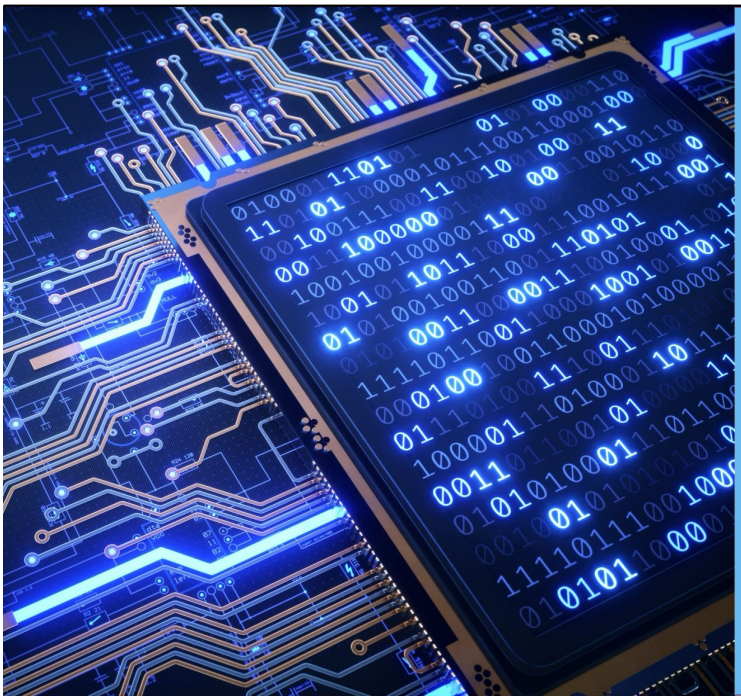


Computer Systems

Central Processing Unit

CS51 – Fall 2025

We are ready to move from memory to the CPU.

The Central Processing Unit (CPU)

- The CPU is the “brain” of the computer. Its job is to run programs, like Word, Chrome, etc.
- Programs are made from **instructions** that tell the computer what to do.
- Some examples of instructions might be to perform mathematical and logical operations (e.g., add) or read/write in memory.
- At this level, we will stop caring about the physical components that the CPU is built out of and focus more broadly on the computer organization or **microarchitecture**, focusing on how instructions are executed, data are handled, and operations are performed by the CPU.
- We know that data, like numbers, characters, etc are stored in memory. Programs are stored in memory, too. We will specifically assume that the instructions are stored in a block of contiguous words in memory.

2

The Central Processing Unit, CPU for short, is the brain of the computer. Its job is to run programs made from instructions that tell the computer what to do. Instructions can be mathematical or logical, or have to do with reading/writing in memory. At this level, we are ready to move one level up in the abstraction and we no longer care about the physical components that the CPU is built out of. Instead, we will focus more broadly on the computer organization or microarchitecture, focusing on how instructions are executed, data are handled, and operations are performed by the CPU. We know that data, like numbers, characters, etc are stored in memory. Programs are stored in memory, too. We will specifically assume that the instructions are stored in a block of contiguous words in memory.

Words and processors

- Processors handle data in chunks of bits called **words.**
  - The size of the word depends on the architecture of the processor. Today, for most computers, the size of a word is 32 or 64-bits/4 or 8 bytes.
- Next, we will see a simplified model of how the CPU reads a program from the memory and reads/writes in memory.

3

Processors handle data in chunks of bits called words. The size of the word depends on the architecture of the processor. Today, for most computers, the size of a word is 32 or 64-bits/4 or 8 bytes. Next, we will see a simplified model of how the CPU reads a program from the memory and reads/writes in memory.

Memory

- Let's assume our (tiny!) RAM consists of 8 words, each having 8 bits.
- To simplify things later, we will assume that addresses are 4-bits.
- Our RAM will hold both the instructions of a small program and the results of the program as it's executed.

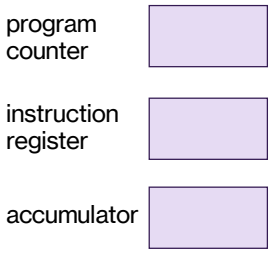
Address	Data
0000	0001 0110
0001	1011 0111
0010	0100 0110
0011	1101 0001
0100	0000 0000
0101	0000 0000
0110	0000 0001
0111	0000 0001

4

Let's assume our (tiny!) memory consists of 8 words, each having 8 bits. To simplify things later, we will assume that addresses' bits are a power of 2, that is 4-bits instead of the minimum needed, which would be 3. Our RAM will hold both the instructions of a small program and the results of the program as it's executed.

