residing**
- **…**

Operands tell us "who" the instruction is to do something to.

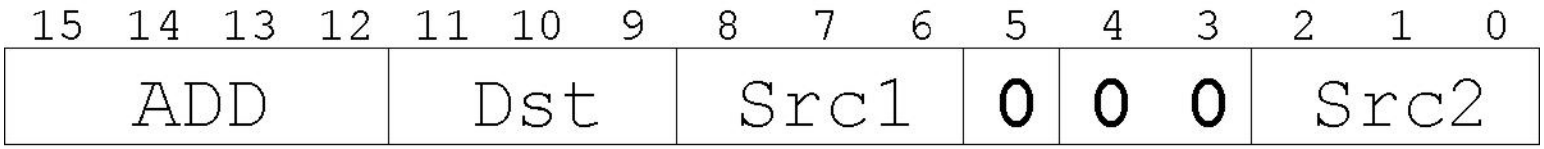
Example: LC-2 ADD Instruction

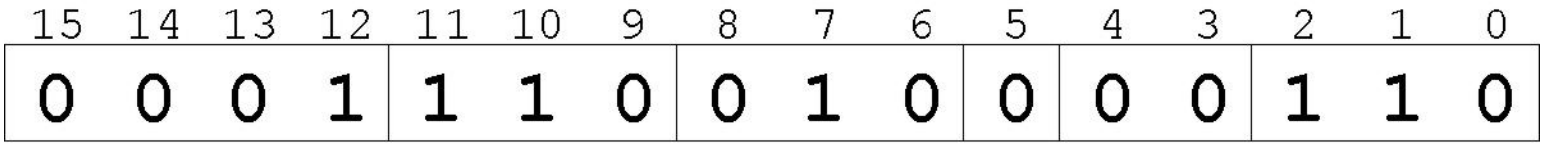
LC-2 has 16-bit instructions.

- **Each instruction has a four-bit opcode, bits [15:12].**

LC-2 has eight *registers* (R0-R7) for temporary storage.

- **Sources and destination of ADD are registers.**





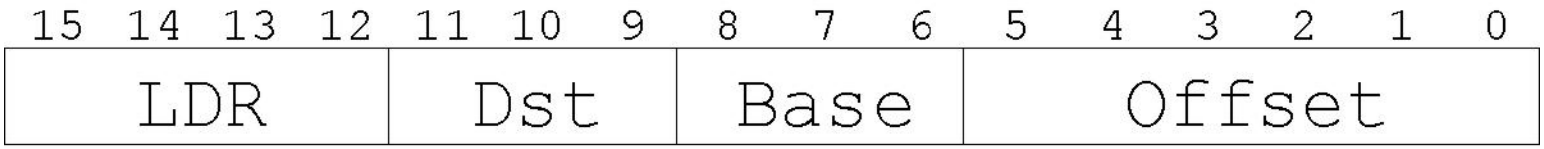
"Add the contents of R2 to the contents of R6, and store the result in R6."

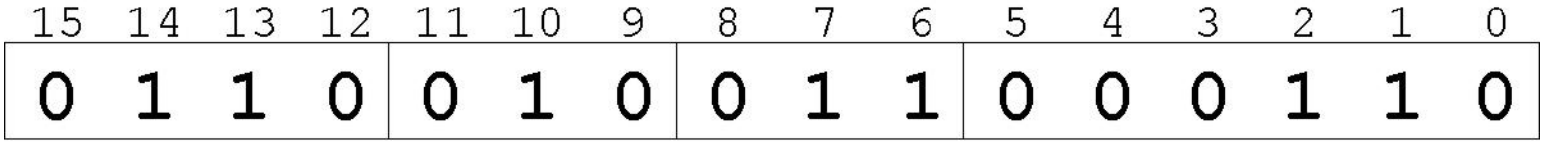
Example: LC-2 LDR Instruction

Load instruction -- reads data from memory

Base + offset mode:

