

Computer Architecture

IN BA3 - Paolo IENNE

Notes by Ali EL AZDI

Introduction

This document is designed to offer a LaTeX-styled overview of the Computer Architecture course, emphasizing brevity and clarity. Should there be any inaccuracies or areas for improvement, please reach out at ali.elazdi@epfl.ch for corrections. For the latest version, check my GitHub repository.

<https://github.com/elazdi-al/comparch/blob/main/main.pdf>

Contents

Contents	3
1 Part I(a) - ISA Reminder, Assembly Language, Compiler - W 1.1	5
1.1 From High Level Languages to Assembly Language	5
1.1.1 High Level Languages	5
1.1.2 Assembly Language	5
1.2 Processors	6
1.3 Joint or Disjoint Program and Data Memories	7
1.4 The Encoding problem	8
1.4.1 The Stupid Solution	8
1.4.2 RISC-V Encoding (The Solution)	8
1.4.3 Automating this process	9
1.5 ISA (Instruction Set Architecture)	9
2 Part I(b) - ISA, Functions, and Stack - W 1.2	11
2.1 The Contract between HW and SW	11
2.2 Arithmetic and Logic Instructions in RISCV	11
2.2.1 Constants must be encoded on 12 bits	12
2.2.2 Assembler Directives	12
2.2.3 The x0 Register	13
2.3 PseudoInstructions	13
2.3.1 Control flow instructions	13
2.3.2 If-Then-Else	13
2.3.3 Jumps and Branches	14
2.3.4 Comparaisons	14
2.3.5 Do-While	14
2.4 Functions	14
2.4.1 Jump to the Function/Retun control to the calling program	15
2.4.2 Jump Instructions	15
2.4.3 Register Conventions	16
2.4.4 Back to the good (not so good) approach	16
2.4.5 One simple solution (still not good)	16
2.4.6 Acquire storage resources the function needs (still not it)	17
2.4.7 The Stack	17
2.4.8 Spilling Registers to Memory	19
2.4.9 Register across functions	19
2.4.10 Preserving Registers	20
2.5 Passing Arguments in RISC-V	20
2.5.1 Option 1: Using Registers	20
2.5.2 Option 2: Using the Stack	21

2.5.3	The RISC-V Approach	21
2.6	Summary of RISC-V Register Conventions	21
3	Part I(c) - ISA Memory and Addressing Modes - W 2.1	22
3.1	Memory	22
3.1.1	Address and Data	22
3.2	Many Types of Memories	23
3.2.1	Functional Taxonomy of Memories	23
3.2.2	Taxonomy of Random Access Memories	23
3.2.3	Basic Structure	24
3.2.4	Write Operations	24
3.2.5	Read Operations	24
3.2.6	Practical SRAMs	24
3.2.7	DRAMs	25
3.2.8	Ideal Random Access Memory	25
3.2.9	Physical Organisation	25
3.2.10	Realistic ROM Array	26
3.2.11	Static Ram Typical Interface	26
3.3	Typical Asynchronous SRAM Read Cycle	26
3.4	Where is Memory in the Processor?	27
3.4.1	Arithmetic and Logic Instructions	27
3.5	More Addressing Modes? Not in RISC-V!	28
3.5.1	Word Adressed Memory	29
3.5.2	Loading Words (lw) and Instructions	29
3.5.3	Loading Bytes (lb)	29
3.5.4	A Few More Load/Store Instructions	29
3.5.5	Access as it is more suitable	30
3.5.6	Loading Bytes (lb)	31
4	Part I(d) - ISA Arrays and Data Structures - W 2.2	32
4.1	Arrays	32
4.1.1	Different Ways to Store Arrays	32
4.1.2	Adding Positive Elements	33
4.1.3	Pointer to Memory vs Index in Array	34

Chapter 1

Part I(a) - ISA Reminder, Assembly Language, Compiler - W 1.1

hum...welcome back

In the first part of the course, professor introduced (for motivational purposes) how computer architecture, specifically processors, have become essential to our lives, and how the field is growing exponentially. (didn't think it was essential to mention here...)

1.1 From High Level Languages to Assembly Language

1.1.1 High Level Languages

When talking about programming we usually think of programs that look like this...

```
1 int data = 0x00123456;
2 int result = 0;
3 int mask = 1;
4 int count = 0;
5 int temp = 0;
6 int limit = 32;
7 do {
8     temp = data & mask;
9     result = result + temp;
10    data = data >> 1;
11    count = count + 1;
12 } while (count != limit);
```

name	value
data	0x00123456
result	0
mask	1
count	...
temp	
limit	
...	
my_float	3.141529
a_string	Hello world!

1.1.2 Assembly Language

We use this code because it enables us to build a *Finite State Machine*, which isn't feasible with C code. This language provides a more rigid format with a sequence of numbered instructions, an *opcode*, predefined variable names, and the ability to **jump between lines**.

```

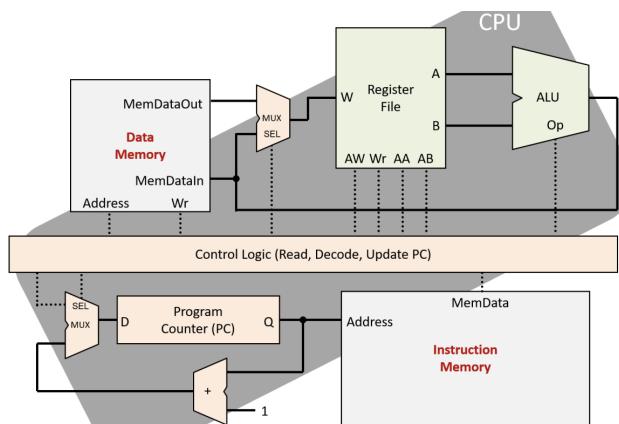
1 li x1, 0x000123456
2 li x2, 0
3 li x3, 1
4 li x4, 0
5 li x5, 0
6 li x6, 32
7 loop: and x5, x1, x3
8     add x2, x2, x5
9     srl x1, x1, 1
10    addi x4, x4, 1
11    bne x4, x6, loop

```

1.2 Processors

Remember, a processor can be decomposed into five components:

- **ALU (Arithmetic and Logic Unit)**: Performs arithmetic and logical operations.
- **Register File**: Stores data temporarily for quick access during processing.
- **Memory**: Holds data and instructions needed by the processor.
- **Control Logic**: Directs the operation of the processor by coordinating the other components.
- **PC (Program Counter)**: Keeps track of the address of the next instruction to be executed.
- **Instruction Memory**: Stores the program instructions that the processor will execute.



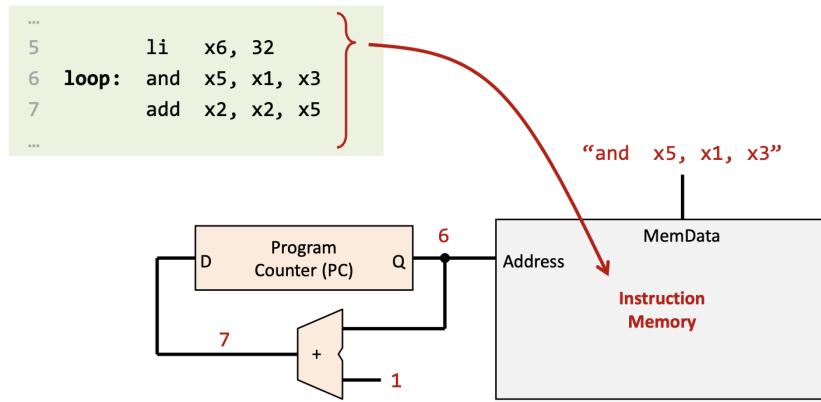
We may distinguish three types of general operations made by the processor:

