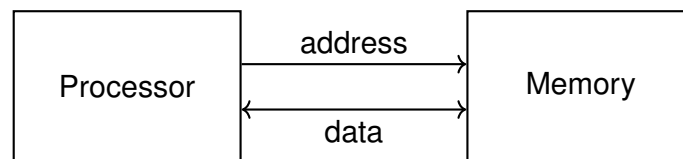


## Chapter 3 – The Von Neumann Architecture

### 1 Introduction

What does it mean for a computer to *execute a program* ?

**Idea** The hardware of a computer does not implement any built-in algorithm. Instead, the **processor** reads program **instructions** from memory and executes them one by one.



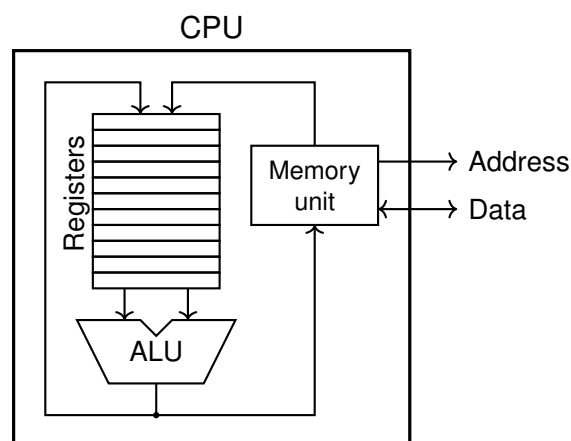
**Memory** is the hardware component where data and program instructions are stored. For historical reasons, memory cells (bits) are not accessed individually but in groups of 8 bits called **bytes**. Every byte in memory is designated by a unique number called its **address**. Memory addresses range from zero to some large number.

Memory behaves as an array of bytes, and the CPU can access it in two ways:

- To **read** from memory, we specify the address of the target location. The memory will respond with (a copy of) the bits stored there.
- To **write** to memory, we specify both an address and a value. The memory will overwrite the bits at that location with the new contents.

**The Central Processing Unit (CPU)** (sometimes just “the processor”) is the hardware component which executes machine instructions. It comprises two important parts:

- The **Arithmetic and Logic Unit (ALU)** is the circuitry performing the actual computations: additions, multiplications, logic operations, etc.
- The **Registers** are where all the values (i.e. operands and results) are stored. The CPU contains a handful of registers, each designated by a number (e.g. R0, R1, R2...).



The wires connecting CPU and memory are collectively known as the **system bus**. Sometimes we draw the “address bus” and “data bus” separately (like above), sometimes we will use just one arrow.

**The von Neumann principle** (aka stored-program concept) refers to the idea of representing both data and program instructions as bits stored in memory. The CPU implements the following behaviour known as the **von Neumann cycle**:

**While True do:**

**Fetch** (aka read, load) one word from memory  
**Decode** its bits: which operation, which operands, etc  
**Execute** the operation and write back the results into the register file  
 then advance to the next instruction and repeat the cycle

One register is dedicated to tracking execution progress: the **Program Counter**. It always holds the address of the current instruction. At the end of the cycle, the CPU increments its PC before going back to the Fetch step.

## 2 SCAT Machine Language

Each type of computer implements a different set of instructions. In this course we use an imaginary toy system called SCAT (Small Computer Architecture for Teaching) to learn the ropes of low-level programming. Some technicalities will not transfer exactly to real-world computer architectures (e.g. ARM or Intel x86) but all the ideas are the same.

**CPU Registers** Our CPU has 16 registers named R0 to R15. Each register holds 32 bits. Some registers are special: for instance R0 is hardwired with all bits equal to 0. Any writes to R0 are ignored. Register R15 is the Program Counter. Other registers (R1 to R14) are **general purpose registers** i.e. they are freely available for the programmer.

**Instruction format** Each SCAT instruction is stored in memory as a word of 32 bits, like illustrated below.<sup>1</sup> For clarity, we depict the bits in groups of 4. Indeed, any combination of four bits can be represented as single a hexadecimal digit, and it is easier for us humans to read 0x12345678 than the same value in binary notation: 0b10010001101000101011001111000.