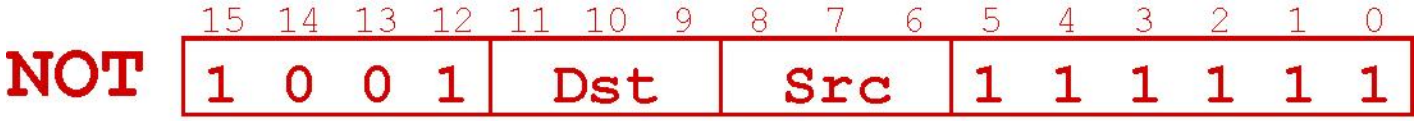
- **add offset to base register -- result is memory address**
- **load from memory address into destination register**

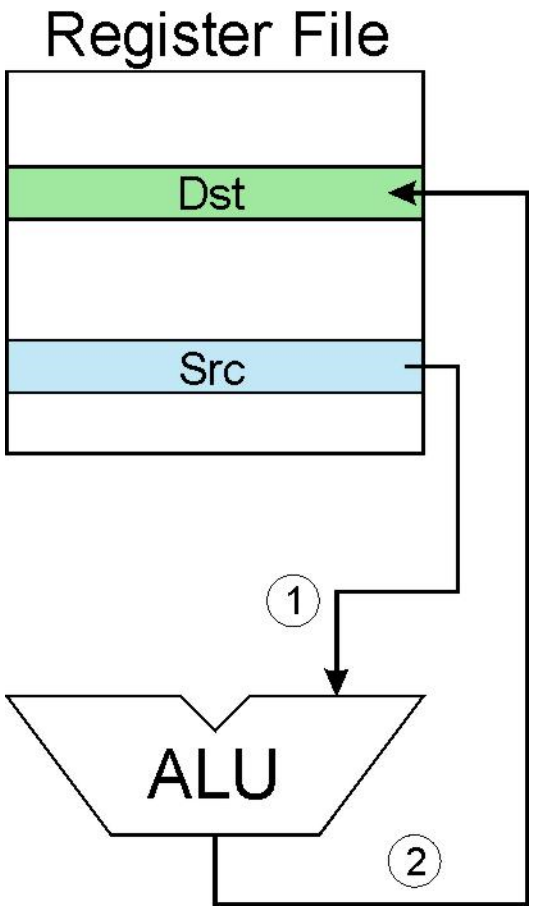




"Add the value 6 to the contents of R3 to form a memory address. Load the contents stored in that address to R2."

NOT (Register)