Encoding

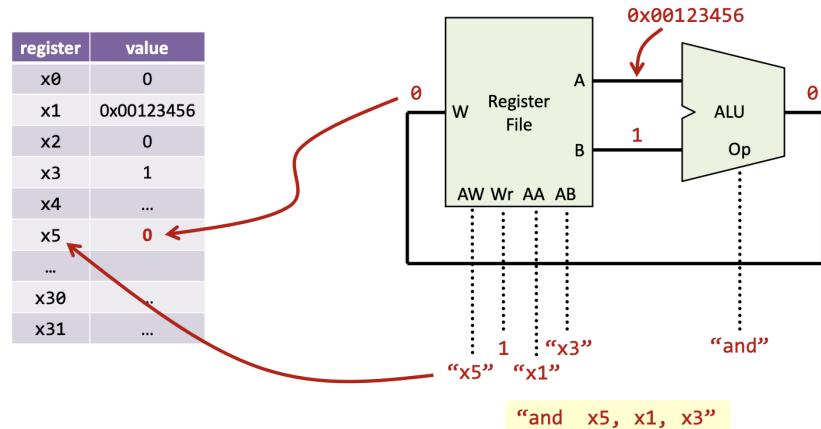
add x1, x1, x1	0 = 0000 0000 0000 0000 0000 0000 0000 0000
add x1, x1, x2	1 = 0000 0000 0000 0000 0000 0000 0000 0001
add x1, x1, x3	2 = 0000 0000 0000 0000 0000 0000 0000 0010
add x1, x1, x4	3 = 0000 0000 0000 0000 0000 0000 0000 0011
add x1, x1, x5	4 = 0000 0000 0000 0000 0000 0000 0000 0100
...	...
and x1, x1, x1	32768 = 0000 0000 0000 0000 1000 0000 0000 0000
and x1, x1, x2	32769 = 0000 0000 0000 0000 1000 0000 0000 0001
and x1, x1, x3	32770 = 0000 0000 0000 0000 1000 0000 0000 0010
and x1, x1, x4	32771 = 0000 0000 0000 0000 1000 0000 0000 0011
and x1, x1, x5	32772 = 0000 0000 0000 0000 1000 0000 0000 0100
...	...

of opcodes x # destinations x # source 1 x # source 1 ≤ 2³² combinations

Fetching



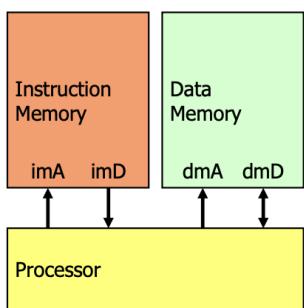
Executing



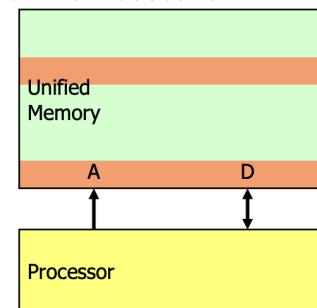
1.3 Joint or Disjoint Program and Data Memories

There are two main types of architectures one called the *Harvard Architecture* (Where the data and the memory are separate) and one called *Unified Architecture* (where data is shared with the program memory)

Harvard Architecture



Unified Architecture



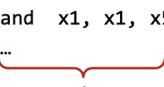
1.4 The Encoding problem

We may ask ourselves how we encode assembly written instructions into actual 0s and 1s.

1.4.1 The Stupid Solution

Now, the professor throws out the "stupid idea" (his words) of just counting all possible instructions, assigning a number to each one, and writing the numbers in binary. The problem with such a method is that the number of instructions could grow exponentially, requiring an unmanageable number of bits to represent each one, leading to inefficiency.

add x1, x1, x1
 add x1, x1, x2
 add x1, x1, x3
 add x1, x1, x4
 add x1, x1, x5
 ...
 and x1, x1, x1
 and x1, x1, x2
 and x1, x1, x3
 and x1, x1, x4
 and x1, x1, x5
 ...

0 = 0000 0000 0000 0000 0000 0000 0000 0000
 1 = 0000 0000 0000 0000 0000 0000 0000 0001
 2 = 0000 0000 0000 0000 0000 0000 0000 0010
 3 = 0000 0000 0000 0000 0000 0000 0000 0011
 4 = 0000 0000 0000 0000 0000 0000 0000 0100
 ...

32768 = 0000 0000 0000 0000 1000 0000 0000 0000
 32769 = 0000 0000 0000 0000 1000 0000 0000 0001
 32770 = 0000 0000 0000 0000 1000 0000 0000 0010
 32771 = 0000 0000 0000 0000 1000 0000 0000 0011
 32772 = 0000 0000 0000 0000 1000 0000 0000 0100
 ...

of opcodes x # destinations x # source 1 x # source 1 <= 2³² combinations

"stupid solution"

1.4.2 RISC-V Encoding (The Solution)

Instead, the chosen solution is to use an instruction set encoding where instructions are grouped into classes, each with a fixed format optimizing both memory usage and processing speed by limiting the number of bits required to represent instructions.

Instruction	Pseudocode	Type	funct7	funct3	opcode
Shift					
sll rd,rs1,rs2	$rd \leftarrow rs1 \ll rs2$	R	0x00	0x1	0x33
slli rd,rs1,imm	$rd \leftarrow rs1 \ll imm$	I	0x00	0x1	0x13
srl rd,rs1,rs2	$rd \leftarrow rs1 \gg_u rs2$	R	0x00	0x5	0x33
srai rd,rs1,imm	$rd \leftarrow rs1 \gg_u imm$	I	0x00	0x5	0x13
sra rd,rs1,rs2	$rd \leftarrow rs1 \gg_s rs2$	R	0x20	0x5	0x33
sra rd,rs1,imm	$rd \leftarrow rs1 \gg_s imm$	I	0x20	0x5	0x13
Arithmetic					
add rd,rs1,rs2	$rd \leftarrow rs1 + rs2$	R	0x00	0x0	0x33
addi rd,rs1,imm	$rd \leftarrow rs1 + sext(immm)$	I	0x00	0x0	0x13
sub rd,rs1,rs2	$rd \leftarrow rs1 - rs2$	R	0x20	0x0	0x33
lui rd,imm	$rd \leftarrow imm$				
auipc rd,imm	$rd \leftarrow pc + imm$				
Logical					
xor rd,rs1,rs2	$rd \leftarrow rs1$	R	funct7	rs2	0
xori rd,rs1,imm	$rd \leftarrow rs1$	I	imm[11:0]	rs1	funct3
or rd,rs1,rs2	$rd \leftarrow rs1$	I	funct7	imm[4:0]	rs1
ori rd,rs1,imm	$rd \leftarrow rs1$	S	imm[11:5]	rs2	rs1
and rd,rs1,rs2	$rd \leftarrow rs1$	B	imm[12-10:5]	rs2	funct3
andi rd,rs1,imm	$rd \leftarrow rs1$	U	imm[31:12]		imm[4:1-11]
		J	imm[20-10:1-11-19:12]		rd
					opcode
					Register-Register
					Register-Immediate
					Register-Immediate Shift
					Store
					Branch
					Upper Immediate
					Jump

1.4.3 Automating this process

Now to automate the processes of decoding assembler code into machine code we use an **Assembler**, and to automate the process of decoding a higher level language to assembler we use a **Compiler**.

Assembler

The program that does this is called an assembler. It takes the assembly code and converts it into machine code.

```

0      li    x1, 0x00123456
1      li    x2, 0
2      li    x3, 1
3      li    x4, 0
4      li    x5, 0
5      li    x6, 32
6  loop: and   x5, x1, x3
7      add   x2, x2, x5
8      srl   x1, x1, 1
9      addi  x4, x4, 1
10     bne   x4, x6, loop

```

```

0101 0101 0101 0000 0100 0111 1010 1110
0001 0100 1001 1101 0011 0000 1100 1001
1101 1100 1101 0110 0000 1101 0001 0111
0010 0011 1101 0110 0010 0000 0001 1001
1100 1010 1011 1010 0111 0100 0000 0110
1111 0010 1001 0011 1001 1110 1001 1101
0011 0000 0010 0111 1111 0000 0100 0011
0111 1001 0101 1101 1000 1000 0111 1011
1100 1010 1011 0000 0100 0100 0110 0101
0111 1001 0010 0110 0000 0011 0001 0010
0101 1100 1000 0101 0000

```

A fairly trivial job

Assembly

Compiler

A compiler is a program that translates high-level source code written in languages like C or Java into machine code or an intermediate representation.

```

int data  = 0x00123456;
int result = 0;
int mask   = 1;
int count  = 0;
int temp   = 0;
int limit  = 32;
do {
    temp   = data & mask;
    result = result + temp;
    data   = data >> 1;
    count  = count + 1;
} while (count != lim

