CPU Organisation and Operation

The Fetch-Execute Cycle

The operation of the CPU[1] is usually described in terms of the Fetch-Execute cycle.[2]

Fetch-Execute Cycle	**The cycle raises many interesting questions, e.g.**
Fetch the *Instruction*	What is an Instruction? Where is the Instruction? Why does it need to be fetched? Isn't it okay where it is? How does the computer keep track of instructions? Where does it put the instruction it has just fetched?
Increment the *Program Counter*	What is the Program Counter? What does the Program Counter count? Increment by how much? Where does the Program Counter point to after it is incremented?
Decode the Instruction	Why does the instruction need to be decoded? How does it get decoded?
Fetch the *Operands*	What are operands? What does it mean to fetch? Is this fetching distinct from the fetching in Step 1 above? Where are the operands? How many are there? Where do we put the operands after we fetch them?
Perform the Operation	Is this the main step? Couldn't the computer simply have done this part? What part of the CPU performs this operation?
Store the results	What results? Where from? Where to?
Repeat forever	Repeat what? Repeat from where? Is it really an infinite loop? Why? How do these steps execute any instructions at all?

In order to appreciate the operation of a computer, we need to answer such questions and to consider in more detail the organisation of the CPU.

Representing Programs

Each complex task carried out by a computer needs to be broken down into a sequence of simpler tasks and a **binary machine instruction** is needed for the most primitive tasks. Consider a task that adds two numbers,[3] held in memory locations designated by B and C[4] and stores the result in memory location designated by A.

$$A = B + C$$

1 Central Processing Unit.
2 Sometimes called the Fetch-Decode-Execute Cycle.
3 Let's assume they are held in two's complement form.
4 A, B and C are actually main memory **addresses**, i.e. natural binary numbers.

This assignment can be broken down (compiled) into a sequence of simpler tasks or **assembly instructions**, e.g.

Assembly Instruction	Effect
LOAD R2, B	Copy the contents of memory location designated by B into Register 2
ADD R2, C	Add the contents of the memory location designated by C to the contents of Register 2 and put the result back into Register 2
STORE R2, A	Copy the contents of Register 2 into the memory location designated by A.

Each of these assembly instructions needs to be encoded into binary for execution by the Central Processing Unit (CPU). Let us try this encoding for TOY1, a simple architecture.

TOY1 Architecture

TOY1 is a fictitious architecture with the following characteristics:

16-bit words of RAM. RAM is word-addressable.

4 general purpose registers R0, R1, R2 and R3. Each general purpose register is 16 bits wide, the same width as a memory location.

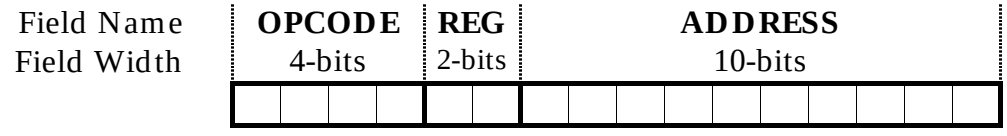
16 different instructions that the CPU can decode and execute, e.g. LOAD, STORE, ADD, SUB and so on. These different instructions constitute the **Instruction Set** of the Architecture.

The representation for integers will be two's complement.

For this architecture, the computer architect needs to define a coding scheme[5] for instructions. This is termed the **Instruction Format**.

TOY1 Instruction Format

TOY1 instructions are 16 bits, to fit into a main-memory word. Each instruction is divided into a number of **instruction fields** that encode a different piece of information for the CPU.



The **OPCODE**[6] field identifies the CPU operation. Since TOY1 supports 16 instructions, these can be encoded as a 4-bit natural number. For TOY1, opcodes 1 to 4 will be:[7]

0001 = LOAD 0010 = STORE 0011 = ADD 0100 = SUB

5 Most architectures actually have different instruction formats for different categories of instruction.

6 Operation Code

7 The meaning of CPU operations is defined in the Architecture's Instruction Set Manual.

The **REG** field defines a General CPU Register. Arithmetic operations will use 1 register **operand** and 1 main memory **operand**, results will be written back to the register. Since TOY1 has 4 registers; these can be encoded as a 2-bit natural number:

00 = Register 0 01 = Register 1 10 = Register 2 11 = Register 3

The **ADDRESS** field defines the address of a word in RAM. Since it is 10 bits long, it means that TOY1 can address up to 1024 memory locations using this method; but think about other methods that enable more memory locations to be addressed. If we define addresses 200H, 201H and 202H for A, B and C, we can encode the example above as:

