Computer Architecture

Introduction
(Chapter 4,5)

Based on slides © McGraw-Hill
Additional material © 2013 Farmer
Additional material © 2014 Narahari

Putting it all together: Microarchitecture and ISA

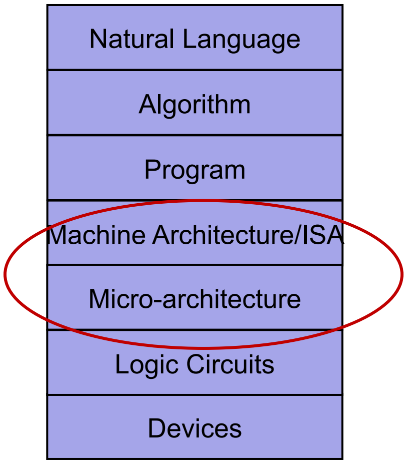
- We now have a collection of combinational and sequential logic devices & methodology for designing these circuits
  - Digital logic circuits used in communications/networking equipment, computers, consumer electronics, …..
  - Our focus is building a computer using these devices

- We next need to combine these devices to accomplish our goal of building a central processing unit of a computer

- To do this we need a master plan: i.e., a model of a computer – von Neuman architetures
  - Chapters 4-5

Levels of abstraction – Hardware stack

To understand these two
levels we will need to jump
Between the two levels:
we need an idea of ISA to
discuss/design microarchitecture



Important Note: Building circuits using ‘standard’ devices

- now that we have a set of combinational devices, we can build/design circuits using these devices from a “library”
  - Adders, Decoders, Multiplexers, Flip Flops, Registers, Memory…..
  - You do not have to keep going to the transistor or gate level when designing a ‘system’
- Analogous to using library functions (or functions you have implemented earlier) to write your program
  - Work on formulating a solution/design by using ‘high level’ abstractions/devices
    - Example: Need to store a value, think ‘register’ or ‘memory’ instead of ‘RS latch’
-

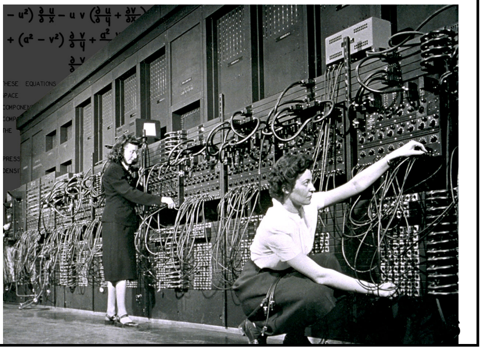
History of the Stored Program Computer

- 1943: ENIAC
  - Presper Eckert and John Mauchly -- first general electronic computer. (or was it John V. Atanasoff in 1939?)
  - Hard-wired program -- settings of dials and switches.
- 1944: Beginnings of EDVAC
  - among other improvements, includes program stored in memory and binary
- 1945: John von Neumann
  - wrote a report on the stored program concept, known as the *First Draft of a Report on EDVAC*
- The basic structure proposed in the draft became known as the “von Neumann machine” (or model).
  - a *memory*, containing instructions and data
  - a *processing unit*, for performing arithmetic and logical operations
  - a *control unit*, for interpreting instructions

For more history, see http://www.maxmon.com/history.htm

Historical Perspective

- ENIAC built during World War II was the first general purpose computer: *Eckert and Mauchly, U.Penn, ~1943*
  - Used for computing artillery firing tables
  - About 1500 square ft.; over 30 tons
  - Used 18,000 vacuum tubes, decimal representation
  - Performed **1900 additions per second**

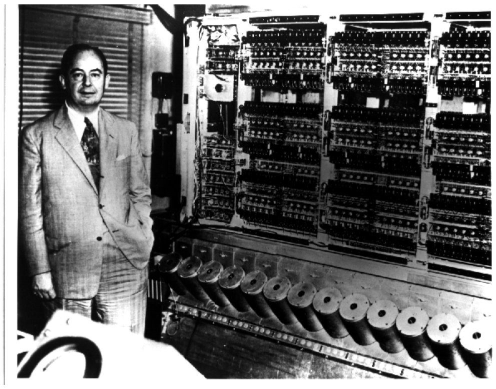


EDVAC, 1944-49: Electronic Discrete Variable Computer

- ‘successor’ to ENIAC – same designers Eckert & Mauchly, Penn
  - von Neumann collaborated with this group to write his report
- Program stored in memory
- Binary digital representation

- Draft report by von Neumann… came to be known as ‘von Neumann model’ !!!

(In reality: it was a collaboration!)

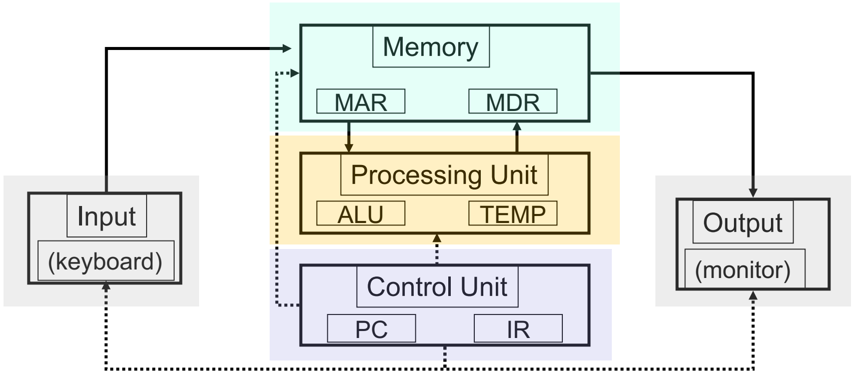


John von Neumann and EDVAC

Von Neumann Model

- The basic structure in the von Neumann architecture model
  - A memory containing instructions and data
  - A processing unit for performing arithmetic & logical operations
  - A control unit for interpreting instructions
- The central idea is:
  - the *program* and *data* are both *stored* as sequences of bits in the *computer's memory*, and
  - the program is executed, one instruction at a time, under the direction of the control unit.