```

```

0      li    x1, 0x00123456
1      li    x2, 0
2      li    x3, 1
3      li    x4, 0
4      li    x5, 0
5      li    x6, 32
6  loop: and   x5, x1, x3
7      add   x2, x2, x5
8      srl   x1, x1, 1
9      addi  x4, x4, 1
bne   x4, x6, loop

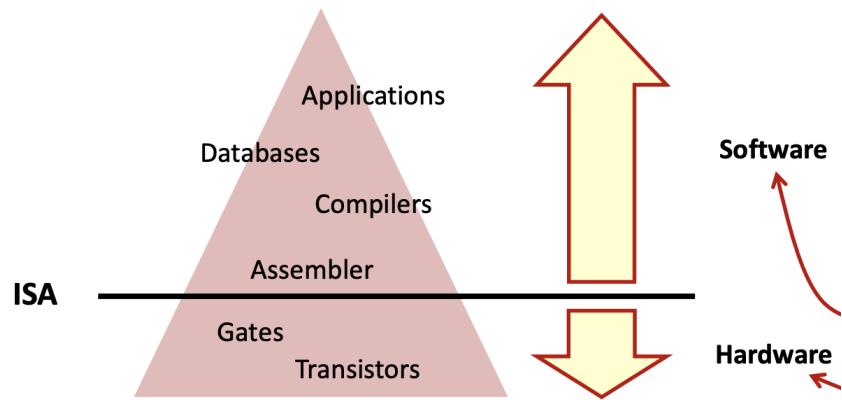
```

A pretty hard job!...

Compilation

1.5 ISA (Instruction Set Architecture)

The ISA is the interface between the hardware and the software. It defines the instructions that a processor can execute, as well as the format of those instructions.

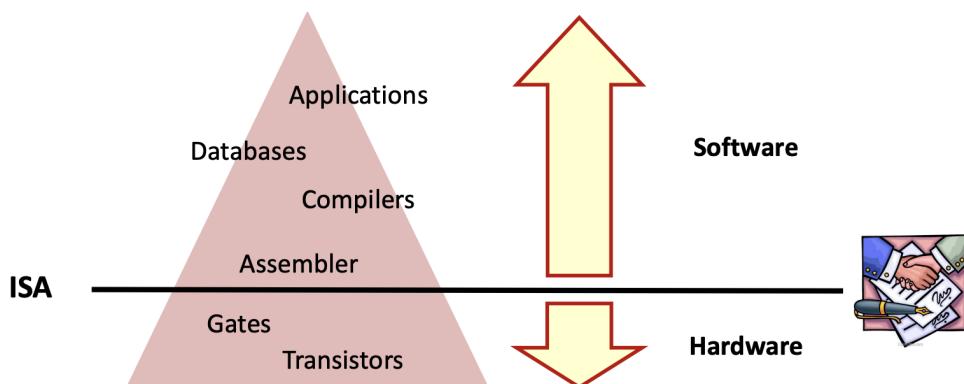


Chapter 2

Part I(b) - ISA, Functions, and Stack - W 1.2

2.1 The Contract between HW and SW

The Contract between hardware and software is **ISA** (Instruction Set Architecture), it defines the rules the hardware and software follow to work together and communicate correctly;



2.2 Arithmetic and Logic Instructions in RISC-V

Below some examples of RISC-V instructions:

Two Operands Instructions

```
1 sll  x5, x5, x9  
2 add  x6, x5, x7  
3 xor  x6, x6, x8  
4 slt  x8, x6, x7
```

Shift $x5$ left by $x9$ positions $\rightarrow x5$
Add $x5$ and $x7 \rightarrow x6$
Logic XOR bitwise $x6$ and $x8 \rightarrow x6$
Set $x8$ to 1 if $x6$ is lower than $x7$, otherwise to 0

Arithmetic Instructions

```
1 slli x5, x5, 3  
2 addi x6, x5, 72  
3 xori x6, x6, -1  
4 slti x8, x6, 321
```

Shift $x5$ left of 3 positions $\rightarrow x5$
Add 72 to $x5 \rightarrow x6$
Logic XOR bitwise $x6$ and $0xFFFFFFFF \rightarrow x6$
Set $x8$ to 1 if $x6$ is lower than 321, to 0 otherwise

Here, you may ask yourself, why are all immediates (constants) written on a maximum of 12bits?

2.2.1 Constants must be encoded on 12 bits

As you may see here, all instructions encode immediates on 12 bits.

	31	25 24	20 19 15	14	12 11	7 6	0	
R	funct7	rs2	rs1	funct3	rd	opcode		Register-Register
I	imm[11:0]		rs1	funct3	rd	opcode		Register-Immediate
I	funct7	imm[4:0]	rs1	funct3	rd	opcode		Register-Immediate Shift
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		Store
B	imm[12–10:5]	rs2	rs1	funct3	imm[4:1–11]	opcode		Branch
U	imm[31:12]				rd	opcode		Upper Immediate
J	imm[20–10:1–11–19:12]				rd	opcode		Jump

2.2.2 Assembler Directives

Assembler directives help write cleaner and more readable code. The code snippets on the left and right below are equivalent.

<pre> lui x5, 0x12345 addiu x5, x5, 0x678 xor x6, x6, x5 </pre>		<pre> .equ something, 0x12345678 lui x5, %hi(something) addiu x5, x5, %lo(something) xor x6, x6, x5 </pre>
--	--	---

The left-hand side code snippet shows an assembly sequence where a 32-bit constant value (0x12345678) is loaded into a register (x5). Since immediate values are 16-bit limited, this requires splitting the 32-bit value into two instructions:

- The first instruction, `lui`, loads the upper 16 bits (0x12345) into the register `x5`.
- The second instruction, `addiu`, adds the lower 16 bits (0x678) to `x5`, completing the full 32-bit value in the register.

This approach, while functional, can become cumbersome when dealing with multiple constants, making the code less readable and harder to maintain.

The right-hand side shows the same functionality but makes use of assembler directives, specifically the `.equ` directive to define a label (`something`) for the constant 0x12345678. Using the `%hi()` and `%lo()` pseudo-instructions, the assembler automatically splits the constant into its upper and lower parts:

- The `%hi(something)` loads the upper 16 bits into `x5`.
- The `%lo(something)` adds the lower 16 bits to `x5`.

This method enhances code clarity and maintainability, especially when working with multiple constants, by using human-readable labels instead of raw numeric values. The assembler handles the details of splitting the 32-bit constant into its upper and lower parts.

Directive	Effect
<code>.text</code>	Store subsequent instructions at next available address in <i>text</i> segment
<code>.data</code>	Store subsequent items at next available address in <i>data</i> segment
<code>.asciiz</code>	Store string followed by null-terminator in <code>.data</code> segment
<code>.byte</code>	Store listed values as 8-bit bytes
<code>.word</code>	Store listed values as 32-bit words
<code>.equ</code>	Define constants

2.2.3 The x0 Register

The `x0` register is hardwired to 0 and cannot be changed. Any attempt to write into `x0` will have no effect.

Why is this useful?

One common application is in introducing wait delays during program execution. By leveraging the fixed nature of `x0`, it simplifies certain instructions that require an immediate zero value.

2.3 PseudoInstructions

PseudoInstructions simplify commands involving the `x0` register by creating easier-to-use alternatives.

Pseudoinstruction	Base Instruction(s)	Meaning
<code>nop</code>	<code>addi x0, x0, 0</code>	No operation
<code>li rd, immediate</code>	Myriad sequences	Load immediate
<code>mv rd, rs</code>	Myriad sequences	Copy register
<code>not rd, rs</code>	<code>xori rd, rs, -1</code>	One's complement
<code>neg rd, rs</code>	<code>sub rd, x0, rs</code>	Two's complement
<code>seqz rd, rs</code>	<code>sltiu rd, rs, 1</code>	Set if = zero
<code>snez rd, rs</code>	<code>sltu rd, x0, rs</code>	Set if \neq zero
<code>sltz rd, rs</code>	<code>slt rd, rs, x0</code>	Set if \downarrow zero
<code>sgtz rd, rs</code>	<code>slt rd, x0, rs</code>	Set if \downarrow zero

The term *myriad sequences* refers to a series of instructions that together achieve the functionality of a single pseudoinstruction, such as using `lui` and `addi` to implement `li rd, immediate`.

