[5.13] RAM

```
RAM with synchronous writes

module ram #(parameter N = 6, M = 32)
            (input logic clk,
             input logic we,
             input logic [N-1:0] adr,
             input logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];

endmodule
```

When creating a memory with SystemVerilog, we must define the bit-width of our data: defining it helps us understand how much addresses we need. We use as inputs the `clock`, the `Writing Enable` signal (`we`), the address `N` (which by default is 6, but it can be changed) and the bit-width `M` of the data we want that gets stored on the RAM (which follows the same principle of the address).

[5.14] ROM

```
module rom(input logic [1:0] adr,
           output logic [2:0] dout):

  always_comb
    case(adr)
      2'b00: dout = 3'b011;
      2'b01: dout = 3'b110;
      2'b10: dout = 3'b100;
      2'b11: dout = 3'b010;
    endcase
endmodule
```

Depending on what's the best solution, the synthetiser decides if it will be synthetised into logic gates or an array. This is an actual ROM circuit, there is no writing.

BEGINNING OF COMPUTER ARCHITECTURE UNIT 2

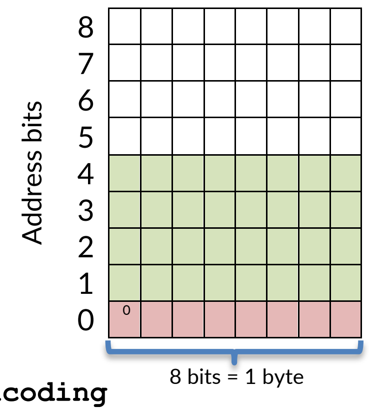
[6] Computer structure

A computer is made out of various components: CPU, memory, IO (Input/Output).

[6.1] Memory

Inside a computer, there are various type of memory. An example is the **`RAM`**, which is used to store data while the computer is running. RAM is volatile, so if power gets cut off then all the data stored will be deleted. Historically, RAM were made out of groups of 8 flip-flops. We have two types of RAM: **`Static`** and **`Dynamic`**.

- **Static RAM** (**SRAM**) is faster but more expensive than DRAM, and uses a different technology rather than flip-flops;
- **Dynamic RAM** (**DRAM**) is slower but cheaper, and is made out of capacitors; capacitors can "remember" a bit for a few milliseconds, so they need to be recharged every then and while; recharge though take time, so that is why DRAM is smaller;
- **Flash Memory** is slower than DRAM, but is cheaper, plus it's non-volatile.

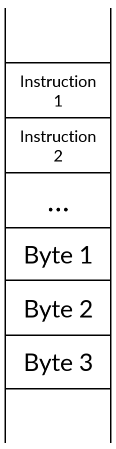


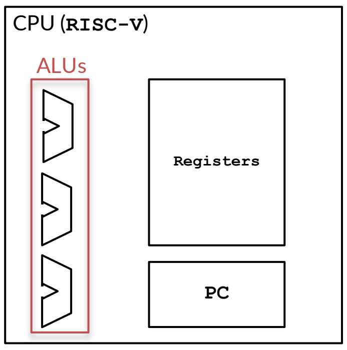
Generally in RAM, every level of flip-flops (so a byte) is marked by an address bit. We can use 1 byte to store integers, generally in 2's Complement notation (so a number x is stored following the equation $2^8 - x$). But numbers in 2's Complement can represent numbers in the range of [-128; 127], so what if we need to store bigger values?

Words for example are composed of 32-bits, so in a RAM we would need 4 bytes to store a word. We need though an encoding, and it's for this reason that in the 60's the **ASCII encoding** was created. ASCII encoding encodes every letter, every character into a number, so for example the letter **A** is **65**, **B** is **66**, **C** is **67**, etc… Even the new line has an encoding. Back in the days, Unix systems would encode the new line as 10, macOS as 13 and DOS based systems with both 10 and 13. This happened because in printers 10 would be used to roll down paper, while 13 was used to bring the head of the printer to the beginning of the line.

So how do computers encode words? We need various bits, such as for example a bit that expresses the length of a word (typically in the 70's the length was saved in the first byte, so that we would know in advance how much memory would be needed in order to store the word. Now this method isn't used anymore) or a bit that expresses when the word starts and one that expresses the end of the word.

[6.2] CPU





```
add 3, 1, 2
reg[3] ← reg[1] + reg[2]
```

What about a CPU? Imagine if in a CPU we have 3 bytes, and we want to store into the 3rd byte the sum between the 1st and the 2nd byte. What would we need? We would first need an **ALU**, capable of summing the data, and then a memory. In RISC-V processors we have a series of **32 registers**, which are called from **x0** to **x31**. An important thing about registers is that in RISC-V processors the register **x0** will always be full of zeroes, and if we try to store something in it, it won't work.