The von Neumann Model



Memory: holds both data and programs

Processing unit: carries out the instructions

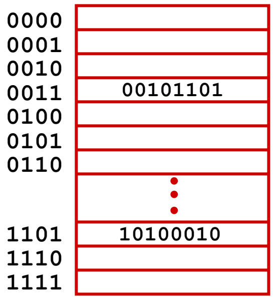
Control unit: sequences and interprets instructions

Input: external information/data into the memory

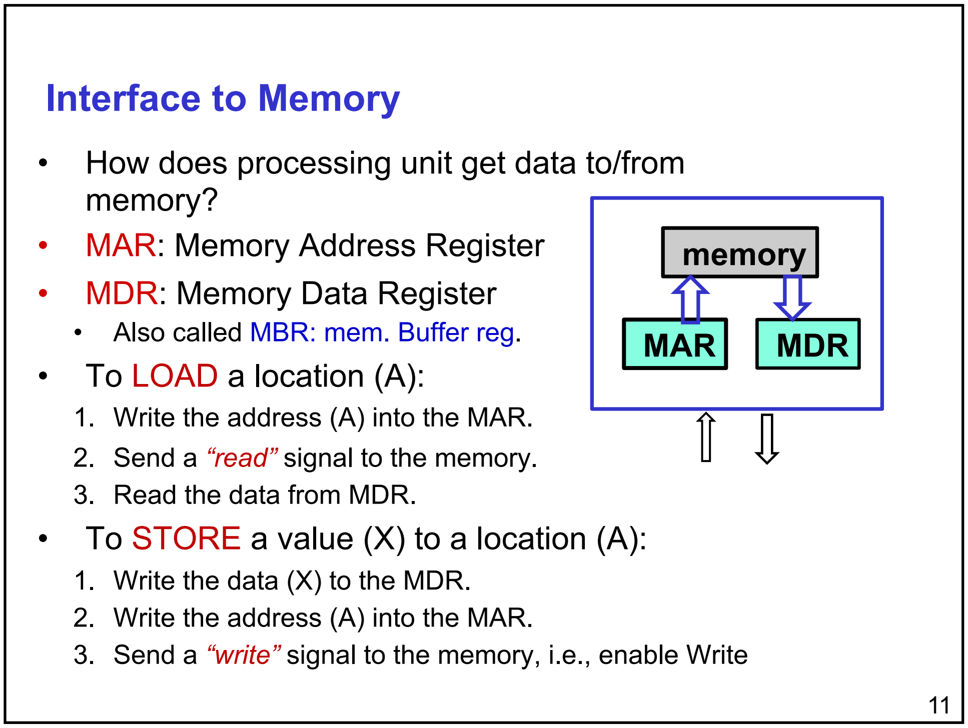
Output: produces results for the user

Von Neuman Model: Memory

- 2^N x *m* array of stored bits
- Address and Addressability (contents)
  - unique (*N*-bit) identifier of location
  - *m*-bit value stored in location
- Interacting with memory
  - (operations):
  - LOAD (READ)
    - ○ read a value from a memory location
  - STORE (WRITE)
    - ○ write a value to a memory location

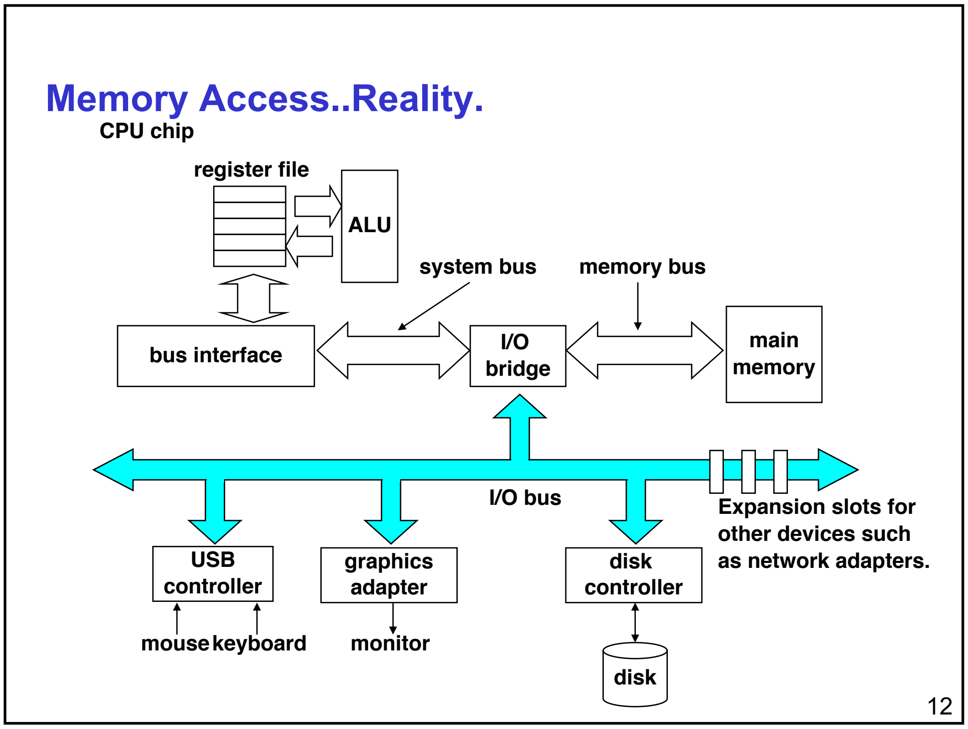


0000	
0001	
0010	
0011	00101101
0100	
0101	
0110	
	⋮
1101	10100010
1110	
1111	



Interface to Memory

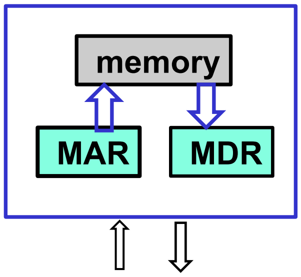
- How does processing unit get data to/from memory?
- MAR: Memory Address Register
- MDR: Memory Data Register
  - Also called MBR: mem. Buffer reg.
- To LOAD a location (A):
  1. Write the address (A) into the MAR.
  2. Send a *"read"* signal to the memory.
  3. Read the data from MDR.
- To STORE a 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, i.e., enable Write



Memory Access..Reality.