According to the professor `li` should be called `mvi` (as move immediate).

2.3.1 Control flow instructions

Control flow instructions are used to change the order of execution of instructions are a kind of pseudo-instructions.

```

1 li x1, 0x00123456
2 li x2, 0
3 li x3, 1
4 li x4, 0
5 li x5, 0
6 li x6, 32
7 loop: and x5, x1, x3
8     add x2, x2, x5
9     srli x1, x1, 1
10    addi x4, x4, 1
11    bne x4, x6, loop

```

2.3.2 If-Then-Else

```

1 if (x5 == 72) {
2     x6 = x6 + 1;
3 } else {
4     x6 = x6 - 1;
5 }

```

```

1 .text
2     li x7, 72
3     beq x5, x7, then_clause
4     else_clause:
5         addi x6, x6, -1
6         j end_if
7     then_clause:
8         addi x6, x6, 1
9     end_if:

```

As seen here, `beqi` does not exist in RISCV, instead we use `beq` and `li` to achieve the same result.

2.3.3 Jumps and Branches

A common but not universal distinction exists between *jumps* and *branches*. In RISC-V (inherited from MIPS and used by SPARC, Alpha, etc.), jumps refer to unconditional control transfer instructions, while branches refer to conditional control transfer instructions. However, not all architectures follow this convention. For instance, in x86, all control transfer instructions are considered jumps, such as `JMP`, `JZ`, `JC`, and `JNO`.

2.3.4 Comparaisons

The processor implements only `<` and `>`, and the assembler “creates” \leq and \geq .

Pseudoinstruction	Base Instruction(s)	Meaning
<code>beqz rs, offset</code>	<code>beq rs, x0, offset</code>	Branch if = zero
<code>bnez rs, offset</code>	<code>bne rs, x0, offset</code>	Branch if \neq zero
<code>blez rs, offset</code>	<code>bge x0, rs, offset</code>	Branch if \leq zero
<code>bgez rs, offset</code>	<code>bge rs, x0, offset</code>	Branch if \geq zero
<code>bltz rs, offset</code>	<code>blt rs, x0, offset</code>	Branch if $<$ zero
<code>bgtz rs, offset</code>	<code>blt x0, rs, offset</code>	Branch if $>$ zero
<code>bgt rs, rt, offset</code>	<code>blt rt, rs, offset</code>	Branch if $>$
<code>ble rs, rt, offset</code>	<code>bge rt, rs, offset</code>	Branch if \leq
<code>bgtu rs, rt, offset</code>	<code>bltu rt, rs, offset</code>	Branch if $>$, unsigned
<code>bleu rs, rt, offset</code>	<code>bgeu rt, rs, offset</code>	Branch if \leq , unsigned

2.3.5 Do-While

Do-while loops look like this (we obviously use control flow instructions here).

```

1 do {
2     x5 = x5 >> 1;
3     x6 = x6 + 1;
4 } while (x5 != 0);

```

```

1 .text
2 loop:
3     srl x5, x5, 1
4     addi x6, x6, 1
5     bnez x5, loop

```

2.4 Functions

In higher-level programming languages, functions (routines, subroutines, procedures, methods, etc.) are used to encapsulate code and make it reusable.

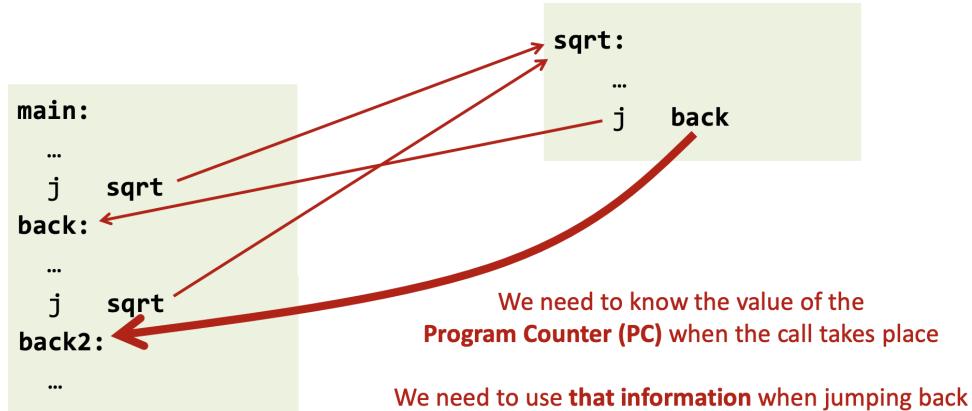
Calling a function involves these steps:

1. Place arguments where the called function can access them.
2. Jump to the function.
3. Acquire storage resources the function needs.
4. Perform the desired task of the function.
5. Communicate the result value back to the calling program.
6. Release any local storage resources.
7. Return control to the calling program.

2.4.1 Jump to the Function/Retun control to the calling program

The too simple not working approach

A simple (not working) approach for creating functions would be to do this:



With this approach the function doesn't know where to return to after being called (back2 or back). For the next part, remember, the Program Counter is distinct from general-purpose registers. It is dedicated to managing the flow of instruction execution, while general registers are used for data manipulation.

The Good Approach

The right approach involves using the Jump and Link instruction *jal*, here loading PC + 4 (remember 4 bytes per Instruction) into *x1* as a way to come back from the function.

```

1 main:
2 ...
3     jal x1, sqrt
4 ...
5 ...
6     jal x1, sqrt

```

```

1 sqrt:
2 ...
3 ...
4     jr x1

```

Both times *x1* was used to store the return address, and there is a reason for that (Register Conventions Sections).

2.4.2 Jump Instructions

There are only two core real jump instructions in RISCV, *jal* (jump and link) and *jalr* (jump and link register), the rest are pseudo instructions using them.

Pseudoinstr.	Base Instruction(s)	Meaning
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, 0(rs)	Jump register
jalr rs	jalr x1, 0(rs)	Jump and link register
ret	jalr x0, 0(x1)	Return from subroutine

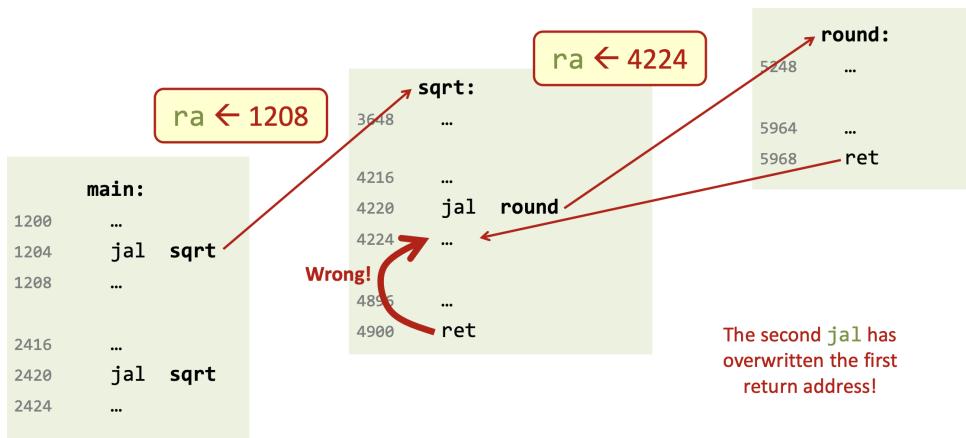
2.4.3 Register Conventions

Register conventions are rules that dictate how registers are used in a program, here are the ones we've seen for now

Register	Mnemonic	Description
x0	zero	Hard-wired zero
x1	ra	Return Address

2.4.4 Back to the good (not so good) approach

There's still a problem with the previous approach, say for example you want to call a function from another function.



Here the allocated space for the return address is overwritten by the second function call, and the first function can't return to the right place.

2.4.5 One simple solution (still not good)

One solution would be to say that a range of registers are used for certain functions and that they can't be used by other functions.



The problem here is that it's still not very scalable.