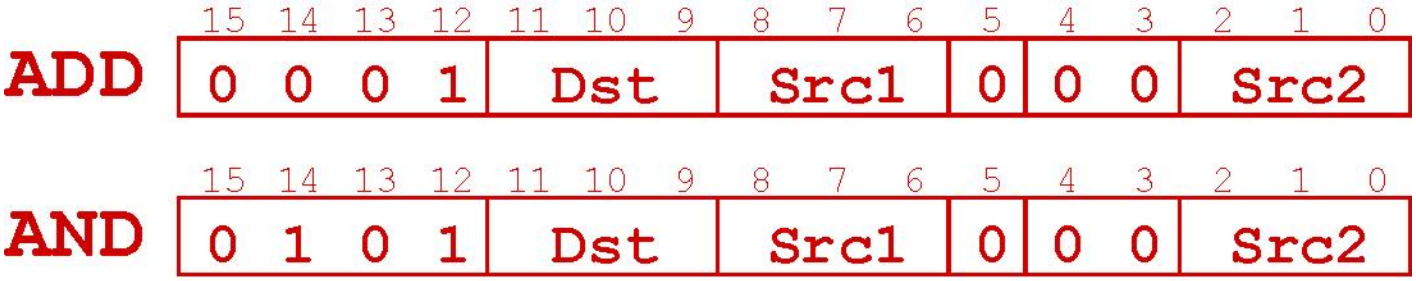


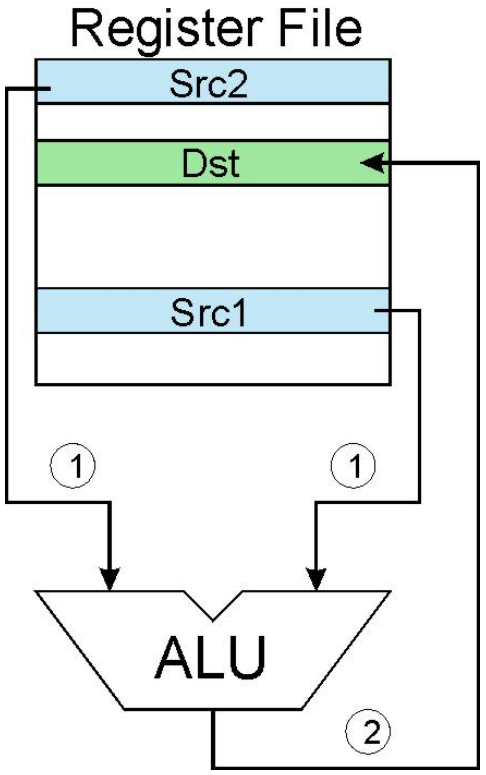


Note: Src and Dst could be the same register.

ADD/AND (Register)

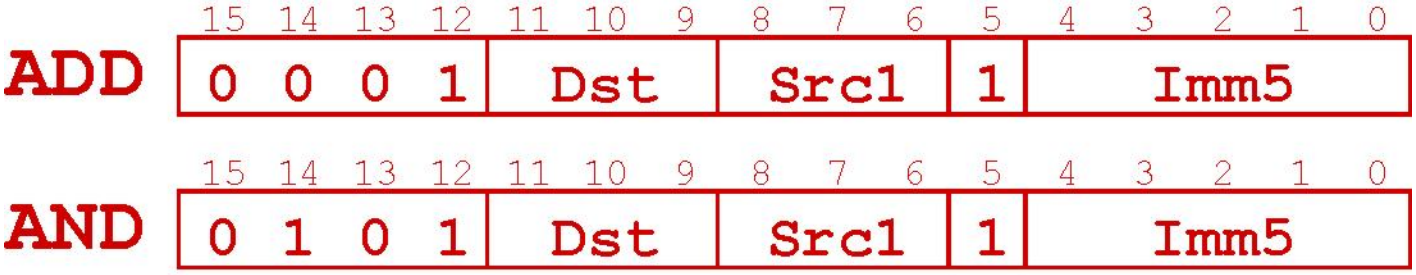
this zero means "register mode"