Registers

- Let's also assume that we have 1 register, the **accumulator**, which can hold one word, i.e. 8 bits.
- And two special registers:
  - the **program counter**, which has the address of the next instruction, and
  - the **instruction register**, which stores the instruction currently being executed.
- The program counter and instruction register are part of the control unit of the CPU while the accumulator is part of the arithmetic logic unit.



5

Let's also assume that we have 1 register, the accumulator, which can hold one word, i.e. 8 bits. And two special registers: i) the program counter, which has the address of the next instruction to be executed, and ii) the instruction register, which stores the instruction currently being executed. The program counter and instruction register are part of the control unit of the CPU while the accumulator is part of the arithmetic logic unit.

Instruction table

- Rather than working with binary numbers for the instructions, we will assume that an instruction can be specified by a 4-bit opcode (operation code) followed by a 4-bit address (addr). For example, for our program, the four instructions we are interested in are:

opcode	Instruction	Description
0001	LOAD addr	Load into accumulator the contents of the RAM at addr
1011	ADD addr	Add to accumulator the value stored in RAM at addr
0100	STORE addr	Store the contents of accumulator into RAM at address addr
1101	JUMP addr	Update program counter with the specified addr

- Such a low-level programming language is known as **assembly**. An **assembler** is a special program that reads text-based instructions like the ones above and translates them into the binary **machine code** (i.e. binary) that is truly the only language that computers understand.

6

Rather than working with binary numbers for the instructions, we will assume that an instruction can be specified by a 4-bit opcode (operation code) followed by a 4-bit address, described as addr here. For example, for our program, the four instructions we are interested can be seen in the table. For example, if we see the opcode 0001, we know that this instruction means that we should load into the accumulator the contents of the RAM at the specified address that will follow. 1011 means that we should add to the accumulator whatever value is stored in RAM at addr. 0100 means that we should store the contents of accumulator into RAM at the specified address. And 1101 means to update the program counter with the specified address. Such low-level programming requires an assembler, a special program to read text-based instructions like the ones above written in assembly and translate them into binary machine code that is the only language that computers truly understand.

Our (more) readable program

opcode	Instruction
0001	LOAD addr
1011	ADD addr
0100	STORE addr
1101	JUMP addr

- Using the instruction table we just saw, the RAM contents are more human-readable.
- For simplicity, let's also use decimal numbers instead of binary addresses

Address	Data
0000	0001 0110
0001	1011 0111
0010	0100 0110
0011	1101 0001
0100	0000 0000
0101	0000 0000
0110	0000 0001
0111	0000 0001

Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

7

Using the instruction table we just saw and translating addresses to decimal numbers, our program is a lot more readable. It consists of 4 instructions and as of now, the addresses 4-7 contain the numbers 0,0, 1, 1.

Clock and fetch-decode-execute cycle

- Our CPU has a clock that acts as the “orchestra conductor.” All components are synchronized with the clock.
- With every tick of the clock, the CPU will go through a step of the **fetch-decode-execute cycle** or **instruction cycle** to execute the instructions of a program.
- The instruction cycle repeats continuously the following steps:
  - **Fetch**:
    - The CPU uses the program counter to locate the next instruction from RAM and put it into the instruction register.
  - **Decode**:
    - The CPU reads the fetched instruction from the instruction register and determines what operation needs to be performed.
  - **Execute**:
    - The CPU performs the operation specified by the instruction, which might involve accessing data, performing calculations, or controlling other parts of the computer.

8

Our CPU has a clock that acts as the conductor and synchronizes everyone. With every tick of the clock, the CPU will go through a step of the fetch-decode-execute cycle or instruction cycle to execute the instructions of a program.

The instruction cycle repeats continuously the following steps:

Fetch:

The CPU uses the program counter to locate the next instruction from RAM and put it into the instruction register.

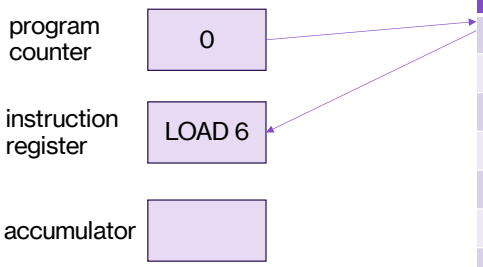
Decode:

The CPU reads the fetched instruction from the instruction register and determines what operation needs to be performed.

Execute:

The CPU performs the operation specified by the instruction, which might involve accessing data, performing calculations, or controlling other parts of the computer.

Executing our program - Fetch

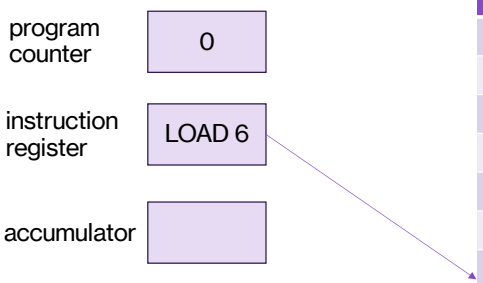


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

9

First clock tick. We are at the fetch step. Program counter is set to 0. The CPU retrieves the instruction LOAD 6 at address 0 and puts in instruction register the instruction LOAD 6.