2.4.6 Acquire storage resources the function needs (still not it)

One simple solution to our problem would be to allocate memory for the function at in the data section of the program.

```

1 .data
2 sqrt_save_ra: .word 0
3 sqrt_save_x5: .word 0

```

```

1 .text
2 sqrt:
3 ...
4 add x5, x7, x8
5 sw ra, sqrt_save_ra
6 sw x5, sqrt_save_x5
7 jal round
8 lw ra, sqrt_save_ra
9 lw x5, sqrt_save_x5
10 sub x6, x6, x5
11 ...
12 ret

```

Problem: Recursive Functions

The problem here is that the return address is overwritten by the recursive call.

```

1 .data
2     find_child_save_ra: .word 0
3 .text
4     find_child:
5     ...
6     sw ra, find_child_save_ra
7     jal find_child
8     lw ra, find_child_save_ra
9     ...
10    ret

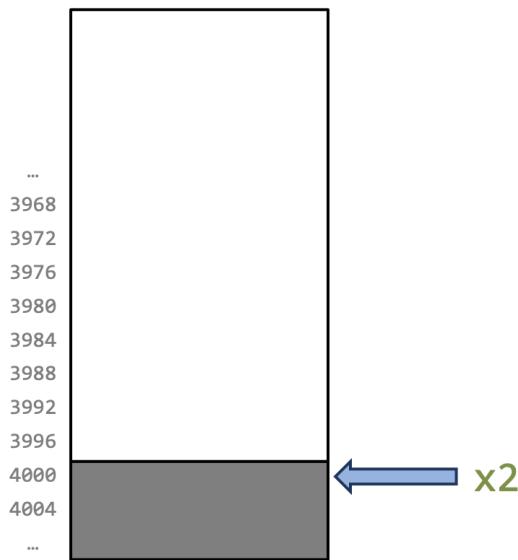
```

2.4.7 The Stack

The Solution to our Problem is this, the Stack.

The Stack is a region of memory that grows and shrinks as needed.

We may use a register (e.g x2) to point to the first used word after the end of the used region.



Dynamic Memory Allocation

The Stack, contrary to the Data Section, is dynamic and can be used to allocate memory when needed. This means that during program execution, variables or temporary data can be stored in the stack, which grows or shrinks depending on the operations performed.

The **stack pointer**, typically register x2, is used to manage the allocation and deallocation of memory.

In this instruction, for example, we allocate 12 bytes in the stack. We achieve this by decrementing the stack pointer (x2) by 12. This ensures that the new memory space is available for temporary storage.

```
1 addi x2, x2, -12
```

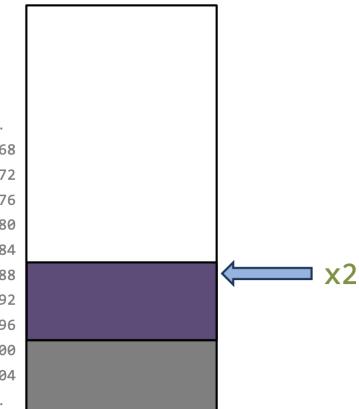


Retrieving Data from the Stack

Once memory has been allocated on the stack, we can store or retrieve data from it. In this case, we are retrieving data that was previously saved in the stack. The lw (load word) instruction is used to load the values stored at different offsets in the stack.

In this case, we retrieve three different values from the stack using the lw instruction, which loads a 4-byte value into the specified registers (ra, x5, and x6). The offsets (0, 4, and 8) refer to different positions in the 12 bytes we allocated earlier.

```
1 lw ra, 0(x2)
2 lw x5, 4(x2)
3 lw x6, 8(x2)
```



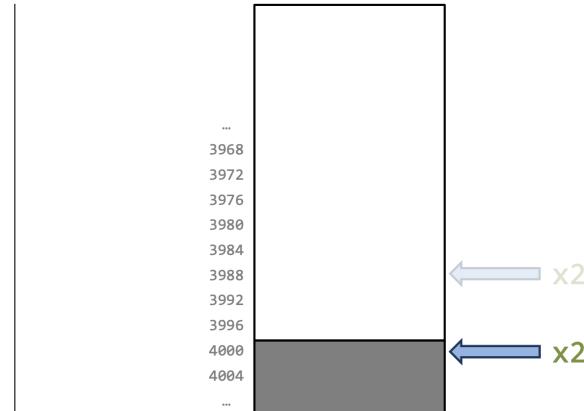
Memory Deallocation

After the data has been used or is no longer needed, it is good practice to deallocate the memory to ensure proper management of the stack. We deallocate memory by adjusting the stack pointer ($x2$) back to its original position.

In this instruction, we restore the stack to its previous state by adding 12 back to the stack pointer ($x2$).

This effectively "frees" the 12 bytes of memory we had allocated earlier.

1 addi x2, x2, 12



The Stack Pointer

The Stack Pointer is a register that points to the top of the stack, by convention it corresponds to the $x2$ register

Register	ABI Name	Description	Preserved across call?
$x2$	sp	Stack pointer	Yes

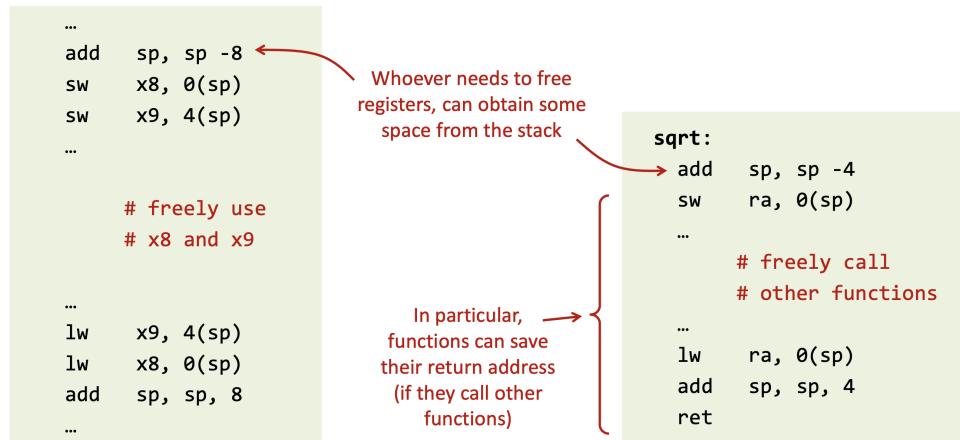
Other architectures have special instructions to place stuff on the stack (push) and to retrieve it (pop)

PUSH AX

1 add sp, sp, -4
2 sw x5, 0(sp)

2.4.8 Spilling Registers to Memory

Spilling registers to memory involves saving register values to the stack when more registers are needed or to prevent overwriting important data, allowing the registers to be reused. This technique is also used in function calls to save the return address, ensuring the program can correctly return control after the function finishes.



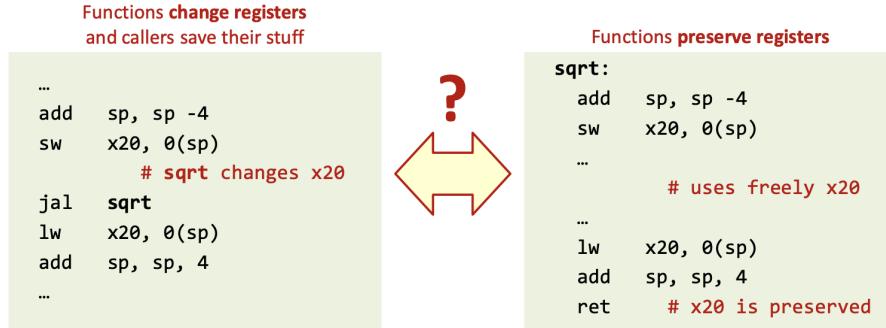
2.4.9 Register across functions

In assembly programming, handling registers across functions can be managed in two main ways: either functions **change registers** and expect the caller to save their values, or functions **preserve registers** and ensure that the register values remain the same across function calls.

- On the left, the function `sqrt` changes the value of register `x20`, requiring the caller to save and restore its value.
- On the right, the function `sqrt` preserves the value of `x20`, ensuring that the caller does not need to manage the saving and restoring.

This distinction is important, but it does not cause issues as long as there is agreement on how registers are handled.

In case it's still not clear, we're looking at the `sw` instruction



2.4.10 Preserving Registers

In RISC-V, register preservation is managed through a combination of callee-saved and caller-saved registers. Callee-saved registers (such as `s0`, `s1`, and `s2-11`) are preserved by the called function, ensuring that their values remain unchanged after the function call.