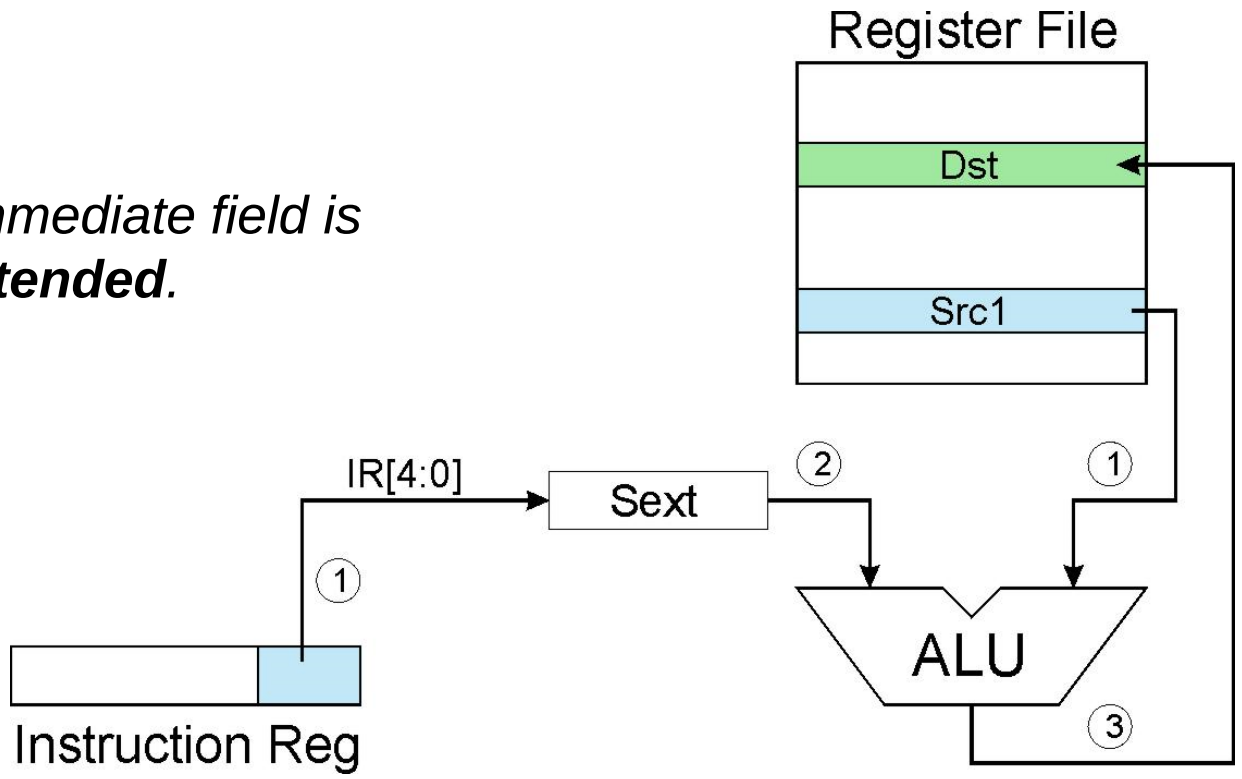


ADD/AND (Immediate)

this one means "immediate mode"



Note: Immediate field is ***sign-extended****.*



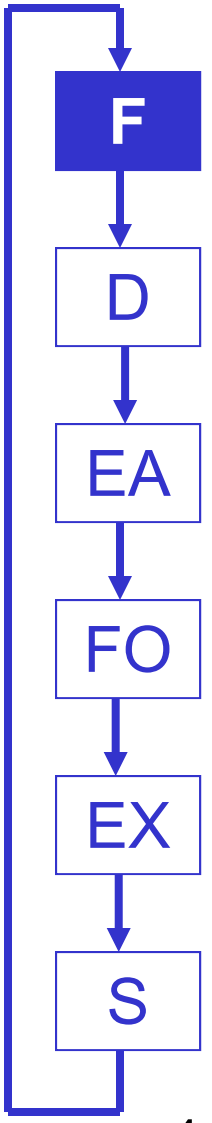
Instruction Processing: FETCH

Load next instruction (at address stored in PC) from memory into Instruction Register (IR).

- **Load contents of PC into MAR.**
- **Send "read" signal to memory.**
- **Read contents of MDR, store in IR.**

Then increment PC, so that it points to the next instruction in sequence.

- **PC becomes PC+1.**



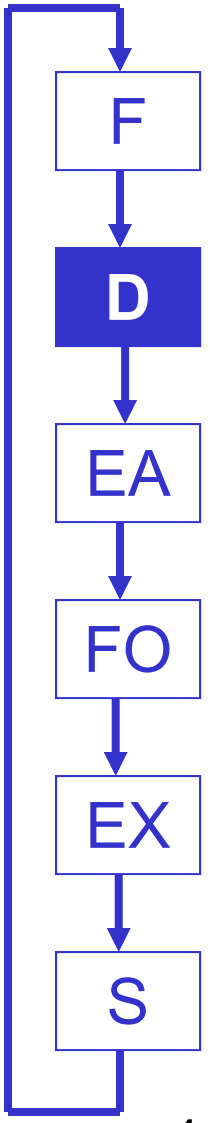
Instruction Processing: DECODE

First identify the opcode.

- **In LC-2, this is always the first four bits of instruction.**
- **A 4-to-16 decoder asserts a control line corresponding to the desired opcode.**

Depending on opcode, identify other operands from the remaining bits.