31	28 27	24 23	20 19	16 15	12 11	8 7	4 3	0
• • • •	• • • •	• • • •	• • • •	• • • •	• • • •	• • • •	• • • •	• • • •

**Example: Addition** Within the instruction word, the most significant byte (bits 31–24, i.e. bits at positions 31 down to 24) indicates which instruction to execute. For instance to add (the contents of) two registers and write the sum into a register, this byte should be 0x10. When the CPU recognizes this pattern at the Decode step, it then goes on to find the operands:

- bits 23–20 indicate where the result should be stored, i.e. the **destination register**
- the two terms of the addition come from two registers whose numbers are given by bits 19–16 and 15–12, respectively. These are the **source registers**.

The remaining bits 11–0 are ignored (they're meaningless for this type of instruction).

In other words, the binary format of the ADD instruction is:

31	28 27	24 23	20 19	16 15	12 11	0
0 0 0 1	0 0 0 0	rd	rs1	rs2	ignored	

**Exercise 1** Suppose that the CPU contains the values illustrated below, and fetches this instruction word: 0x10538123. Indicate the state of the CPU after executing the instruction (you may omit unchanged registers).

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
0	7	0x1234	0x2A	0x10	0x24	0xF	0x43215678	0xFF	−5	0x42	0x10	0xCAFE	1	0x100	8

<sup>1</sup>This means that each Fetch step will load four bytes at once. And the PC will be incremented by four after each iteration of the cycle. But these are implementation details.

**Exercise 2** Assuming the same initial state: what is the effect of the instruction 0x10340000 ?

**Exercise 3** Assuming the same initial state: what is the effect of the instruction 0x1066D000 ?

**Exercise 4** Assuming the same initial state: what is the effect of the instruction 0x10EEE000 ?

**Example 2: Subtraction** is also implemented in our CPU. The corresponding **opcode** (i.e. the first byte of the instruction word) is 0x11. The binary format of the SUB instruction is:

31	28 27	24 23	20 19	16 15	12 11	0
0	0	0	1	0	0	0
1	0	0	0	1	rd	rs1
					rs2	ignored

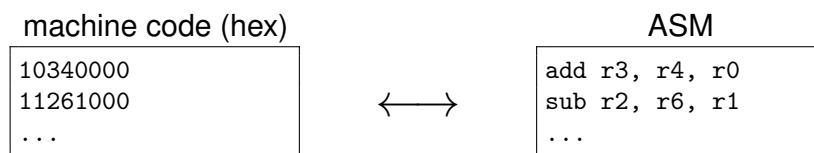
The effect of this instruction is to compute  $rs1 - rs2$  and store the result in  $rd$ .

**Exercise 5** Again, we assume that the CPU contains the values illustrated in exercise 1. What is the effect of the instruction 0x11261000 ?

**Exercise 6** Assuming the same initial state: what is the effect of the instruction 0x11189000 ?

### 3 Assembly Language

Working with machine code in binary form is not convenient for us humans. For this reason we will most often use an equivalent, textual representation called **assembly language** (aka assembly, asm, ASM...). In assembly source code, each instruction is described by a line of text, as illustrated below:



Converting an ASM file into binary form suitable for execution is called **assembling** the program, and is typically performed by a software tool called an **assembler**. The inverse operation (decoding a binary file into assembly) is called *disassembling*.

**ASM syntax** For register-register operations like addition and subtraction, the syntax is straightforward: we write `op rd, rs1, rs2` where “op” is the opcode **mnemonic**, e.g. add, sub, etc.

**Type 1 – Arithmetic and Logic Register-Register Operations** Our CPU offers many instructions similar to addition and subtraction, for instance multiplication, division (two instructions: one for the quotient, another one for the remainder) as well as bitwise (i.e. column-by-column) logic operations<sup>2</sup>:

asm	Name	opcode	Description
add	Addition	0x10	$rd = rs1 + rs2$
sub	Subtraction	0x11	$rd = rs1 - rs2$
mul	Multiplication	0x12	$rd = rs1 * rs2$
div	Integer Division	0x13	$rd = rs1 / rs2$
mod	Modulo	0x14	$rd = rs1 \% rs2$
or	Bitwise OR	0x15	$rd = rs1   rs2$
and	Bitwise AND	0x16	$rd = rs1 \& rs2$
xor	Bitwise exclusive OR	0x17	$rd = rs1 \wedge rs2$

**Exercise 7** Give the machine code for instruction `xor r3, r8, r1`.

<sup>2</sup>The full list is provided in the *instruction set reference sheet* provided on our Moodle page

## 4 Immediate Operands

**Idea** SCAT arithmetic and logic instructions exist in two variants: register-register operations (as we've seen so far) and register-immediate operations, where the second operand is a constant value. These constants are called **immediates**, because their values are immediately available from the instruction word and do not require a register or memory access.