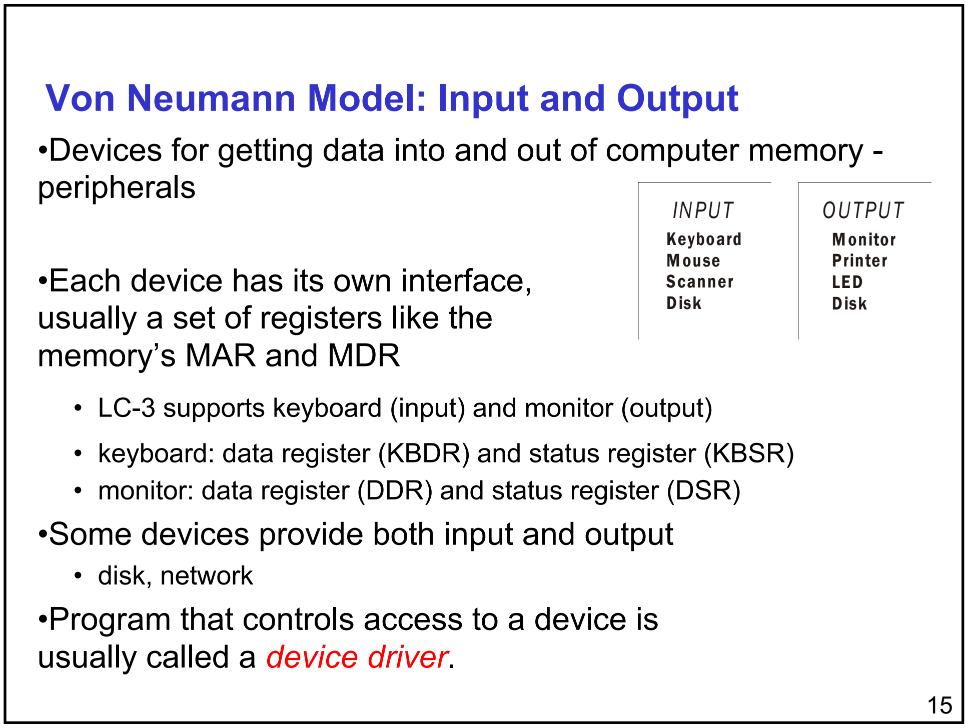
Will return to memory hierarchy later in the course

- For now, keep it simple: one memory device
  - *N* bit address space*, m* bits in each location



Von Neumann Model: Processing Unit

- Processing Unit- does the actual work!
  - (At a minimum) has Arithmetic & Logic Unit (**ALU**) and General Purpose Registers (**GPR**s).
  - The number of bits a basic Processing Unit operation can handle is called the **WORD SIZE** of the machine.
  - Today: can consist of many units, each specializing in some complex functions
- ALU
  - Performs basic operations: add, subtract, and, not, etc.
  - Generally operates on whole words of data.
    - Some can also operate on subsets of words (eg. single bits or bytes)
  - LC3 does ADD, AND, NOT
  - You have seen a design of a simple ALU (to Add/Subtract)!
- Registers:
  - Fast "on-board" storage for a small number of words.
  - Invaluable for intermediate data storage while processing
  - Close to the ALU (much faster access than RAM)
  - LC3 has 8 general purpose registers R_0,R_1,…,R_7.

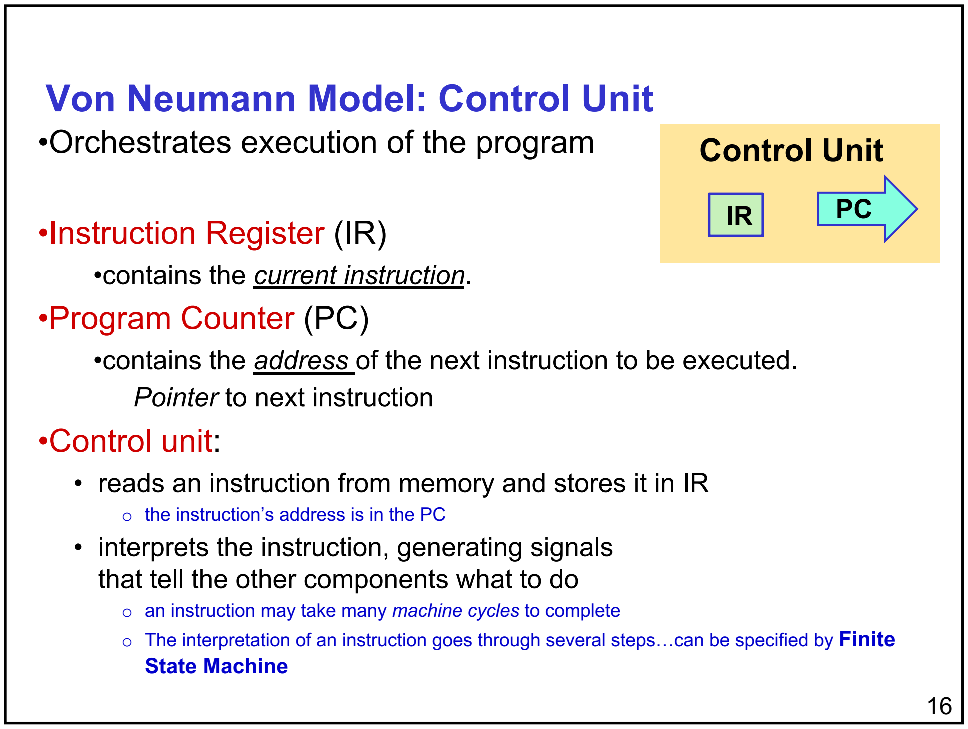


Von Neumann Model: Input and Output

- Devices for getting data into and out of computer memory - peripherals

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-3 supports keyboard (input) and monitor (output)
  - keyboard: data register (KBDR) and status register (KBSR)
  - monitor: data register (DDR) and status register (DSR)
- Some devices provide both input and output
  - disk, network
- Program that controls access to a device is usually called a *device driver*.