Executing our program - Decode

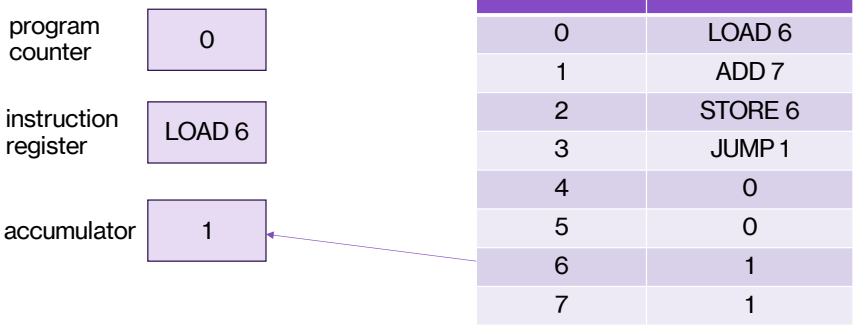


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

10

Second clock tick. We are at the decode step. The CPU decodes LOAD 6 as "Load into accumulator the contents of the RAM at address 6".

Executing our program - Execute

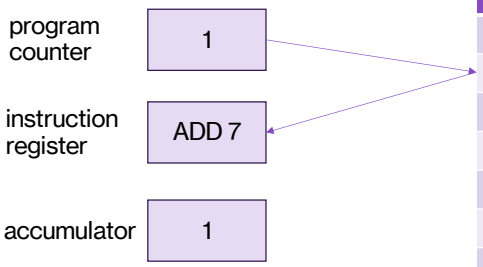


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

11

Third clock tick. We are at the execute step. The CPU loads the contents of address 6 (i.e. the number 1) into the accumulator.

Executing our program - Fetch

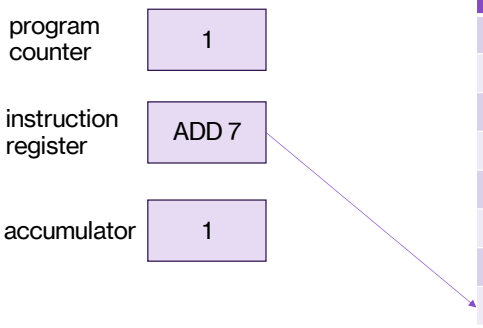


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

12

Fourth clock tick. We are at the fetch step. Program counter is increased to 1. The CPU retrieves the instruction ADD 7 at address 1 and puts in instruction register the instruction ADD 7.

Executing our program - Decode

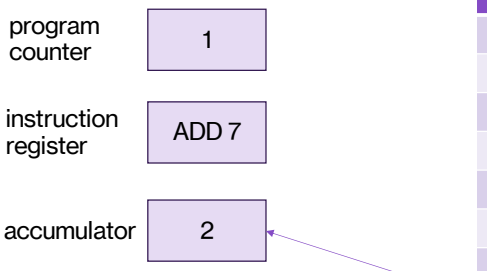


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

13

Fifth clock tick. We are at the decode step. The CPU decodes ADD 7 as “Add to the accumulator the value stored in RAM at address 7”.

Executing our program - Execute

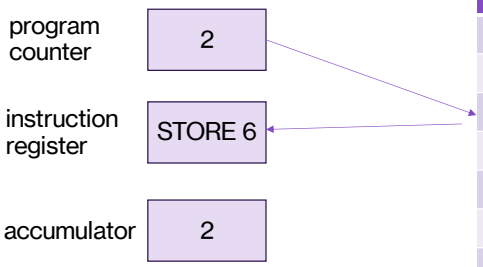


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

14

Sixth clock tick. We are at the execute step. The CPU adds to the existing value of accumulator (1) the value at address 7 (1), that is it updates the accumulator to 2.

Executing our program - Fetch

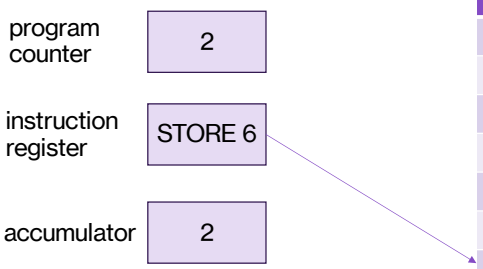


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

15

Seventh clock tick. We are at the fetch step. Program counter is increased to 2. The CPU retrieves the instruction STORE 6 at address 2 and puts in instruction register the instruction STORE 6.