In assembly code, register-immediate instructions have distinct mnemonics. For example writing `addi r1, r2, 4` will add 4 to the value of `r2` and store the result in `r1`.

Note: only the second operand can be an immediate constant: `addi r1, 4, r2` is a syntax error.

**Binary format** In machine code, register-immediate instructions are encoded in a similar way to register-register instructions:

- bits 31–28 encode the type of instruction. Type 1 (0b0001) is for register-register, while register-immediate is type 2 (0b0010).
- bits 27–24 indicate the ALU operation. The encoding is the same as for type 1 instructions.
- bits 23–20 and 19–16 indicate the destination and source registers, respectively.
- the value of the second operand is taken from bits 16–0.

31	28 27	24 23	20 19	16 15	0
0 0 1 0	op	rd	rs1	imm	

The example from above (`addi r1, r2, 4`) is encoded as 0x20120004.

**Exercise 8** Remember that register `R0` is hardwired to value zero. Give an instruction which sets register `R1` to value 7. Then assemble it to machine code.

**Sign Extension** When executing a type 2 instruction, the 16-bits immediate constant must be sign-extended to a 32-bits value before being fed to the ALU. For that, the CPU duplicates its *sign bit* (i.e. bit 15) all the way to bit 31. In other words, immediate constants are encoded in the instruction word as 16-bits signed numbers, like illustrated below.

decimal value	4	2020	32767	−32768	−7	−1
16 bits encoding	0x0004	0x07e4	0x7fff	0x8000	0xffff9	0xffff

**Type 2 – Arithmetic and Logic Register-Immediate Operations** The various instructions available are listed below. In assembly you may either write the constant as a decimal number (between -32768 and 32767) or a hexadecimal number (between 0 and 0xFFFF). You can even use binary notation e.g. 0b111 if it makes your code more readable.

asm	Name	opcode	Description
<code>addi</code>	Addition	0x20	$rd = rs1 + sxt(imm)$
<code>subi</code>	Subtraction	0x21	$rd = rs1 - sxt(imm)$
<code>mul</code>	Multiplication	0x22	$rd = rs1 * sxt(imm)$
<code>divi</code>	Integer Division	0x23	$rd = rs1 / sxt(imm)$
<code>modi</code>	Modulo	0x24	$rd = rs1 \% sxt(imm)$
<code>ori</code>	Bitwise OR	0x25	$rd = rs1   sxt(imm)$
<code>andi</code>	Bitwise AND	0x26	$rd = rs1 \& sxt(imm)$
<code>xori</code>	Bitwise exclusive OR	0x27	$rd = rs1 \wedge sxt(imm)$

**Exercise 9** Give two different instructions that will each set register `R9` to value −5. Then assemble these instructions to machine code.

## 5 Using the SCAT assembler and simulator

**Exercise 10** Download `scat.zip` from Moodle and extract it in the directory of your choice. Open a terminal window and navigate to that directory. From now on, everything will happen there.

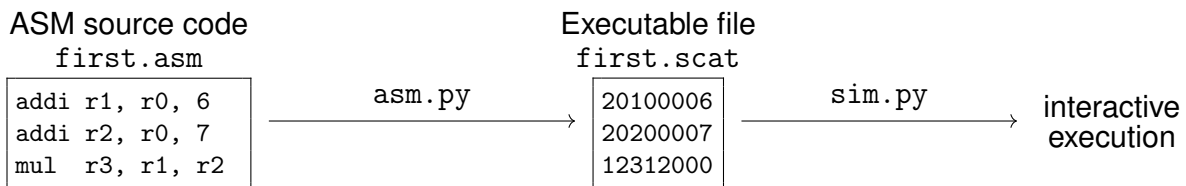
**Exercise 11** Type `python3 scat/asm.py --help` and then `python3 scat/sim.py --help` to get the corresponding help screens. If it does not work, do ask us for help !