Von Neumann Model: Control Unit

- Orchestrates execution of the program

Control Unit
IR
PC

- Instruction Register (IR)
  - contains the *current instruction*.
- Program Counter (PC)
  - contains the *address* of the next instruction to be executed.

    Pointer to next instruction
- Control unit:
  - reads an instruction from memory and stores it in IR
    - 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
    - The interpretation of an instruction goes through several steps…can be specified by **Finite State Machine**

What is an 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 (16,32,..),
- Control unit interprets instruction:
  - ○ generates sequence of control signals to carry out operation.
- Operation is either executed completely, or not at all.

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

ISA

- The ISA specifies all the information about the computer that the software needs to be aware of.
- Who uses an ISA?
- What is specified?
- How big an ISA
  - Reduced Instruction set (RISC)
  - Complex Instruction set (CISC)
- ISA serves as the interface b/w hardware and software
  - Software needs to know instructions in the hardware
  - Hardware needs to know instructions to be implemented in the hardware by the Mircoarchitecture

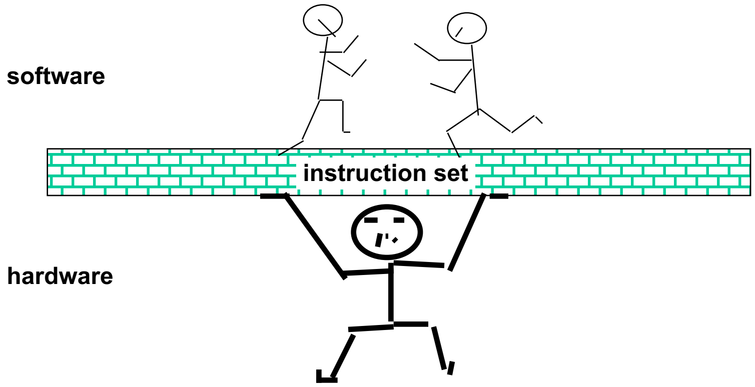
Instruction Set Architecture

•ISA = All of the *programmer-visible* components and operations of the computer

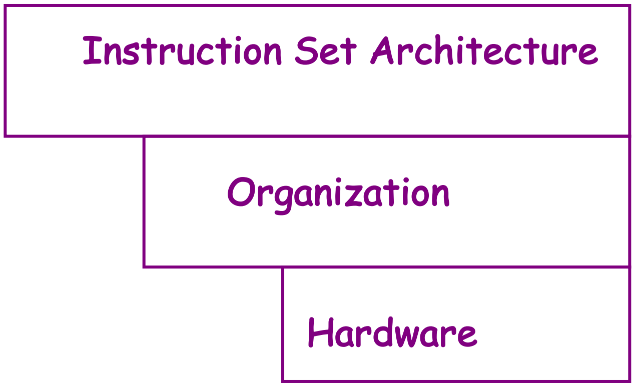
- memory organization
  - address space -- how many locations can be addressed?
  - addressibility -- how many bits per location?
- register set
  - how many? what size? how are they used?
- instruction set
  - opcodes
  - data types
  - addressing modes

•ISA provides all information needed for someone that wants to write a program in machine language
or translate from a high-level language to machine language

What is the Hardware/Software Interface ?



Computer Architecture is ...



Instruction Set Architecture

Organization

Hardware

21

ISA: Types of Instruction

- *1. Operate* Instructions
  - process data (addition, logical operations, etc.)
- *2. Data Movement* Instructions …
  - move data between memory locations and registers.
- *3. Control* Instructions …
  - change the sequence of execution of instructions in the stored program.
    - The default is sequential execution: the PC is incremented by 1 at the start of every Fetch, in preparation for the next one.
    - Control instructions set the PC to a new value during the Execute phase, so the next instruction comes from a different place in the program.
    - This allows us to build control structures such as loops and branches.

22

22

Encoding the operations/opcode

- N-bit word used by processor (addressability)
- Use some of these bits to encode the different instructions

- Example: We have 32-bit processor
  - We have 50 instructions we need to encode
  - We need 6 bits to encode 50 different binary strings
  - Opcode is specified using these 6 bits
- In reality: could get more ‘creative’ than just sticking to these 6 bits…..

Example: LC-3 ADD Instruction

- LC-3 has 16-bit instructions.
  - Each instruction has a four-bit opcode, bits [15:12].
- LC-3 has eight *registers* (R0-R7) for temporary storage.
  - Sources and destination of ADD are registers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD				Dst			Src1			0	0	0	Src2		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0

Semantics: “Add the contents of R2 to the contents of R6, and store the result in R6.”

Example: LC-3 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR				Dst			Base			Offset					

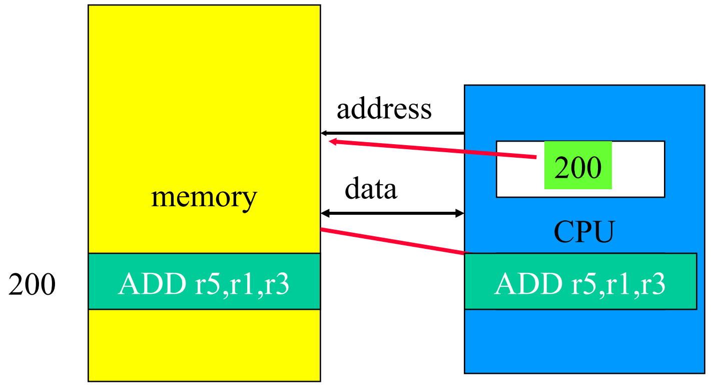
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	0	0	0	1	1	0

“Semantics: Add the value 6 to the contents of R3 to form a memory address. Load the contents of that memory location to R2.”