Executing our program - Decode

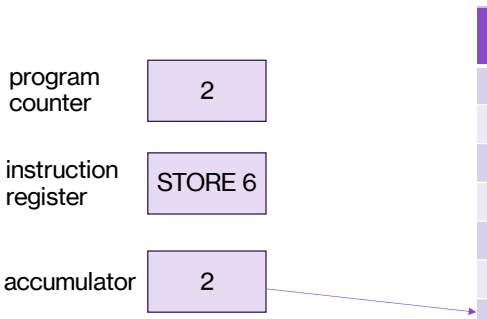


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	1
7	1

16

Eighth clock tick. We are at the decode step. The CPU decodes STORE 6 as "Store the contents of the accumulator into RAM at address 6"

Executing our program - Execute

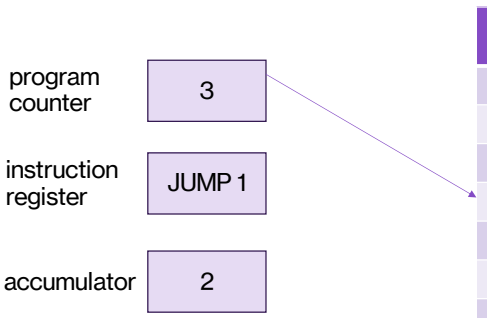


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

17

Ninth clock tick. We are at the execute step. The CPU stores the contents of the accumulator into memory address 6, that is it stores the number 2 in memory address 6.

Executing our program - Fetch

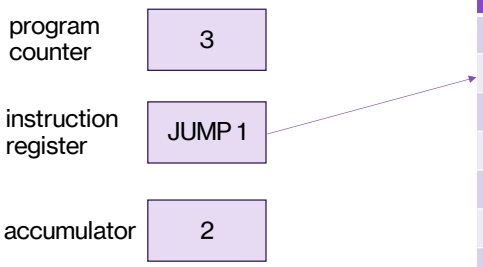


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

18

Tenth clock tick. We are at the fetch step. Program counter is increased to 3. The CPU retrieves the instruction JUMP at address 1 and puts in instruction register the instruction JUMP 1.

Executing our program - Decode

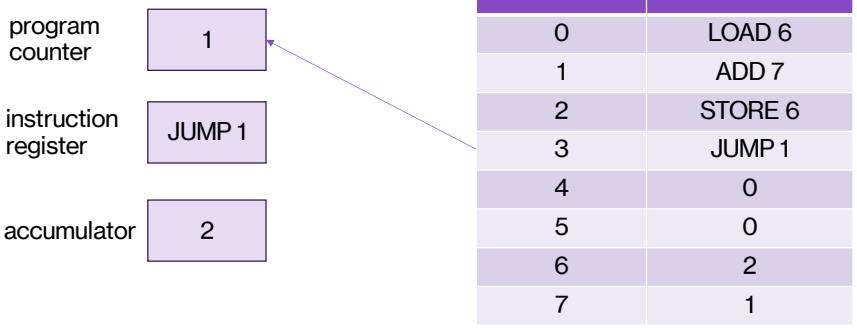


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

19

Eleventh clock tick. We are at the decode step. The CPU decodes JUMP 1 as "Update program counter with the specified address, i.e. 1"

Executing our program - Execute

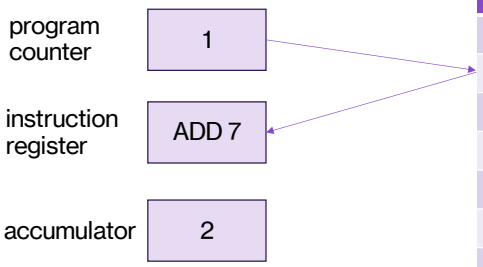


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

20

Twelfth clock tick. We are at the execute step. The CPU updates the program counter to 1.

Executing our program - Fetch

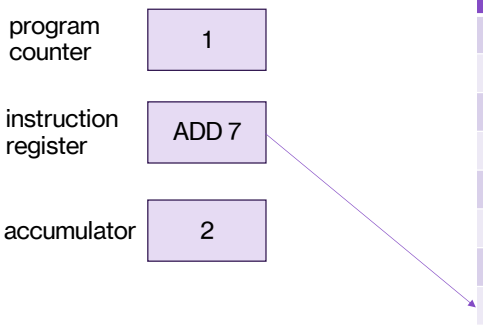


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

21

Thirteenth clock tick. We are at the fetch step and the program counter is back at 1, looping us very close to the beginning of our program! The CPU retrieves the instruction ADD 7 at address 1 and puts in instruction register the instruction ADD 7.