**Exercise 12** In your favorite text editor, create a file named `first.asm` with this contents:

```
addi r1, r0, 6
addi r2, r0, 7
mul  r3, r1, r2
```

Note that assembly is case-insentive: writing `addi` is the same as `ADDI`, and `r1` is the same as `R1`.

**Exercise 13** Now type `python3 scat/asm.py first.asm` to run the assembler and produce your first **executable**<sup>3</sup>:



**Exercise 14** Type `python3 scat/sim.py first.scatscats` to start the simulator. By default, the simulator shows the contents of all sixteen CPU registers, as well as an exerpt from memory contents:

name	hex	dec	name	hex	dec
R0	00000000	0	R8	00000000	0
R1	00000000	0	R9	00000000	0
R2	00000000	0	R10	00000000	0
R3	00000000	0	R11	00000000	0
R4	00000000	0	R12	00000000	0
R5	00000000	0	R13	00000000	0
R6	00000000	0	R14	00000000	0
R7	00000000	0	PC	00000000	
<0000>					
	00: 20 10 00 06		addi r1, r0, 6		
	04: 20 20 00 07		addi r2, r0, 7		
	08: 12 31 20 00		mul r3, r1, r2		
(sim)					

**Remark** Each line in the “memory view” is a 32bits memory word: byte at address zero has value 0x20, then byte at address 1 is 0x10, byte at address 2 is 0, byte at address 3 is 0x06, etc. The text on the right is the original source code: the CPU doesn’t understand it at all, but it is useful for us humans to follow along.

**Exercise 15** In the simulator, type `step` to execute just one instruction. Observe how this changes the values of registers, and how we “advance” in the program.

What is the next instruction to be executed ? Try to predict its effect on the CPU, then type `step` again to verify your guess.

**Exercise 16** Go on with this **step-by-step execution** until the program is finished. Warning: beyond the end of our executable, memory contents is still random, so the *decode* step of the von-neumann cycle will fail with an “CPU error: illegal instruction” error message. Don’t worry, this is the expected behaviour for now. In later chapters, you will learn how to stop execution properly.

<sup>3</sup>In real life, executable files contain raw machine code. In this course however, we chose to store our “executables” as text files full of hexadecimal digits, so that you can read (and even modify) them in a text editor if you want. Try it !

**Exercise 17** In the simulator, type `help` to get a list of available commands. Then type `help step` to learn more about step-by-step execution.

**Exercise 18** Write a program that sets register `R5` to value `50000`.

**Exercise 19 (hard)** Write a program that sets all registers to the values illustrated in exercise 1. Don't hesitate to ask us for help !

**Remark** Like all programming languages, assembly allows for **comments** in the source code. You can use the semicolon character “`;`” as a **line comment marker**: on each line, the assembler will ignore any text that comes after a semicolon.

## 6 Pseudo-instructions

**Idea** The mapping between machine code and assembly language is almost one-to-one, but in certain situations, it is useful to work with some **alternate syntax** to make the assembly code more readable.

**The NOP pseudo-instruction** Even though it is perfectly legal, executing instruction `add r0, r1, r2` will have no visible effect: it is an example of a “do nothing” instruction, also known as a “**no operation**”. There are situations where the programmer actually wants to “do nothing” for one CPU cycle, but expressing that as an addition makes the source code harder to read. For this reason, our assembler offers the more readable `nop` mnemonic.

**Exercise 20** What is the machine code encoding of `nop` ?

**The LETI pseudo-instruction** As we've seen from exercises 18 and 19, even simple programming tasks like initializing a certain register to a certain value can prove challenging to implement in assembly. For this reason, our assembler offers the more readable `leti` opcode mnemonic: writing just `leti rd, value` will always initialize register `rd` to the requested `value`, even when doing so requires several machine instructions.

**Exercise 21** Solve exercises 18 and 19 again but with the `leti` pseudo-instruction, and compare the generated assembly to your hand-written solutions: are you able to reach some values with fewer instructions by hand than with `leti` ?

**Exercise 22** Open the *Instruction set reference sheet* (cf Moodle, or `scat/isca-card.pdf`) and read the last page on pseudo-instructions. Ask us about anything confusing.