How do instructions get executed ?
Instruction Cycle - overview

- The Control Unit orchestrates the complete execution of each instruction:
  - At its heart is a Finite State Machine that sets up the state of the logic circuits according to each instruction.
  - This process is governed by the system clock - the FSM goes through one transition (“machine cycle”) for each tick of the clock.
  - 1 Ghz (10^9) clock frequency = 1 nanosecond clock cycle

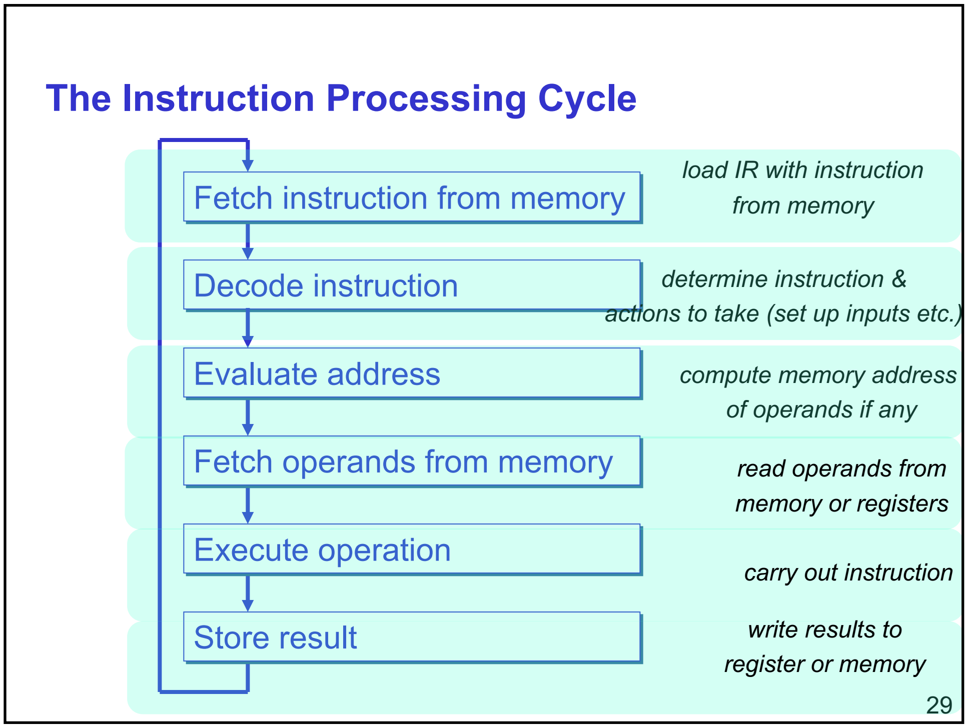
CPU + memory



Instruction Cycle - overview

Six phases of the complete Instruction Cycle

- Fetch: load IR with instruction from memory
- Decode: determine action to take (set up inputs for ALU, RAM, etc.)
- Evaluate address: compute memory address of operands, if any
- Fetch operands: read operands from memory or registers
- Execute: carry out instruction
- Store results: write result to destination (register or memory)



The Instruction Processing Cycle

Fetch instruction from memory — *load IR with instruction from memory*

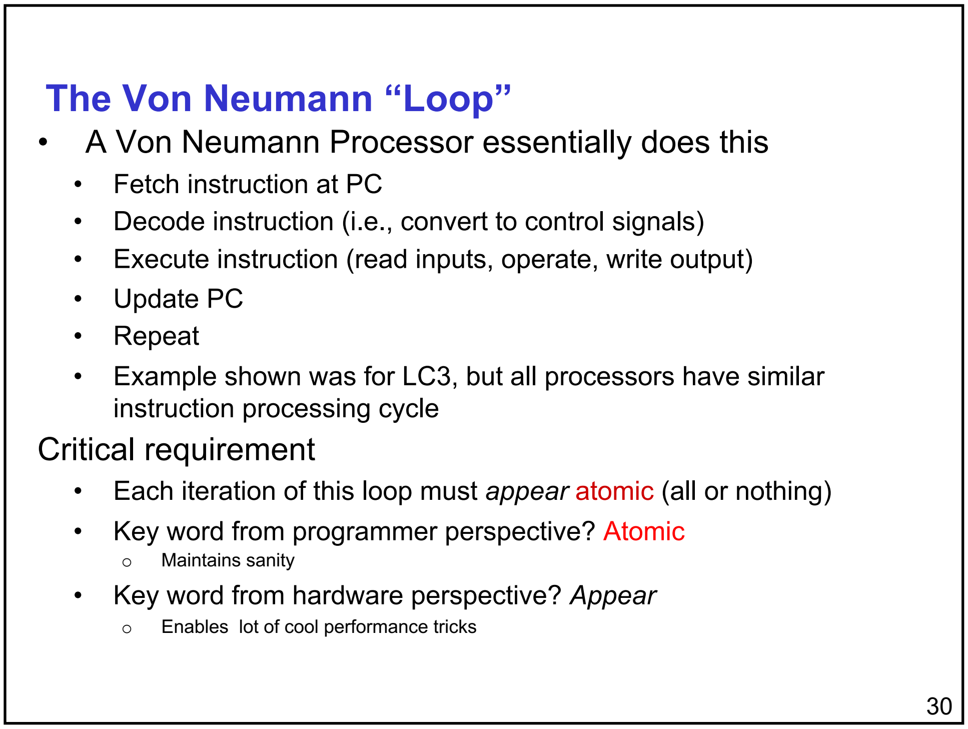
Decode instruction — *determine instruction & actions to take (set up inputs etc.)*

Evaluate address — *compute memory address of operands if any*

Fetch operands from memory — *read operands from memory or registers*

Execute operation — *carry out instruction*

Store result — *write results to register or memory*



The Von Neumann “Loop”

- A Von Neumann Processor essentially does this
  - Fetch instruction at PC
  - Decode instruction (i.e., convert to control signals)
  - Execute instruction (read inputs, operate, write output)
  - Update PC
  - Repeat
  - Example shown was for LC3, but all processors have similar instruction processing cycle

Critical requirement

- Each iteration of this loop must *appear* atomic (all or nothing)
- Key word from programmer perspective? Atomic
  - Maintains sanity
- Key word from hardware perspective? *Appear*
  - Enables lot of cool performance tricks

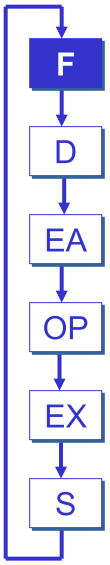
What actions take place in each step….