In machine code, we would ask the processor to do a given operation between the data contained into some registers. If we consider the example on the right, we would do the sum between register 1 and 2 and store the result into register 3. But how does a CPU remember what to do? Previously there was a separate memory built into the

CPU dedicated only to store the instructions, but now it's all built in into the main registers set. Each instruction needs 4 bytes to be saved, so each 4 bytes we can find an instruction, starting from the address 400 (so 400, 404, 408, 412, ...), while words for example start from address 1000. Each RISC-V CPU needs to remember the operation that he is doing, and that's where the **`PC`** (**`Program Counter`**) comes in handy: the PC is a temporary memory that gets edited each time that an operation is being done.

[6.3] Programs and instructions

We consider a **`program`** as a set of instructions, which is also called **`text`**. But how do we encode the instructions in the registers? It's important to say that there are different formats of instructions. Let us suppose that we have a simple instruction like **`add x3, x4, x5`**: we want to sum **`x4`** and **`x5`** and store their sum into **`x3`**. The encoding for add is the **`R-Type`**, and it works in this way:

F7							2nd register					1st register					F3			Destination Register					Opcode						
From 31 to 25							From 24 to 20					From 19 to 15					From 14 to 12			From 11 to 7					From 6 to 0						
0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	1

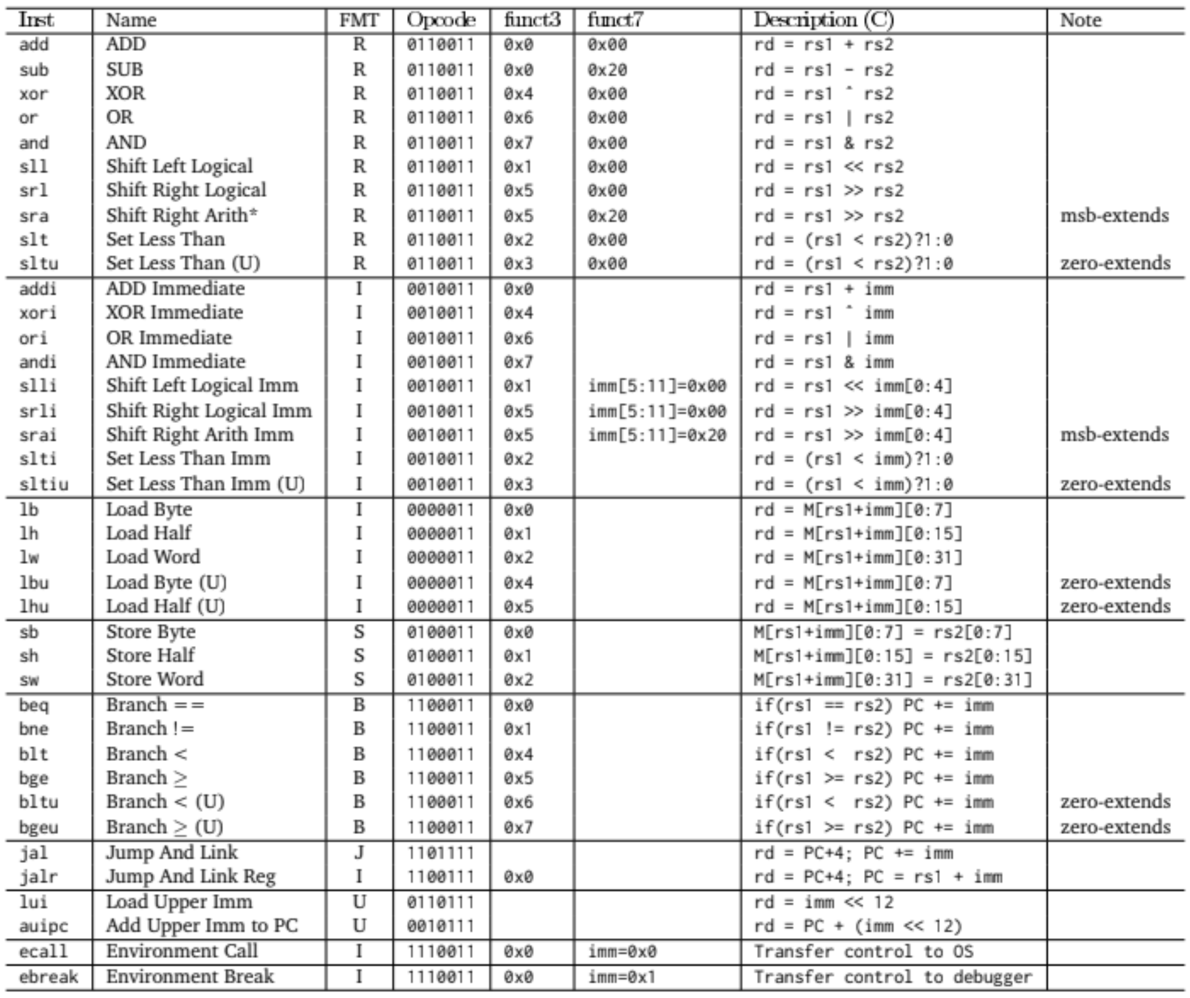
The encoding includes 6 parts:
- [0-6] **`Opcode`**: stands for **`O`**peration **`Code`**, it specifies the operation that must be done;
- [7-11] **`Destination Register`**: it's the address of the register where we want to store the result of our operation;
- [12-14] **`F3`**: it stands for function 3, and it's a further specification of the **`opcode`**. That happens because internally, in the CPU, much instructions are similar;
- [15-19] **`1st register`**: it's the address of the first register;
- [20-24] **`2nd register`**: it's the address of the second register;
- [25-31] **`F7`**: like **`F3`**, it stands for function 7, and it's a further specification of the operation that must be done.