Caller-saved registers (such as `t0`, `t1-2`, and `t3-6`) are temporary and do not need to be preserved by the called function, meaning the caller must save them if their values are important.

Register	ABI Name	Description	Preserved across call?
<code>x0</code>	zero	Hard-wired zero	—
<code>x1</code>	ra	Return address	No
<code>x2</code>	sp	Stack pointer	Yes
<code>x5</code>	<code>t0</code>	Temporary/alternate link register	No
<code>x6-7</code>	<code>t1-2</code>	Temporaries	No
<code>x8</code>	<code>s0/fp</code>	Saved register/frame pointer	Yes
<code>x9</code>	<code>s1</code>	Saved register	Yes
<code>x18-27</code>	<code>s2-11</code>	Saved registers	Yes
<code>x28-31</code>	<code>t3-6</code>	Temporaries	No

2.5 Passing Arguments in RISC-V

In RISC-V, there are two main ways to pass arguments to functions:

2.5.1 Option 1: Using Registers

- Specific registers are used to pass arguments and return results.
- This can be done in a straightforward way, where each function uses different registers (e.g., passing an argument in `x5` and returning the result in `x6`).
- A more structured approach is to follow a convention where arguments are passed in registers `x10` to `x17`, with results returned in `x10`.
- The limitation: if there are more arguments than available registers (e.g., more than 8 arguments), this approach is insufficient.

2.5.2 Option 2: Using the Stack

- When registers are not enough, extra arguments are placed on the stack.
- The stack offers a universal solution because it has no practical limit on size.
- However, using the stack is more complex and requires additional work compared to using registers.

2.5.3 The RISC-V Approach

- RISC-V uses a combination of both methods.
- Registers x10 to x17 are used to pass arguments, with x10 and x11 also handling return values.
- If more arguments are needed beyond what these registers can handle, they are passed via the stack.

Register	ABI Name	Description	Preserved across call?
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No

Register reserved for arguments and return values in RISC-V.

2.6 Summary of RISC-V Register Conventions



Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/ alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/ frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No

Chapter 3

Part I(c) - ISA Memory and Addressing Modes - W 2.1

3.1 Memory

Memory is a really important component of a computing system, we store our programs in it, we store our data in it, and it's through memory that we receive and send data.

Though memory is very useful it has three main drawbacks:

- It's **slow** → Caches
- It's **finite** → Virtual Memory
- It can make an ISA **too complex** → Pipelining

no worries we'll cover each one of these in this chapter.

3.1.1 Address and Data

Data in Memory can be accessed by an address, meaning it's a Random Access (it can access a memory value without going through the preceding ones).

Professor Remark: "There's not anything random about this memory, we'd better call it and arbitrary access memory. (!not and official name)"

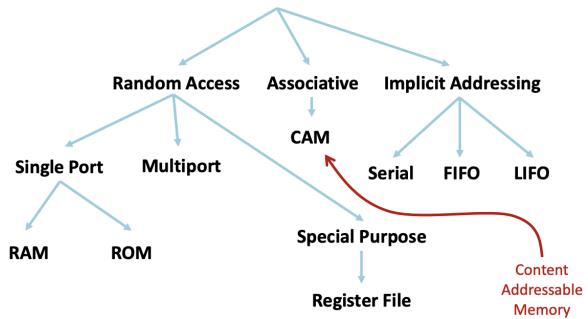
Address	Value
0	12
1	6
2	4
3	1
4	0
5	3
6	1
7	13
8	15
9	9
10	3
11	5
12	0
13	0
14	0
15	0

Write	Read
Memory[5] = 3	Memory[5]?

3.2 Many Types of Memories

We may distinguish between different types of memories based on their **technology**, such as SRAM, DRAM, EPROM, and Flash, and their **capabilities**, including **speed**, **capacity**, **density**, **writability** (whether they are writable, permanent, or reprogrammable), as well as their **size**, **volatility**, and **cost**.

3.2.1 Functional Taxonomy of Memories

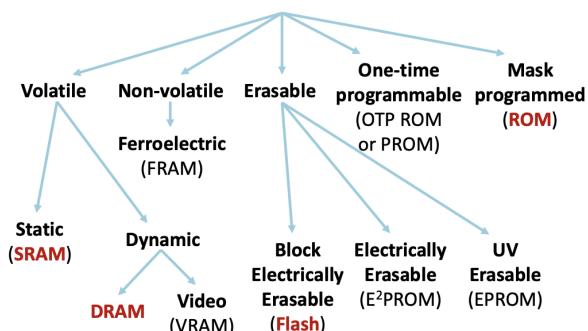


Multiport memory allows simultaneous access by multiple processors, while **single-port** memory supports only one at a time.

Non-Random Access memories

- **Associative** memories enable fast data retrieval by content rather than address, making it useful for cache memory, pattern recognition, and efficient lookups in large datasets.
 - In **Implicit addressing** the address of the data to be operated on is inferred directly by the operation code (opcode), without explicitly specifying the address in the instruction.

3.2.2 Taxonomy of Random Access Memories

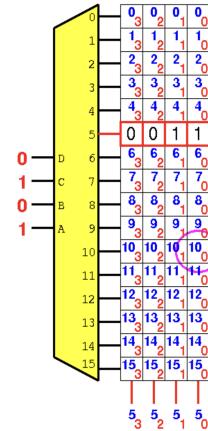


3.2.3 Basic Structure

Remember, a Data Flip Flop, stores a 1 bit value by updating the output value to the input value at the rising edge of the clock signal.

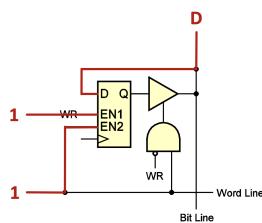
Address	Value
0	12
1	6
2	4
3	1
4	0
5	3
6	1
7	13
8	15
9	9
10	3
11	5
12	0
13	0
14	0
15	0

16 x 4 Memory Cells (Special DFFs (Data Flip-Flops))



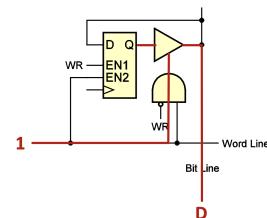
3.2.4 Write Operations

The *D* is connected to the Data outside of the system and at the rising edge it updates the value of the DFF. The AND gate ensures that the write signal is high when the clock signal is high.



3.2.5 Read Operations

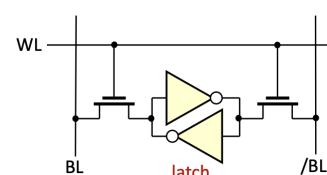
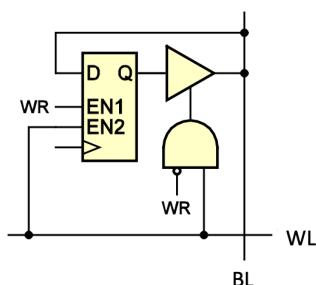
D is still connected to the Data, remember the tri-state driver is active when it's enable signal is active (so when the wr is off and the operation signal is sent.).



3.2.6 Practical SRAMs

DISCLAIMER !!: Combinational loops are prohibited as they can lead to unstable behavior, unpredictable timing, simulation and synthesis issues, excessive power consumption, and lack of a defined reset state, making them unsuitable for reliable digital circuit design.

While the type of memory we've just seen is small, and very fast, SRAM memories uses 6 transistors per cell (less than the previous design). We've also seen (in Taxonomy) that SRAM is static meaning it doesn't require periodic refresh.