- Next, take a closer look at the “control” signals needed and the actions that take place at each step of the instruction cycle
  - We can then go into the actions/steps to implement each instruction

- Important: we need this information (control signals) when we design/implement a processor!
  - Next topic we will go into detail on how the processor datapath and control is implemented, and some sequential logic devices.

Instruction Processing Step 1: FETCH

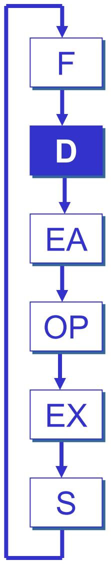
- Load next instruction (at address stored in PC) from memory into Instruction Register (IR).
- 1.Copy contents of PC into MAR: MAR ← (PC)
- 2.Send “read” signal to mem and read: MDR ← (MAR)
- 3.Copy contents of MDR into IR: IR ← MDR
- 4. increment PC, so that it points to next inst in sequence: PC = PC+1
- FETCH takes at least 3 steps/cycles
  - 1,3,4 take one cycle, but 2 can take more
  - 1,4 can be done in same cycle



Instruction Processing Step 2: DECODE

- First identify the opcode.
  - In LC-3, this is always the first four bits of instruction.
    - A 4-to-16 decoder asserts a control line corresponding to the desired opcode.

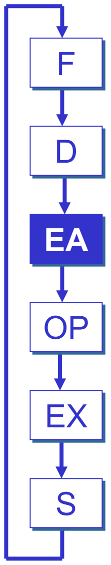
- Depending on opcode, identify other operands from the remaining bits.
  - Example:
    - for LDR, last six bits is offset
    - for ADD, last three bits is source operand #2

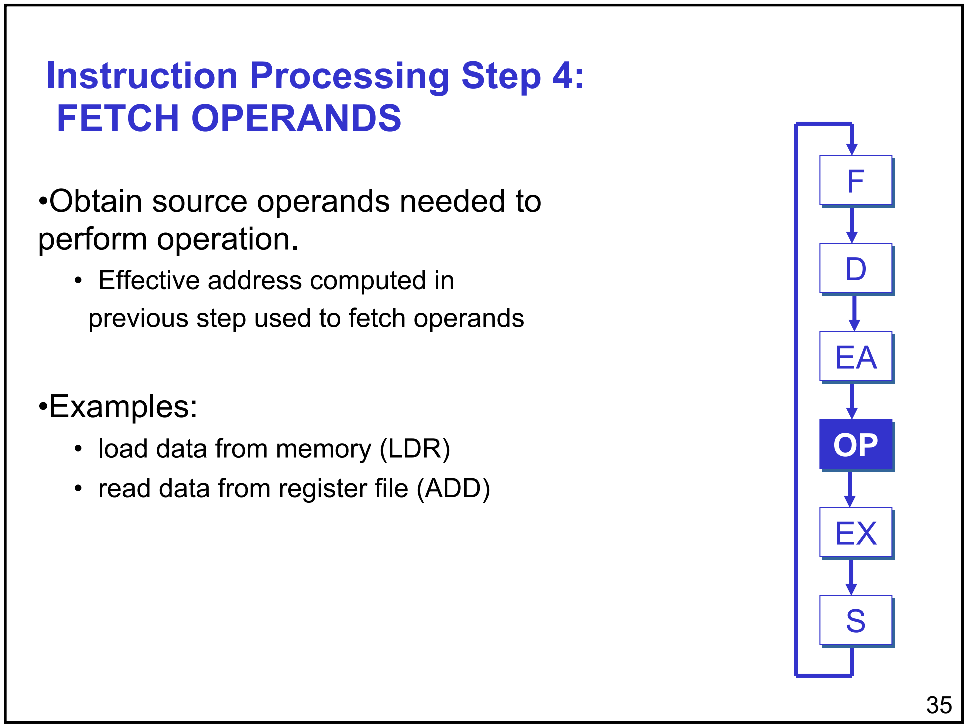


Instruction Processing Step 3: EVALUATE ADDRESS

- For instructions that require memory access, compute address used for access.
  - Called Effective Address (EA)

- Examples:
  - add offset to base register (as in LDR)
  - add offset to PC
  - add offset to zero

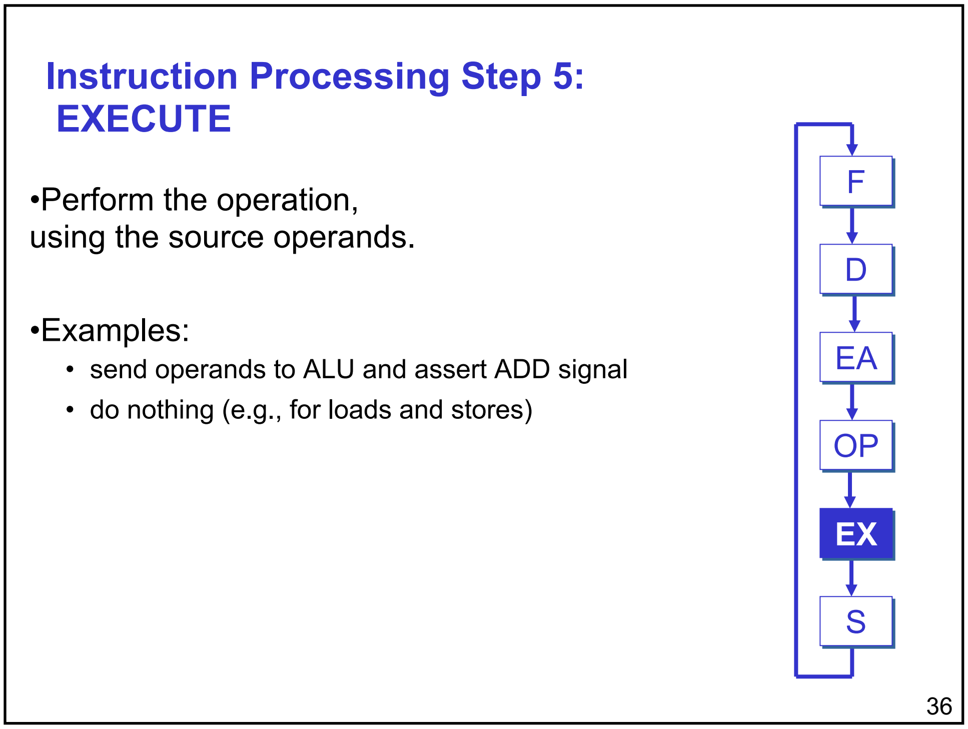




Instruction Processing Step 4: FETCH OPERANDS

- Obtain source operands needed to perform operation.
  - Effective address computed in previous step used to fetch operands