- **Example:**
  - **for LDR, last six bits is offset**
  - **for ADD, last three bits is source operand #2**



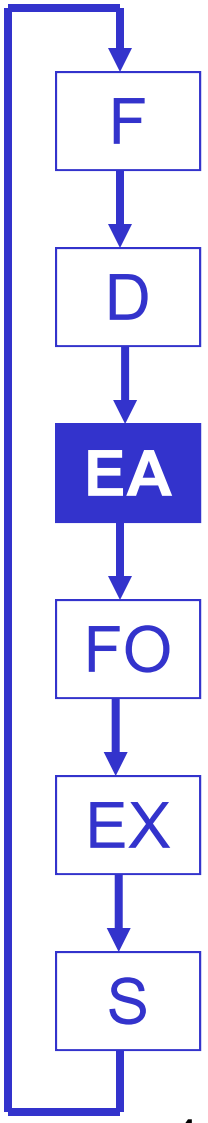
Instruction Processing: EVALUATE ADDRESS

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**

Computed address is called the
Effective Address (EA)

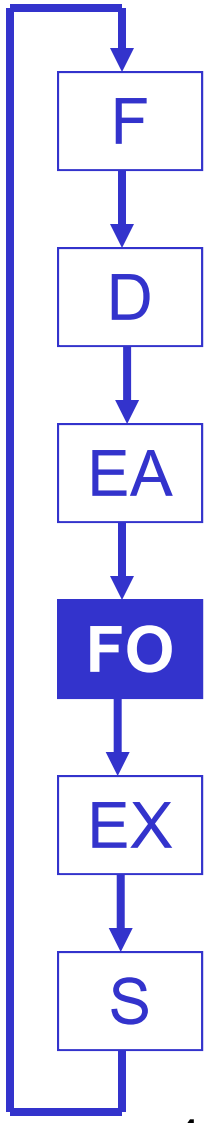


Instruction Processing: FETCH OPERANDS

Obtain source operands needed to perform operation.

Examples:

- **load data from memory (LDR)**
- **read data from registers (ADD)**

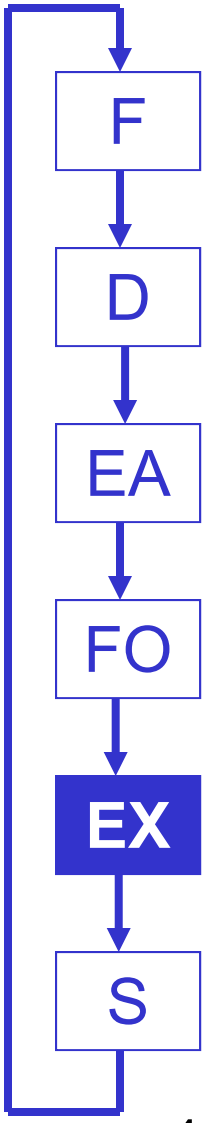


Instruction Processing: EXECUTE

**Perform the operation,
using the source operands.**

Examples:

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



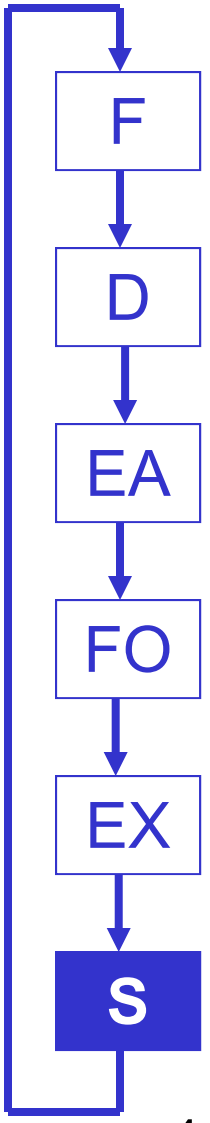
Instruction Processing: STORE

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**

After the STORE phase – start over with Fetch!



Changing the Sequence of Instructions

**In the FETCH phase,
we incremented the Program Counter by 1.**

What if we don't want to always execute the instruction that follows this one?