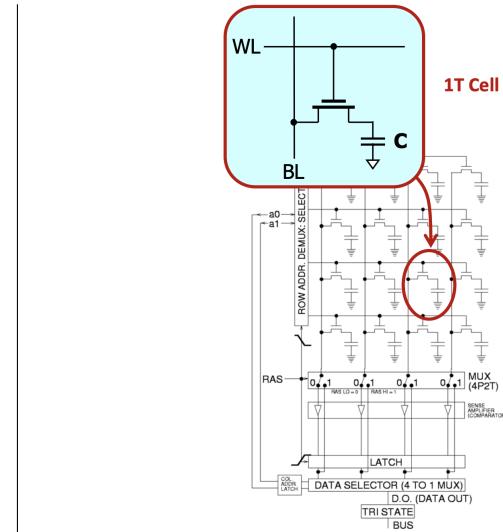


3.2.7 DRAMs

Dynamic RAMS(DRAMs) are the densest and cheapest type of RAM memory, it stores information as charge in small capacitors. This makes the DRAM need periodic refresh otherwise the charge might leak off (60ms) the capacitor due to parasitic resistances and the information lost

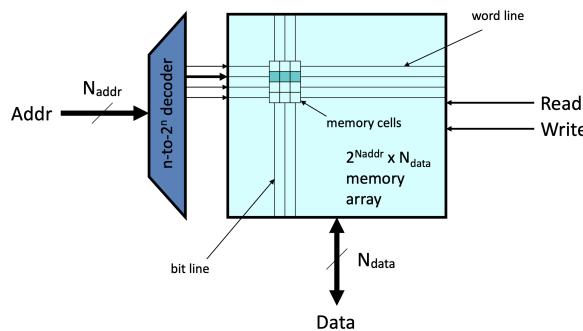
Refresh means, we come back before the end of a charge (60ms) and we rewrite the value, if there is still some charge, we add charge, if there's no charge and we keep as is.

Personal Remark: Dynamic = Bad, data disappears and needs refresh

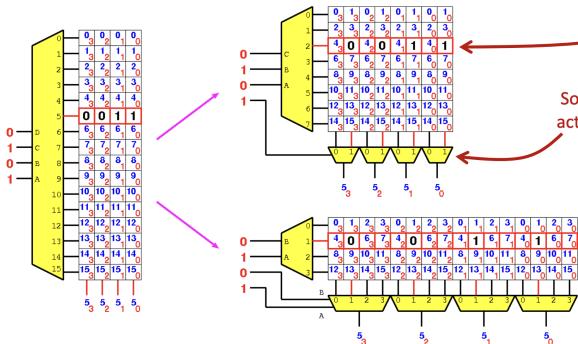


3.2.8 Ideal Random Access Memory

A memory array uses an n -to- 2^n decoder to select a word line based on the input address, enabling data to be read or written through the bit lines.



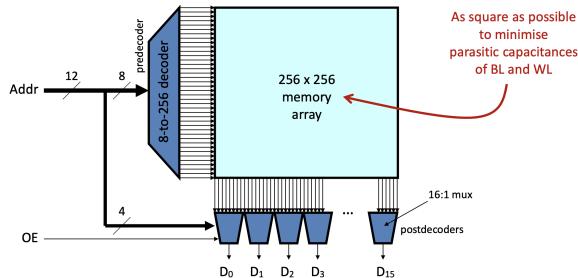
3.2.9 Physical Organisation



Out of all physical organizations, the squared one is the best one as it has the best performance. This layout facilitates faster access times and simplified wiring, resulting in improved computational efficiency and system scalability.

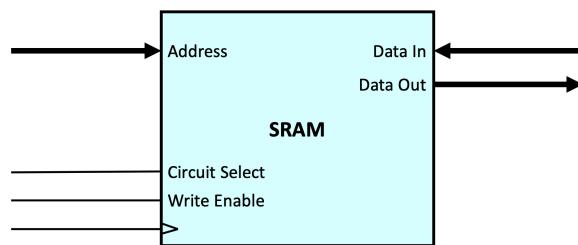
3.2.10 Realistic ROM Array

ROMs are Read-Only Memories, they are used to store the program of the computer, they are non-volatile and can't be written to.



3.2.11 Static Ram Typical Interface

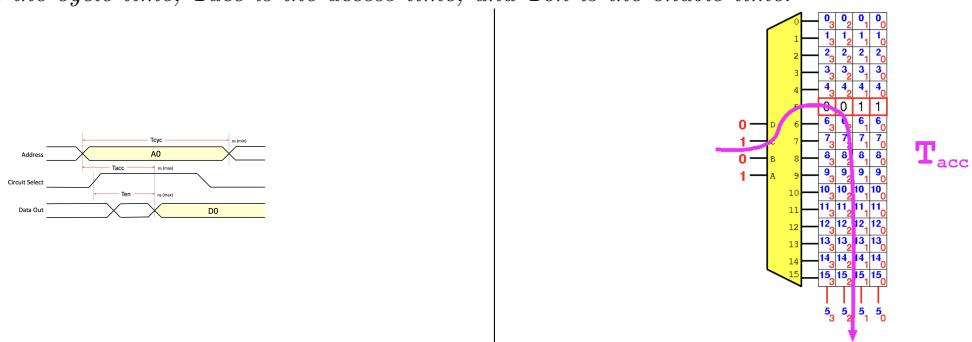
This is a typical interface of a SRAM, it has a 16-bit data input/output, a 16-bit address input, a write enable signal, and a circuit select signal.



3.3 Typical Asynchronous SRAM Read Cycle

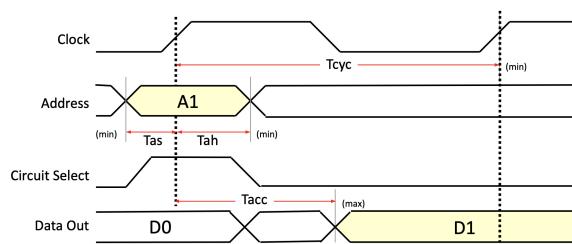
The read cycle of an asynchronous SRAM is initiated by the address input, which is decoded to select the word line, enabling the data to be read from the memory array and output to the data bus.

Here, Tcyc is the cycle time, Tacc is the access time, and Ten is the enable time.



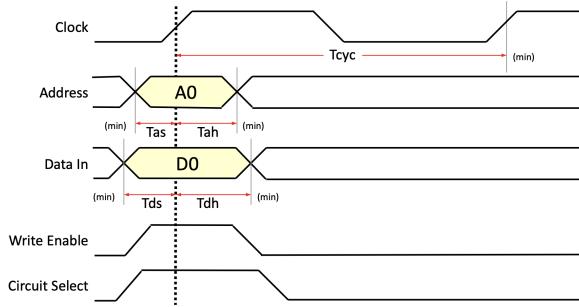
Read Cycle

Latency defined as the number of cycles between the address asserted and data available



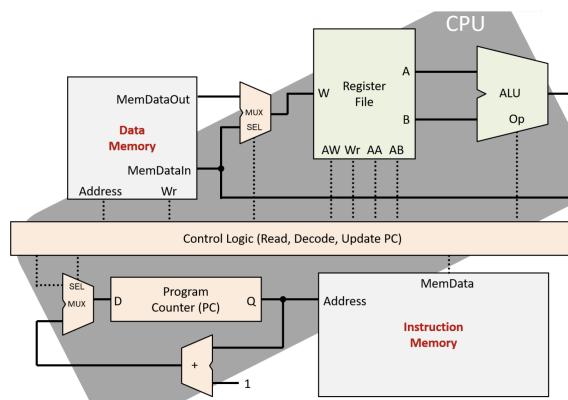
Write Cycle

Writes on the edge of the clock signal, as a DFF



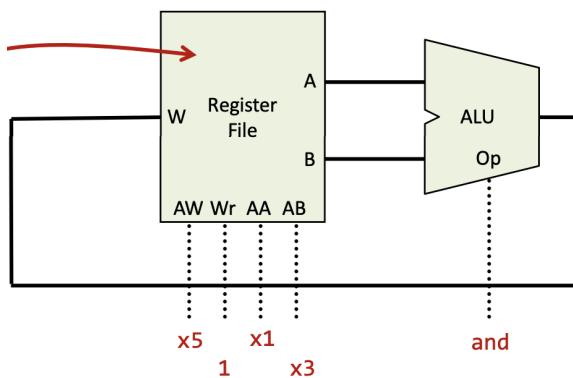
3.4 Where is Memory in the Processor?

In the processor we have memory in the Data memory component and in the Instruction memory component.

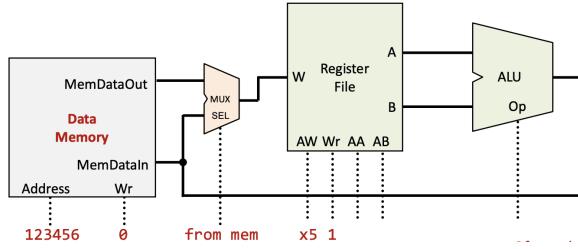


3.4.1 Arithmetic and Logic Instructions

The register file can only contain a limited number of registers making it difficult to handle more complex computations and managing data input/output efficiently.