Assembly Instruction		**Machine Instruction**
LOAD	R2, [201H]	0001 10 10 0000 0001
ADD	R2, [202H]	0011 10 10 0000 0010
STORE	R2, [200H]	0010 10 10 0000 0000

Memory Placement of Program and Data

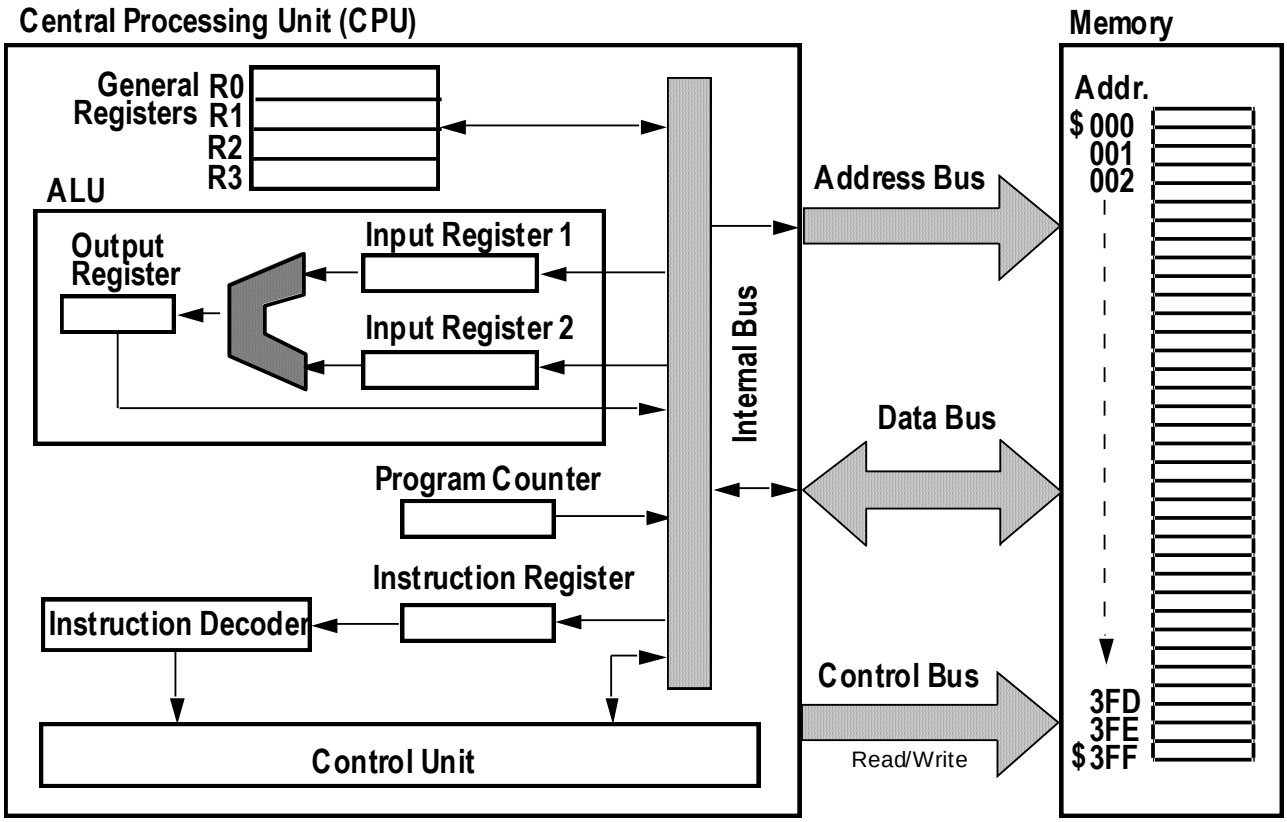
In order to execute a TOY1 program, its instructions and data needs to placed within main memory.[8] We'll place our 3-instruction program in memory starting at address 080H and we'll place the variables A, B and C at memory words 200H, 201H, and 202H respectively. Such placement results in the following memory layout prior to program execution. For convenience, memory addresses and memory contents are also given in hex.

Memory Address in binary & hex			**Machine Instruction** OP Reg Address				**Assembly Instruction**
0000 0	1000 8	0000 0	0001 1	10 A	0000 0	0001 1	LOAD R2, [201H]
0000 0	1000 8	0001 1	0011 3	10 A	0000 0	0010 2	ADD R2, [202H]
0000 0	1000 8	0010 2	0010 2	10 A	0000 0	0000 0	STORE R2, [200H]
Etc			**Etc**				**Etc**
0010 2	0000 0	0000 0	0000 0	0000 0	0000 0	0000 0	A = 0
0010 2	0000 0	0001 1	0000 0	0000 0	0000 0	1001 9	B = 9
0010 2	0000 0	0010 2	0000 0	0000 0	0000 0	0110 6	C = 6

Of course, the big question is: "How is such a program executed by the TOY1 CPU?"