Executing our program - Decode

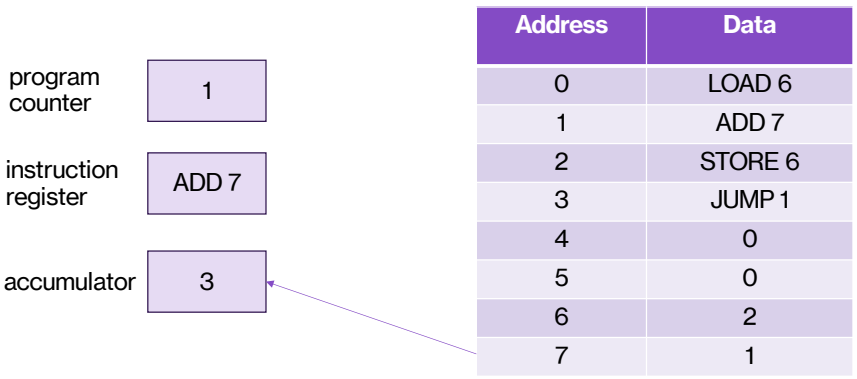


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

22

Fourteenth clock tick. We are at the decode step. The CPU decodes ADD 7 as "Add to the accumulator the value stored in RAM at address 7".

Executing our program - Execute

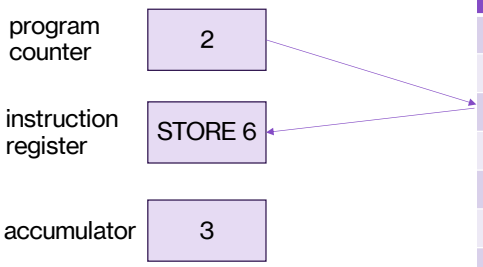


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

23

Fifteenth clock tick. We are at the execute step. The CPU adds to the existing value of accumulator (2) the value at address 7 (1), that is it updates the accumulator to 3.

Executing our program - Fetch

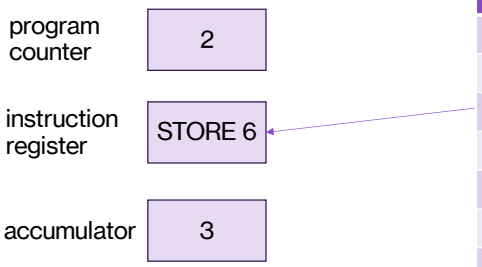


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

24

Sixteenth clock tick. We are at the fetch step. Program counter is increased to 2. The CPU retrieves the instruction STORE 6 at address 2 and puts in instruction register the instruction STORE 6.

Executing our program - Decode

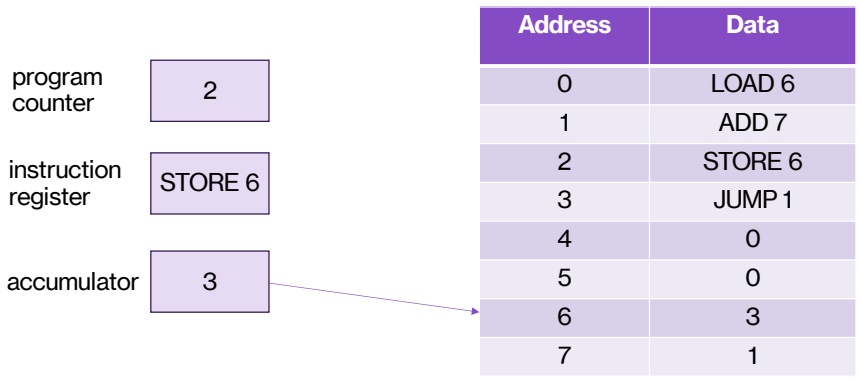


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	2
7	1

25

Seventeenth clock tick. We are at the decode step. The CPU decodes STORE 6 as “Store the contents of accumulator into RAM at address 6”

Executing our program - Execute

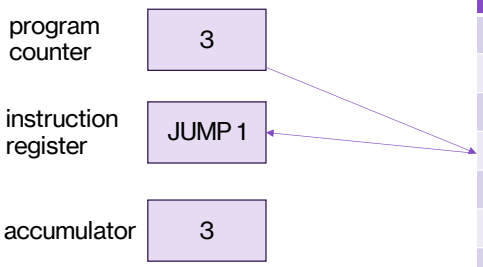


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	3
7	1

26

Eighteenth clock tick. The CPU stores the contents of the accumulator, i.e. the number 3, into memory address 6.

Executing our program - Fetch

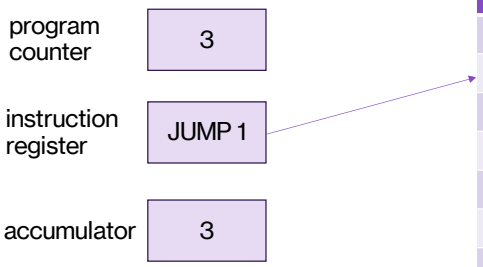


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	3
7	1

27

Nineteenth clock tick. We are at the fetch step. Program counter is increased to 3. The CPU retrieves the instruction JUMP 1 at address 3 and puts in instruction register the instruction JUMP 1.