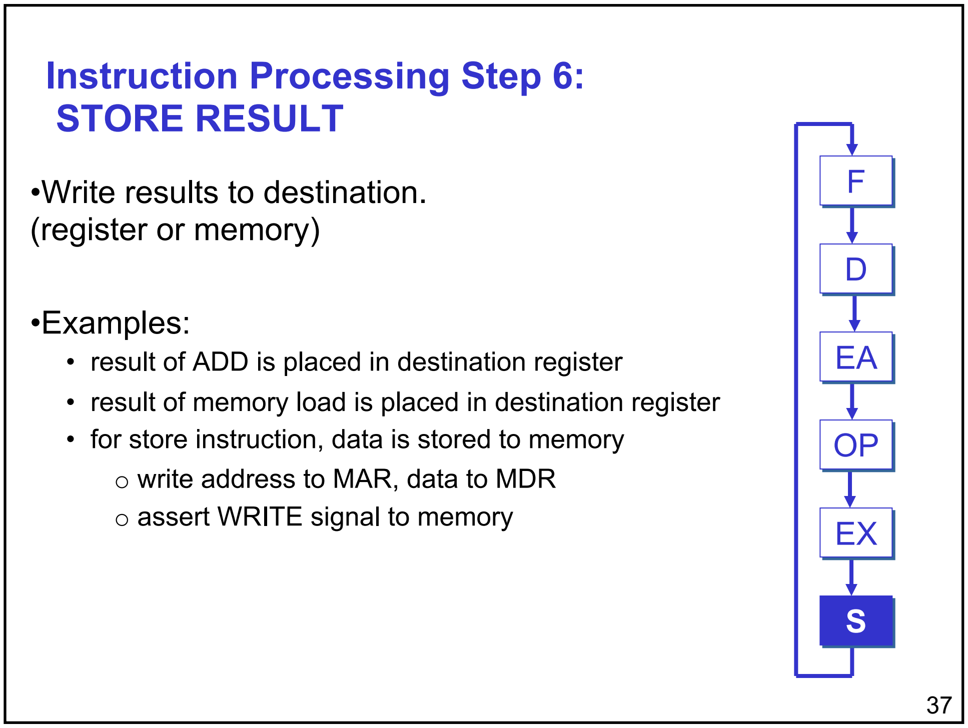
- Examples:
  - load data from memory (LDR)
  - read data from register file (ADD)



Instruction Processing Step 5: 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)

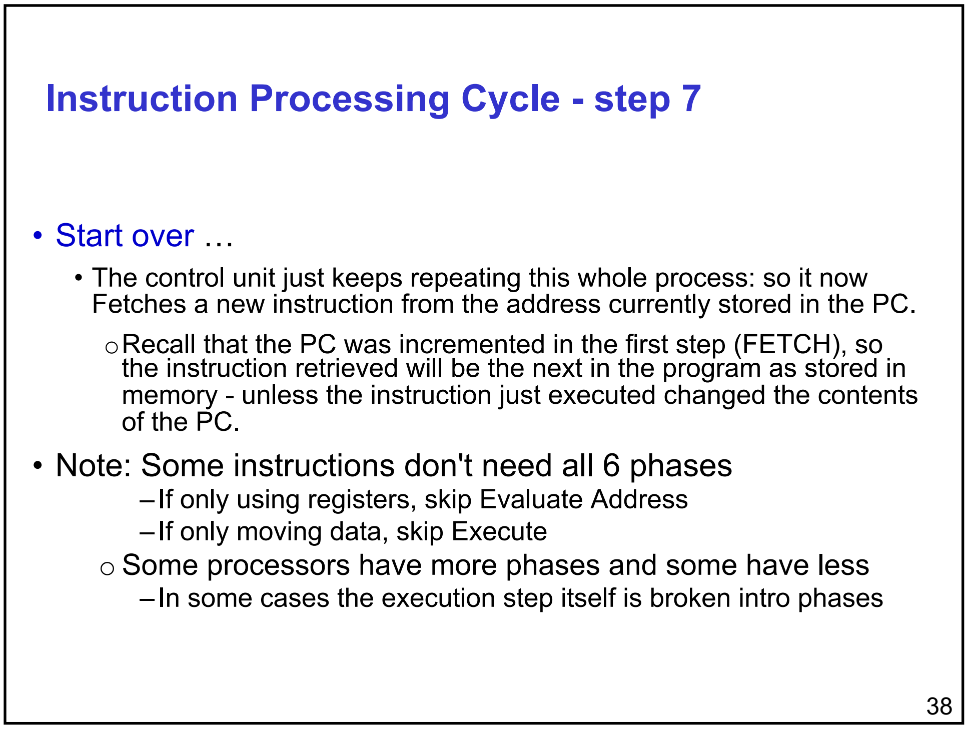


Instruction Processing Step 6: STORE RESULT

•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 Cycle - step 7

- Start over …
  - The control unit just keeps repeating this whole process: so it now Fetches a new instruction from the address currently stored in the PC.
    - ○ Recall that the PC was incremented in the first step (FETCH), so the instruction retrieved will be the next in the program as stored in memory - unless the instruction just executed changed the contents of the PC.
- Note: Some instructions don't need all 6 phases
  - –If only using registers, skip Evaluate Address
  - –If only moving data, skip Execute
  - ○ Some processors have more phases and some have less
    - –In some cases the execution step itself is broken intro phases

Flow Control

- Normally we execute instructions one after another
- When might we not want to do this?

Changing the Sequence of Instructions

•In the FETCH phase, we increment 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 *control instructions*.