- **examples: loop, if-then, function call**

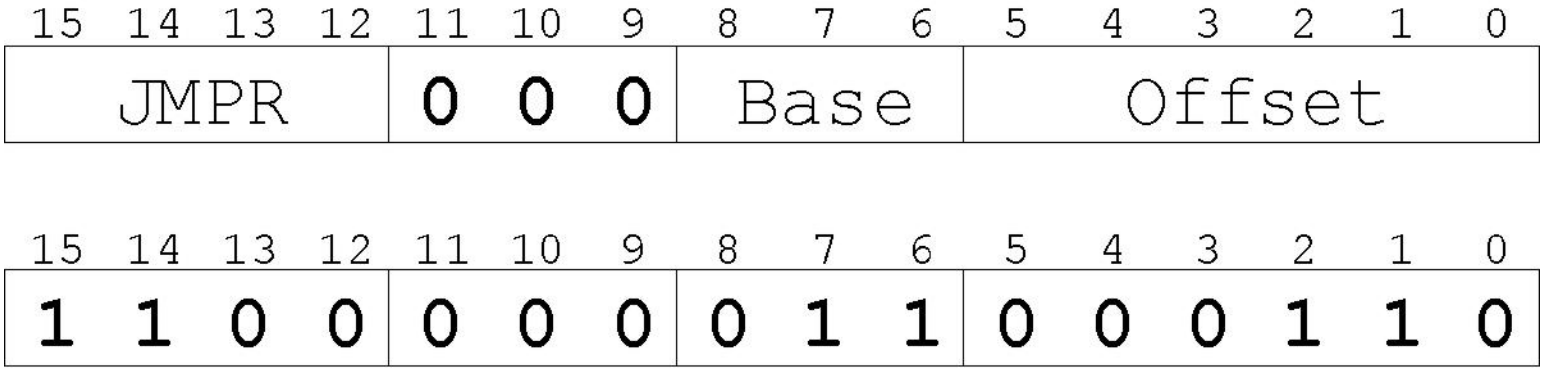
Need special instructions that change the contents of the PC.

These are 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-2 JMPR Instruction

Set the PC to the value obtained by adding an offset to a register. This becomes the address of the next instruction to fetch.



"Add the value of 6 to the contents of R3, and load the result into the PC."

Instruction Processing Summary

Instructions look just like data -- it's all interpretation.

Three basic kinds of instructions:

- **computational instructions (ADD, AND, …)**
- **data movement instructions (LD, ST, …)**
- **control instructions (JMP, BRnz, …)**

Six basic phases of instruction processing:

$$F \rightarrow D \rightarrow EA \rightarrow FO \rightarrow EX \rightarrow S$$

- **not all phases are needed by every instruction**
- **phases may take variable number of machine cycles**