8 The Operating System software is normally responsible for undertaking this task.

CPU Organisation



The **Program Counter (PC)** is a special register that holds the **address** of the next instruction to be fetched from Memory (for TOY1, the PC is 16 bits wide). The PC is incremented[9] to "point to" the next instruction while an instruction is being fetched from main memory.

The **Instruction Register (IR)** is a special register that holds each instruction after it is fetched from main memory. For TOY1, the IR is 16 bits since instructions are 16-bit wide.

The **Instruction Decoder** is a CPU component that decodes and interprets the contents of the Instruction Register, i.e. it splits a whole instruction into fields for the Control Unit to interpret. The Instruction decoder is often considered to be a part of the Control Unit.

The **Control Unit** is the CPU component that co-ordinates all activity within the CPU. It has connections to all parts of the CPU, and includes a sophisticated timing circuit.

The **Arithmetic & Logic Unit (ALU)** is the CPU component that carries out arithmetic and logical operations e.g. addition, comparison, boolean AND/OR/NOT.

The **ALU Input Registers 1 & 2** are special registers that hold the input operands for the ALU.

The **ALU Output Register** is a special register that holds the result of an ALU operation. On completion of an ALU operation, the result is copied from the ALU Output register to its final destination, e.g. to a CPU register, or main-memory, or to an I/O device.

---

9 By the appropriate number of memory words.

The **General-Purpose Registers R0, R1, R2, R3** are available for the programmer to use in his/her programs. Typically the programmer tries to maximise the use of these registers in order to speed program execution. For TOY1, the general registers are the same size as memory locations, i.e. 16 bits.

The **Buses** serve as communication highways for passing information within the CPU (CPU internal bus) and between the CPU and the main memory (the **address bus**, the **data bus**, and the **control bus**). The address bus is used to send addresses from the CPU to the main memory; these addresses indicate the memory location the CPU wishes to read or write. Unlike the address bus, the data bus is bi-directional; for writing, the data bus is used to send a word from the CPU to main-memory; for reading, the data bus is used to send a word from main-memory to the CPU. For TOY1, the Control bus[10] is used to indicate whether the CPU wishes to read from a memory location or write to a memory location. For simplicity we've omitted two special registers, the **Memory Address Register (MAR)** and the **Memory Data Register (MDR).** These registers lie at the boundary of the CPU and Address bus and Data bus respectively and serve to buffer data to/from the buses.

Buses can normally transfer more than 1-bit at a time. For the TOY1, the address bus is 16 bits (so the size of an address can be up to 16 bits), the data bus is 16 bits (the size of a memory location), and the control bus is 1-bit (to indicate a memory read operation or a memory write operation).

Interlude: the Von Neumann Machine Model

Most computers conform to the von Neumann machine model, named after the Hungarian-American mathematician John von Neumann (1903-57).

In von Neumann's model, a computer has **3 subsystems** (i) a CPU, (ii) a main memory, and (iii) an I/O system. The main memory holds the program as well as data and the computer is allowed to manipulate its own program.[11] Instructions are executed **sequentially** (one at a time). A single path exists between the control until and main-memory, this leads to the so-called "**von Neumann bottleneck**" since memory fetches are the slowest part of an instruction they become the bottleneck in any computation.

Instruction Execution (Fetch-Execute-Cycle Micro-steps)

In order to execute our 3-instruction program, the control unit has to issue and coordinate a series of micro-instructions. These micro-instructions form the fetch-execute cycle. For our example we will assume that the Program Counter register (PC) already holds the address of the first instruction, namely 080H.

---

10 Most control-buses are wider than a single bit, these extras bits are used to provide more sophisticated memory operations and I/O operations.