Let's try with the subtraction instead:

Instruction: **`sub x7, x5, x7`**

F7							2nd register					1st register					F3			Destination Register					Opcode						
From 31 to 25							From 24 to 20					From 19 to 15					From 14 to 12			From 11 to 7					From 6 to 0						
0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0	1	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1

We note that the opcode is the same for both addition and subtraction, but F7 differs. What about if we want to move data from a register to another? There is no instruction for moving data between registers, so how do we do it? We can use the fact that the register **`x0`** is always full of zeroes. We can then for example do **`add x8, x10, x0`**, or **`sub x8, x10, x0`**. Below here is shown a table with all the instructions available for RISC-V processors:



Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

What if for example we want to add a fixed real number to a register? We can use the addi instruction: this instruction will sum a real number to a register, but we would have to change encoding type. Why do we have to? Because we don't need a second register to be added, so we can just reserve the 12 MSBs to the number (this way we can sum a number in the range [`0`, `2047`]. This is called **`I-Type`** encoding (because of the presence of the **`immediate`** field):

Immediate												1st register					F3			Destination Register					Opcode						
From 31 to 20												From 19 to 15					From 14 to 12			From 11 to 7					From 6 to 0						
0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1

Another important instruction is the **L**oad **U**pper **I**mmediate (**`lui xDest, imm`**). This instruction is used to load a 20-bits wide immediate into a register, but it will need another encoding format. We need the **`U-Type`**:

Immediate																				Destination Register					Opcode						
From 31 to 12																				From 11 to 7					From 6 to 0						
0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1

In we need to load a word from the memory into a register we use the **L**oad **W**ord instruction (**`lw xDest, address`**). The **`lw`** operation has 20 bits for the memory address, giving the possibility of having at least 2^{20} addresses (around 1MB of memory). We can also use a custom bias to reach further memory addresses up to 2^{12} (for example if the maximum that our register can save is 2048 but we need to access to the address 2052, we can do 2048+4; the operation will be **`lw x16 4(x18)`**). The address has to contain a register, but the bias isn't necessary. The bias can be a hexadecimal number too, but it mustn't be bigger than 12 bits.

For traditional reasons, we start to store data in memory after the address **`0x10010000`**.

`EXERCISE`: Load the words from addresses **`0x10010000`** into register **`x3`** and **`0x10010004`** into register **`x4`**.

```
lui x5, 0x10010
lw x3, 0(x5)
lw x4, 4(x5)
```

First we store part of the address into a register (so the first 5 hex digits, since they represent the first 20 MSBs in binary) with Load Upper Immediate (**`lui`**) and then we can use a bias up to 2^{12} to fix the address.

What about a larger address? Let's try with **`0x67F59C1D`**. We can do:

```
lui x5, 0x67F59
ori x5, x0, 0xC1D
lw x6, 0(x5)
```

This way with the **`ori`** instruction we can compute the whole number and store it into a register.

But what if we want to store a word into a memory address? We use the **S**ave/**S**tore **W**ord (**`sw reg2, bias(reg1)`**) instruction, which uses the **`S-Type`** encoding format:

Immediate [11:5]	2nd register	1st register	F3	Immediate [4:0]	Opcode
From 31 to 25	From 24 to 20	From 19 to 15	From 14 to 12	From 11 to 7	From 6 to 0
0 0 0 0 0 0 0	0 0 1 0 1	0 0 1 0 0	0 0 0	0 0 0 0 1	0 1 1 0 0 1 1

The immediate is used to store the bias / offset, since the address is stored into the 1st register. This instruction stores the word of the 2nd register into the memory address given by the 1st register and the immediate

`EXERCISE`: Save the sum of the previous two words into the memory address **`0x10010008`**.

```
add x6, x3, x4
sw x6, 8(x5)
```

When we don't have to use anymore the CPU we have to terminate the process. In order to terminate the process, we have to make a call to the Operating System with the instruction **`ecall`**, which will shut down the system and the computer. In order to terminate the process we have to store into the register **`x17`** the number **`10`**. So the instruction would be:

```
ori x17, x0, 10
ecall
```

`ecall` doesn't need any parameter, it's all within the instructions.

[6.4] RARS

RARS is a software that emulates a RISC-V processor. When programming in Assembly for a RISC-V processor we have to follow a given basis:

- Data section (load from 0x10010000)
    - Word 17
    - Word 25
- Text (it gets loaded in address 0x00400000)
    - lui x5, 0x10010
    - lw x3, 0(x5)
    - lw x4, 4(x5)
    - add x6, x3, x4
    - sw x6, 8(x5)