Executing our program - Decode

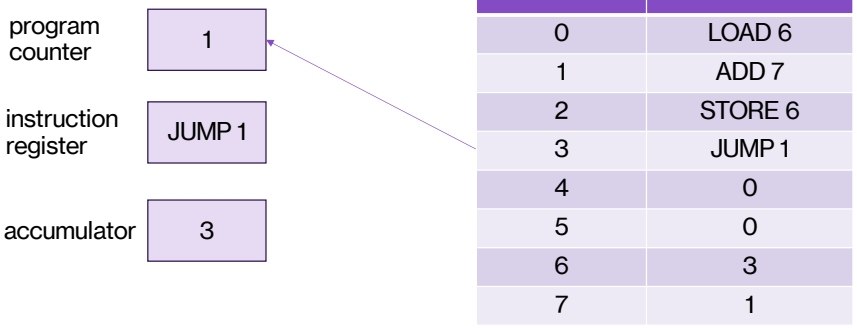


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	3
7	1

28

Twenteeth clock tick. We are at the decode step. The CPU decodes JUMP 1 as “Update program counter with the specified address, i.e. 1”

Executing our program - Execute

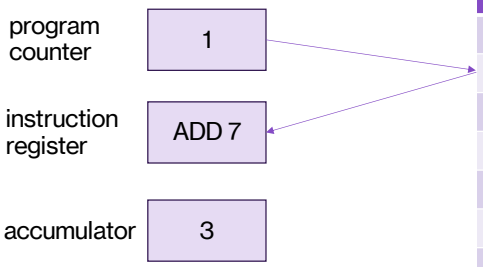


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	3
7	1

29

Twenty-first clock tick. We are at the execute step. The CPU updates the program counter to 1.

Executing our program - Fetch

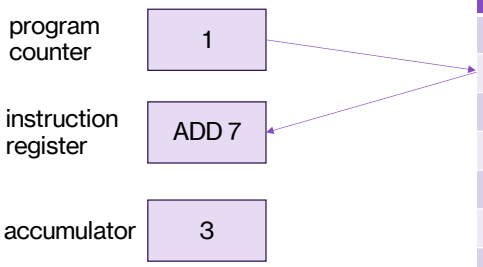


Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	3
7	1

30

Twenty-second clock tick. We are at the fetch step and the program counter is back at 1, looping us very close to the beginning of our program! The CPU retrieves the instruction ADD 7 at address 1 and puts in instruction register the instruction ADD 7.

What does this program do??



Address	Data
0	LOAD 6
1	ADD 7
2	STORE 6
3	JUMP 1
4	0
5	0
6	3
7	1

31

Our program

- It starts with the value 1 at address 6 and it continuously increments it by 1.
  - The increment happens on every ninth clock cycle.

From 0s and 1s to high-level programming

- Working with 0s and 1s (machine code) or even assembly code is rather hard. This is where high-level programming languages come to play.
- The program we saw would be similar in function to the following Python code. But code in high-level programming languages like Python is much more readable and quickly conveys its functionality:

Machine Code	Assembly Program	High-Level Language (e.g., Python)

0001 0110
1011 0111
0100 0110
1101 0001
0000 0000
0000 0000
0000 0001
0000 0001
	LOAD 6
	ADD 7
	STORE 6
	JUMP 1
	0
	0
	1
	1
		`x = 1`
		`while (True):`
		`  x = x + 1`

33

Practice Time

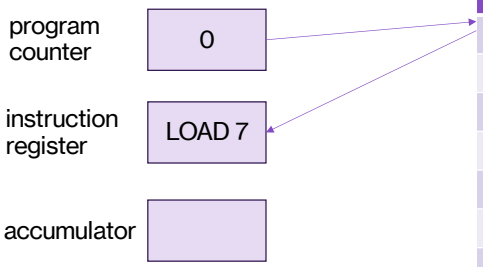
- Assume you are given the following program. What does it do?

Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

34

Your turn to practice. What does the following program do?

Executing our program - Fetch

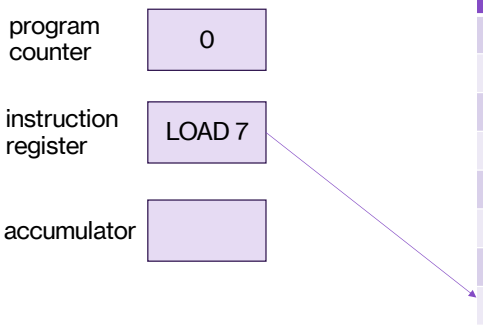


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

35

First clock tick. We are at the fetch step. Program counter is set to 0. The CPU retrieves the instruction LOAD 7 at address 0 and puts in instruction register the instruction LOAD 7.