- jumps are unconditional -- always change the PC
- branches are conditional -- change the PC only if some condition is true (e.g., the result of an ADD 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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP				0	0	0	Base			0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0

“Load the contents of R3 into the PC.”

Early programming languages had a “GOTO ..” statement

Instruction Processing Summary

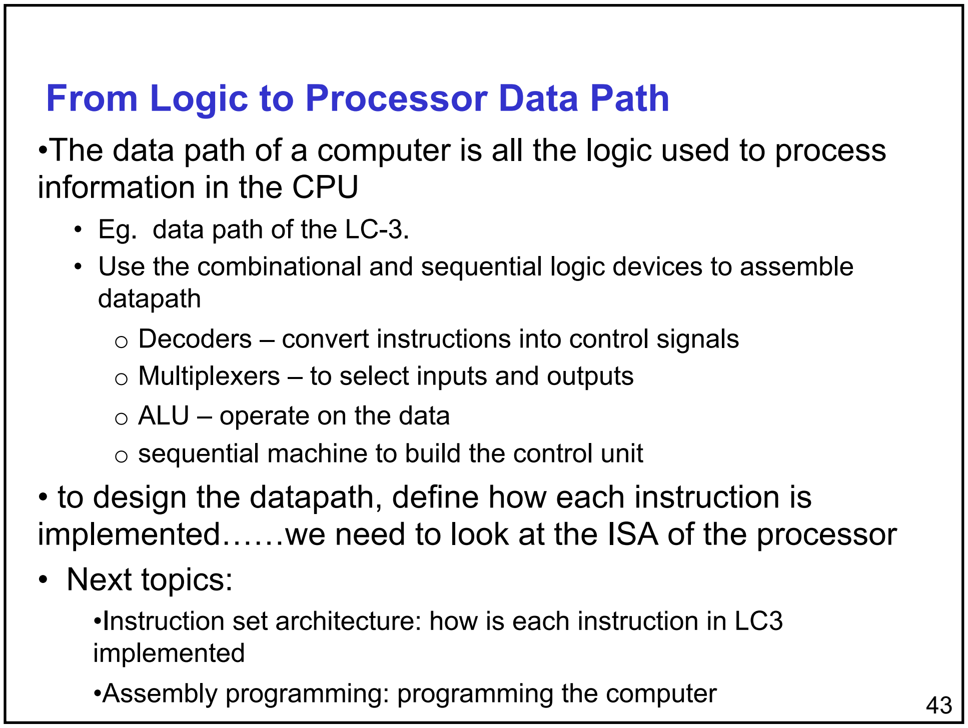
•Instructions look just like data -- it’s all interpretation.

•Three basic kinds of instructions:

- Compute/operate instructions (ADD, AND, …)
- data movement instructions (LD, ST, …)
- control instructions (JMP, BRnz, …)

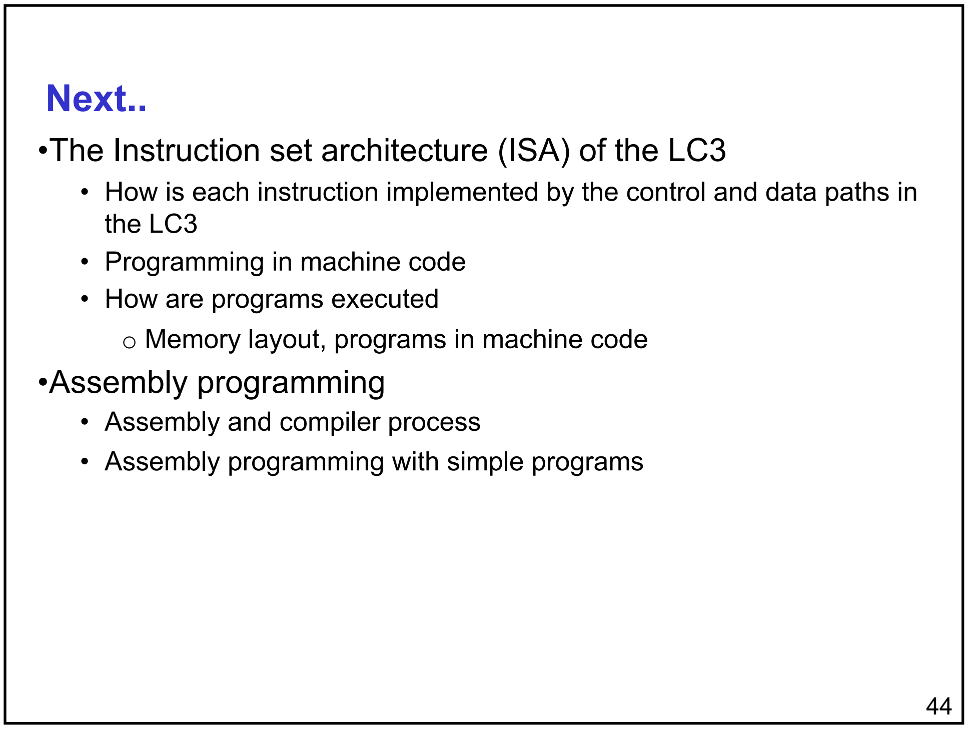
•Six basic phases of instruction processing:

- $F \rightarrow D \rightarrow EA \rightarrow OP \rightarrow EX \rightarrow S$
  - not all phases are needed by every instruction
  - phases may take variable number of machine cycles



From Logic to Processor Data Path

- The data path of a computer is all the logic used to process information in the CPU
  - Eg. data path of the LC-3.
  - Use the combinational and sequential logic devices to assemble datapath
    - Decoders – convert instructions into control signals
    - Multiplexers – to select inputs and outputs
    - ALU – operate on the data
    - sequential machine to build the control unit
- to design the datapath, define how each instruction is implemented……we need to look at the ISA of the processor
- Next topics:
  - Instruction set architecture: how is each instruction in LC3 implemented
  - Assembly programming: programming the computer



Next..

- The Instruction set architecture (ISA) of the LC3
  - How is each instruction implemented by the control and data paths in the LC3
  - Programming in machine code
  - How are programs executed
    - Memory layout, programs in machine code
- Assembly programming
  - Assembly and compiler process
  - Assembly programming with simple programs