11 This type of manipulation is not regarded as a good technique for general assembly programming.

LOAD R2, [201H]

0000	1000	0000	0001	10	10	0000	0001
0	8	0	1	A		0	1

Copy the value in memory word 201H into Register 2

Control Unit Action	**Data flows**			
FETCH INSTRUCTION[12]				
PC to Address Bus[13]	080H	→	080H	Address Bus
0 to Control Bus[14]	0	→	0	Control Bus
Address Bus to Memory	080H	→	080H	Memory
Control Bus to Memory	0	READ →	0	Memory
Increment PC[15]	080	INC →	081H	PC becomes PC+1[16]
Memory [080H] to Data Bus	1A01H	→	1A01H	Data Bus
Data Bus to Instruction Register	1A01H	→	1A01H	Instruction Register
DECODE INSTRUCTION				
IR to Instruction Decoder	1A01H	→	1A01H	Instruction Decoder
Instruction Decoder to Control Unit[17]	1, 2, 201H	→	1, 2, 201H	Control Unit
EXECUTE INSTRUCTION[18]				
Control Unit to Address Bus	201H	→	201H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	201H	→	201H	Memory
Control Bus to Memory	0	READ →	0	Memory
Memory [201H] to Data bus	0009H	→	0009H	Data Bus
Data Bus to Register 2	0009H	→	0009H	Register 2

12 The micro-steps in the Fetch and Decode phases are common for all instructions.

13 This and the next 4 micro-steps initiate a fetch of the next instruction to be executed, which is to found at memory address 80H. In practice a Memory Address Register (MAR) acts as an intermediate buffer for the Address, similarly a Memory Data Register (MDR) buffers data to/from the data bus.

14 We will use 0 for a memory READ request, and 1 for a memory WRITE request.

15 For simplicity, we will assume that the PC is capable of performing the increment internally. If not, the Control Unit would have to transfer the contents of the PC to the ALU, get the ALU to perform the increment and send the results back to the PC. All this while we are waiting for the main-memory to return the word at address 80H.

16 Since TOY1's main-memory is word-addressed, and all instructions are 1 word. If main-memory was byte-addressed we would need to add two.

17 The Instruction decoder splits the instruction into the individual instruction fields OPCODE, REG and ADDRESS for interpretation by the Control Unit.

18 The micro-steps for the execute phase actually perform the operation.

ADD R2, [202H]

0000	1000	0001
0	8	1

0011	10	10	0000	0002
3	A		0	2

Add[19] the value in memory word 202H to Register 2

Control Unit Action		Data flows		
FETCH INSTRUCTION				
PC to Address Bus	081H	→	081H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	081H	→	081H	Memory
Control Bus to Memory	0	**READ** →	0	Memory
Increment PC	081H	**INC** →	082H	PC becomes PC+1
Memory [081H] to Data Bus	3A02H	→	3A02H	Data Bus
Data Bus to Instruction Register	3A02H	→	3A02H	Instruction Register
DECODE INSTRUCTION				
IR to Instruction Decoder	3A02H	→	3A02H	Instruction Decoder
Instruction Decoder to Control Unit	3, 2, 202H	→	3, 2, 202H	Control Unit
EXECUTE INSTRUCTION				
Register 2 to ALU Input Reg 1	0009	→	0009	ALU Input Reg 1
Control Unit to Address Bus	202H	→	202H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	202H	→	202H	Memory
Control Bus to Memory	0	**READ** →	0	Memory
Memory [202H] to Data bus	0006H	→	0006H	Data Bus
Data Bus to ALU Input Reg 2	0006H	→	0006H	ALU Input Reg 2
Control Unit to ALU		**ADD** →	000FH	Output Register
ALU Output Reg to Register 2	000F	→	000FH	Register 2

[19] Using two's complement arithmetic.

STORE R2, [200H]

0000	1000	0001
0	8	2

0010	10	10	0000	0000
2	A		0	0

Copy the value in Register 2 into memory word 202H

Control Unit Action		Data flows		
FETCH INSTRUCTION				
PC to Address Bus	082H	→	082H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	082H	→	082H	Memory
Control Bus to Memory	0	**READ** →	0	Memory
Increment PC	082H	**INC** →	083H	PC becomes PC+1
Memory [082] to Data Bus	2A00H	→	2A00H	Data Bus
Data Bus to Instruction Register	2A00H	→	2A00H	Instruction Register
DECODE INSTRUCTION				
IR to Instruction Decoder	2A00	→	2A00	Instruction Decoder
Instruction Decoder to Control Unit	2, 2, 200H	→	2, 2, 200H	Control Unit
EXECUTE INSTRUCTION				
Register 2 to Data Bus	000FH	→	000FH	Data Bus
Control Unit to Address Bus	200H	→	200H	Address Bus
1 to Control Bus	1	→	1	Control Bus
Data Bus to Memory	000FH	→	000FH	Memory
Address Bus to Memory	200H	→	200H	Memory
Control Bus to Memory	1	**WRITE** →	1	Memory