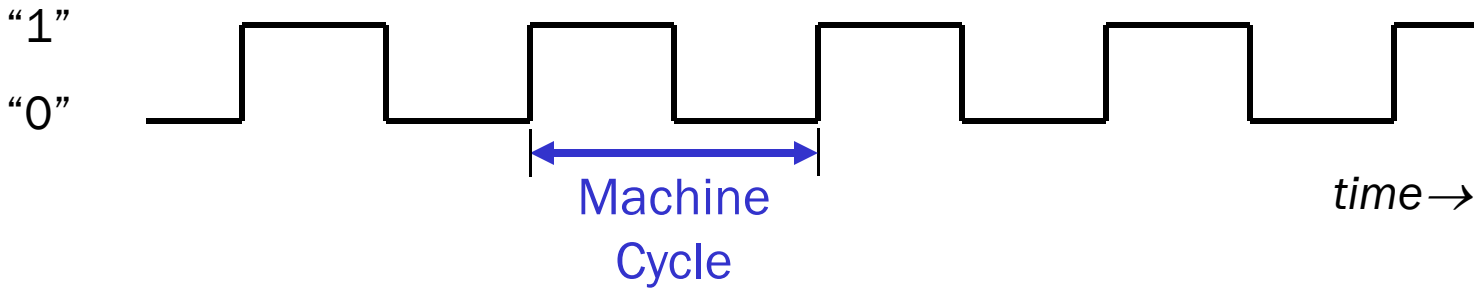
Driving Force: The Clock

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**



Instructions vs. Clock Cycles

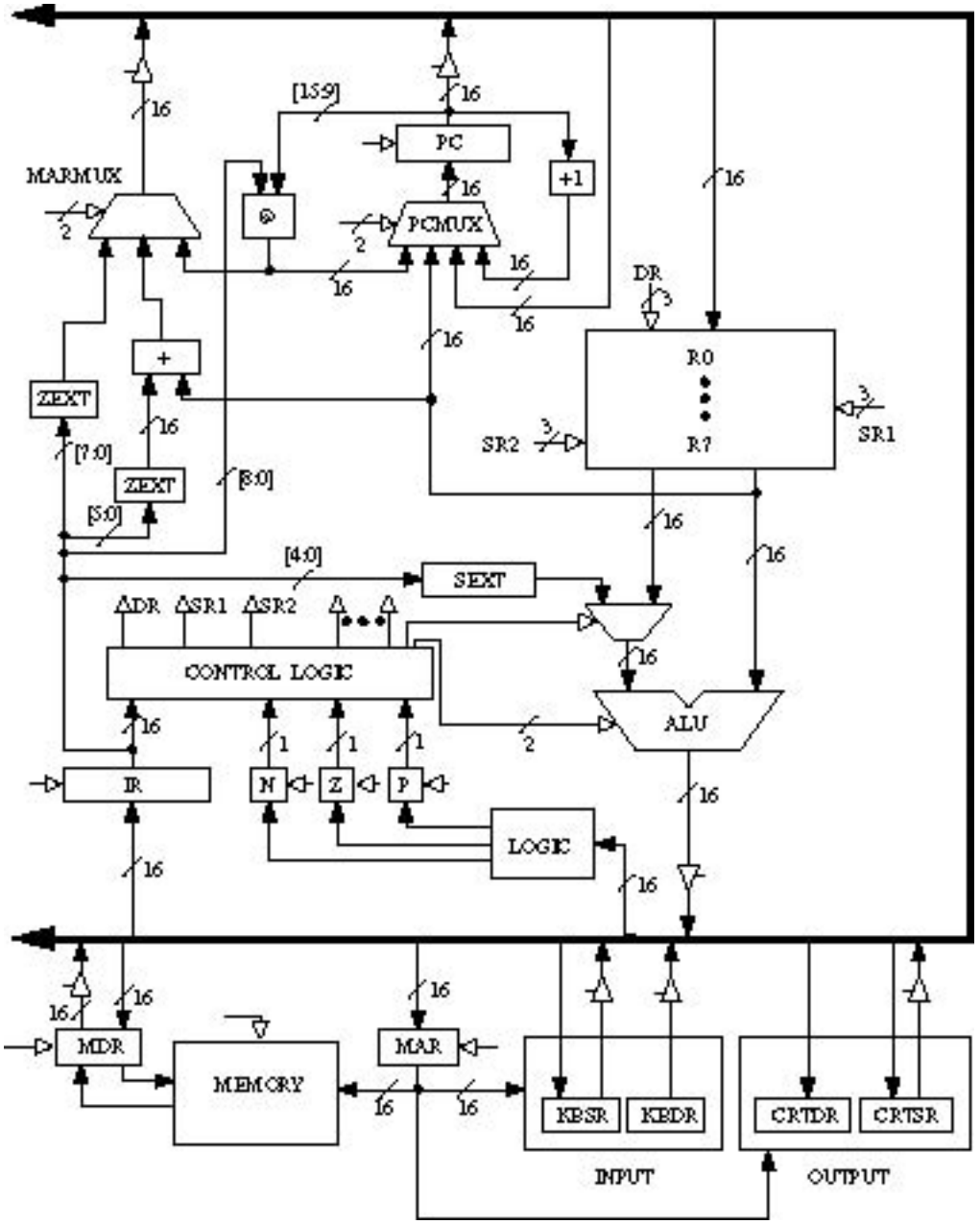
MIPS vs. MHz

- MIPS = millions of instructions per second
- MHz = millions of clock cycles per second

These are not the same -- why?

LC-2 Data Path Revisited

Filled arrow
= info to be processed.
Unfilled arrow
= control signal.



Human-Readable Machine Language

Computers like ones and zeros…

```
0001110010000110
```

Humans like symbols…

```
ADD  R6,R2,R6   ; increment index reg.
```

Assembler is a program that turns symbols into machine instructions.