Executing our program - Decode

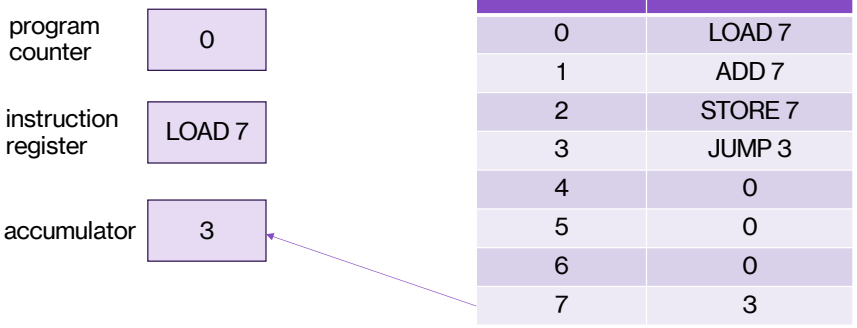


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

36

Second clock tick. We are at the decode step. The CPU decodes LOAD 7 as “Load into accumulator the contents of the RAM at address 7”.

Executing our program - Execute

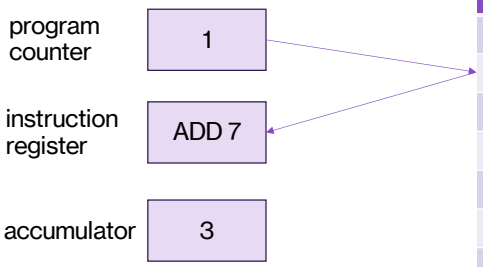


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

37

Third clock tick. We are at the execute step. The CPU loads the contents of address 7 (i.e. the number 3) into the accumulator.

Executing our program - Fetch

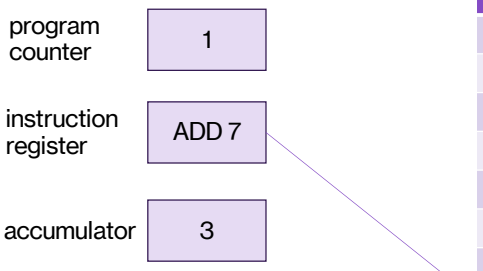


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

38

Fourth clock tick. We are at the fetch step. Program counter is increased to 1. The CPU retrieves the instruction ADD 7 at address 1 and puts in instruction register the instruction ADD 7.

Executing our program - Decode

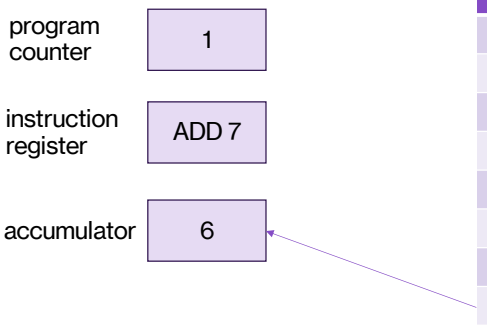


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

39

Fifth clock tick. We are at the decode step. The CPU decodes ADD 7 as “Add to the accumulator the value stored in RAM at address 7”.

Executing our program - Execute

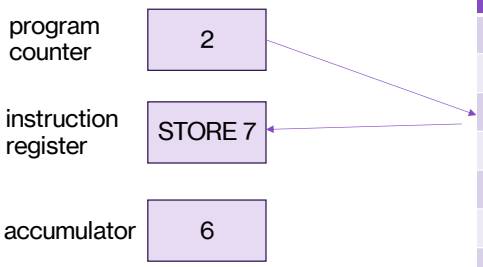


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

40

Sixth clock tick. We are at the execute step. The CPU adds to the existing value of accumulator (3) the value at address 7 (3), that is it updates the accumulator to 6.

Executing our program - Fetch

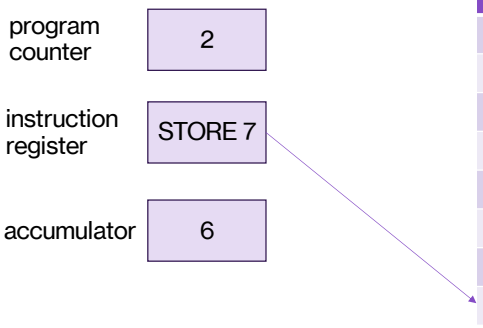


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

41

Seventh clock tick. We are at the fetch step. Program counter is increased to 2. The CPU retrieves the instruction STORE 7 at address 2 and puts in instruction register the instruction STORE 7.

Executing our program - Decode

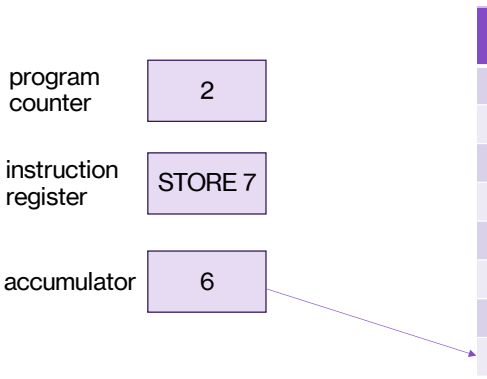


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	3

42

Eighth clock tick. We are at the decode step. The CPU decodes STORE 7 as "Store the contents of the accumulator into RAM at address 7"

Executing our program - Execute

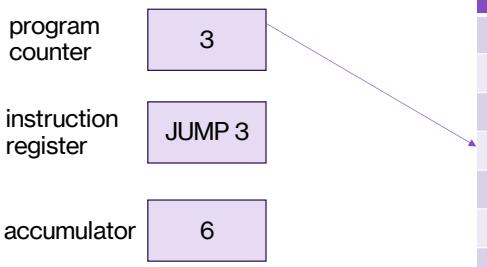


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	6

43

Ninth clock tick. We are at the execute step. The CPU stores the contents of the accumulator into memory address 7, that is it stores the number 6 in memory address 7.

Executing our program - Fetch

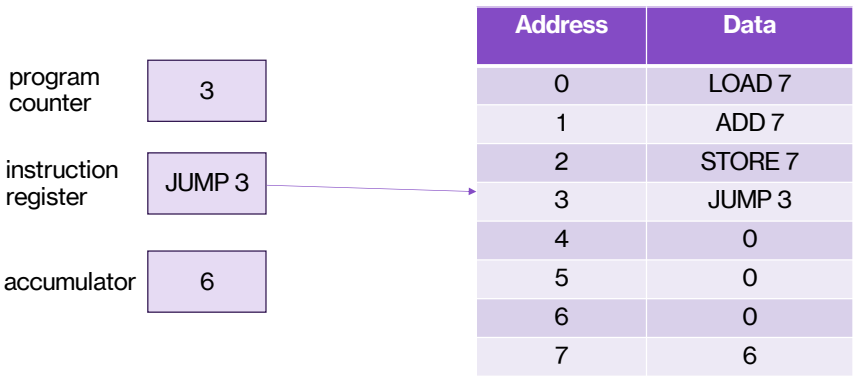


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	6

44

Tenth clock tick. We are at the fetch step. Program counter is increased to 3. The CPU retrieves the instruction LOAD at address 7 and puts in instruction register the instruction LOAD 7.

Executing our program - Decode

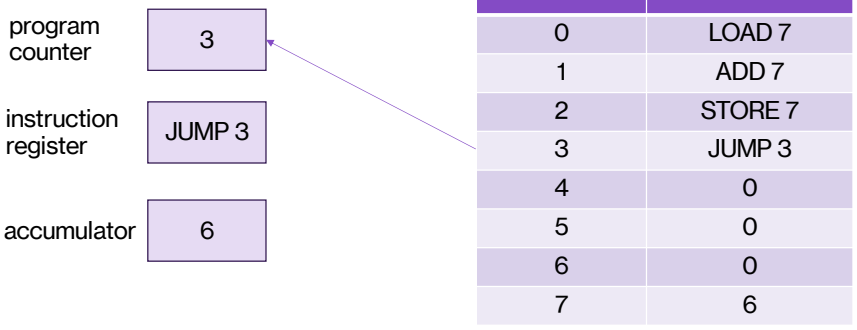


Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	6

45

Eleventh clock tick. We are at the decode step. The CPU decodes JUMP 3 as "Update program counter with address 3"

Executing our program - Execute



Address	Data
0	LOAD 7
1	ADD 7
2	STORE 7
3	JUMP 3
4	0
5	0
6	0
7	6

46

Twelfth clock tick. We are at the execute step. The CPU updates the program counter to 3. We are getting into an infinite loop.

Answer

- It doubles the value at address 7 once (i.e. it doubles 3 to 6), then loops infinitely.

In practice

- In practice, there are a few more registers and more instructions supported, e.g., to halt the execution of the program, call and return from subroutines (functions), jump when certain conditions are met to reflect control flow, and streamline standard logical and arithmetic operations.

48

In practice, there are a few more registers and more instructions supported, e.g., to halt the execution of the program, call and return from subroutines (functions), jump when certain conditions are met to reflect control flow, and streamline standard logical and arithmetic operations.