- **ISA-specific:**
  close correspondence between symbols and instruction set
  - **mnemonics for opcodes**
  - **labels for memory locations**
- **additional operations for allocating storage and initializing data**

An Assembly Language Program

```
;
; Program to multiply a number by the constant 6
;
        .ORIG  x3050
        LD     R1, SIX
        LD     R2, NUMBER
        AND    R3, R3, #0    ; Clear R3.  It will
                             ; contain the product.
; The inner loop
;
AGAIN   ADD    R3, R3, R2
        ADD    R1, R1, #-1   ; R1 keeps track of
        BRp    AGAIN         ; the iteration.
;
        HALT
;
NUMBER  .BLKW  1
SIX     .FILL  x0006
;
        .END
```

LC-2 Assembly Language Syntax

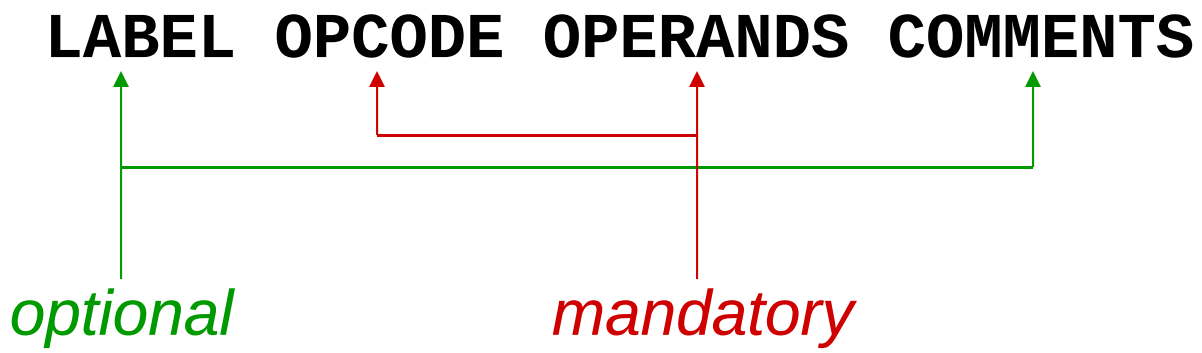
Each line of a program is one of the following:

- **an instruction**
- **an assember directive (or pseudo-op)**
- **a comment**

Whitespace (between symbols) and case are ignored.

Comments (beginning with ";") are also ignored.

An instruction has the following format:



Opcodes and Operands

Opcodes

- reserved symbols that correspond to LC-2 instructions
- listed in Appendix A
  - ex: ADD, AND, LD, LDR, ...

Operands

- registers -- specified by Rn, where n is the register number (R2, R6, …)
- numbers -- indicated by # (decimal) or x (hex) or b(binary)
  - #8, x8, b1000; other examples: #1, x30FA, b1100
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format
  - ex:

```
ADD  R1,R1,R3
ADD  R1,R1,#3
LD   R6,NUMBER
BRz  LOOP
```

Labels and Comments

Label

- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
  - ex:

```
LOOP  ADD  R1,R1,#-1
      BRp  LOOP
```

Comment

- anything after a semicolon is a comment
- ignored by assembler
- used by humans to document/understand programs
- tips for useful comments:
  - avoid restating the obvious, as “decrement R1”
  - provide additional insight, as in “accumulate product in R6”
  - use comments to separate pieces of program

Assembler Directives

Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but "opcode" starts with dot

Opcode	*Operand*	*Meaning*
`.ORIG`	address	starting address of program
`.END`		end of source program (does not put a HALT in code)
`.BLKW`	n	allocate n words of storage
`.FILL`	n	allocate one word, initialize with value n
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Intel 64 and IA-32

The definitive Intel 64 and IA-32 Architectures Software Developer's Manuals are available online. These include:

Volume 1: Basic Architecture

Volume 2a: Instruction Set Reference, A-M

Volume 2b: Instruction Set Reference, N-Z

Volume 3a: System Programming Guide, Part 1

Volume 3b: System Programming Guide, Part 2

https://software.intel.com/en-us/articles/intel-sdm