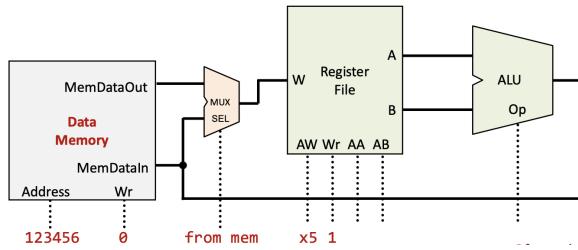


Load Instructions



Load and Store: The RISC-V Way

This instruction would never work for example because the address is too big to be sent as an immediate value : lw x5, (x7)



A Load/Store Architecture

A feature of RISC-V is that it's a Load/Store architecture, meaning that the only way to access memory is through load and store instructions. Also, instructions reading and writing in memory do exactly that and nothing else, contrary to more complex instruction set architectures (CISC), where instructions may combine memory access with other operations like arithmetic or logic. This simplicity in RISC-V's instruction set helps with streamlining the pipeline and improving performance efficiency.

Load		I	0x2	0x03
lw	rd, imm(rs1)	rd	$\leftarrow \text{mem}[\text{rs1} + \text{sext}(\text{imm})]$	
Store		S	0x2	0x23
sw	rs2, imm(rs1)	$\text{mem}[\text{rs1} + \text{sext}(\text{imm})] \leftarrow \text{rs2}$		

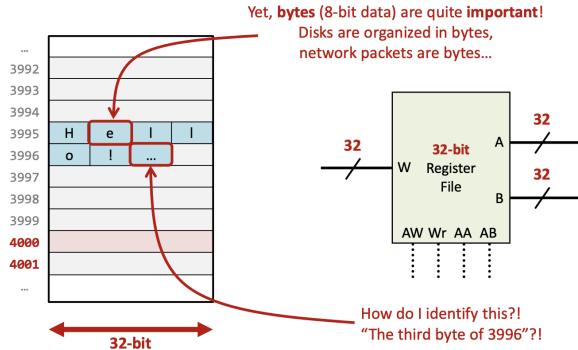
3.5 More Addressing Modes? Not in RISC-V!

Addressing Mode	Instruction	Description
Register	add x0, x1, x2	Adds the value of x1 and x2, stores the result in x0.
Immediate	add x0, x1, 123	Adds the value of x1 and the immediate constant 123, stores the result in x0.
Direct or Absolute	add x0, x1, (1234)	Adds the value of x1 and the value at memory address 1234, stores the result in x0.
Register Indirect	add x0, x1, (x2)	Adds the value of x1 and the value in memory at the address held in x2, stores in x0.
Displacement or Relative	add x0, x1, 123(x2)	Adds the value of x1 and the value in memory at x2 plus the displacement 123, stores in x0.
Base or Indexed	add x0, x1, i5(x2)	Adds the value of x1 and the value in memory at x2 plus index i5, stores in x0.
Auto-increment/-decrement	add x0, x1, (x2+)	Adds the value of x1 and the value in memory at the address in x2, then increments x2, stores in x0.
PC-Relative	add x0, x1, 123(pc)	Adds the value of x1 and the value in memory at pc plus 123, stores in x0.

Syntax here looks like RISC-V but most of these instructions do not exist in RISC-V.

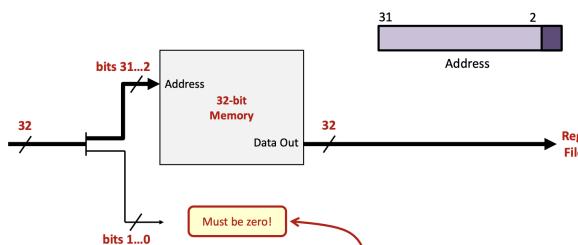
3.5.1 Word Addressed Memory

In a word addressed memory, the address is the index of the word in the memory.
The letters inside the word are identified as eg. for Hello World, H:3980, E:3981, L:3982,



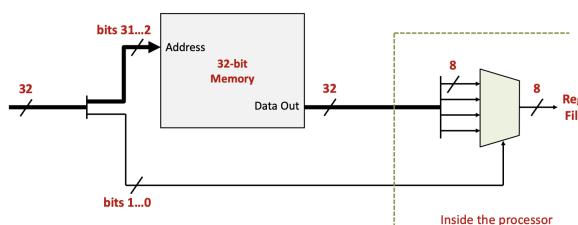
3.5.2 Loading Words (lw) and Instructions

The `lw` instruction is used to load a word from memory into a register.
The address of such words would necessarily be a multiple of 4 meaning the two least significant bits must be 0s. (to ensure the data is word aligned...)



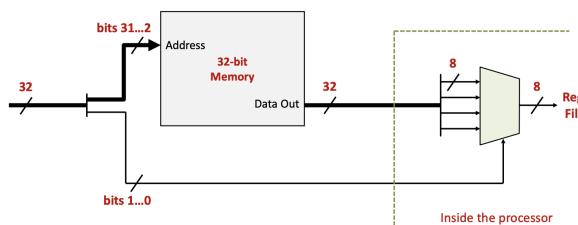
3.5.3 Loading Bytes (lb)

The `lb` (Load Byte) instruction doesn't require alignment because it only loads 1 byte (8 bits), which can be accessed at any memory address, unlike `lw` which requires word alignment to efficiently load 4 bytes (32 bits).
The `lb` instruction is used to load a byte from memory into a register.



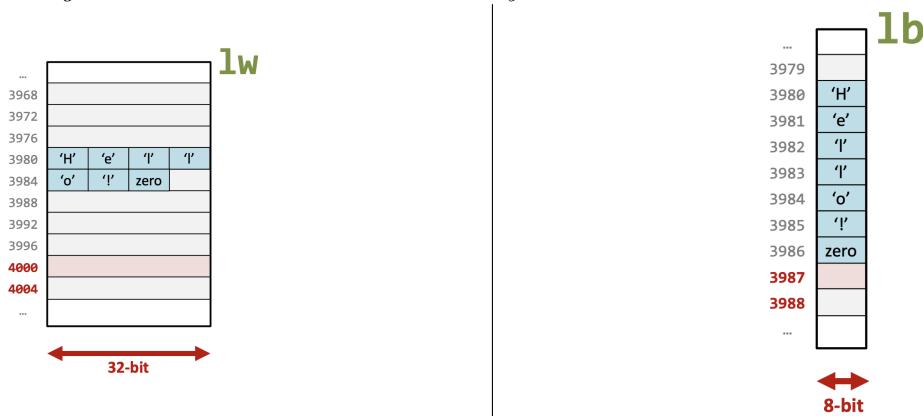
3.5.4 A Few More Load/Store Instructions

Access bytes (and half-words) as if memory were made of bytes



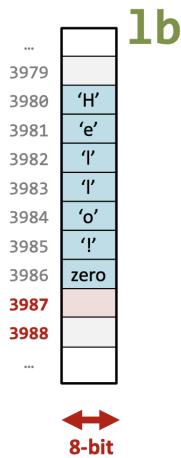
3.5.5 Access as it is more suitable

For example storing the "Hello!" zero value in the memory would like this:



Counting Characters in a String

As an example, for counting the number of characters in a string, the load byte instruction would be more suitable as seeing the string as a sequence of bytes makes use of the memory as a sort of array.



```

1  strlen:
2      mv t0, a0 # Copy the pointer (a0) into t0 to traverse the string
3      li t1, 0 # t1 will hold the length (initialized to 0)
4  loop:
5      lbu t2, 0(t0) # Load byte at address t0 into t2
6      beq t2, zero, end # If t2 is 0 (null byte), we are done
7      addi t1, t1, 1 # Increment the length counter (t1)
8      addi t0, t0, 1 # Point to the next character in the string
9  j loop # Repeat the loop
10 end:
11     mv a0, t1 # Move the length (t1) into a0 as the return value
12     ret # Return to caller

```

lbu is used here to ensure that the byte is treated as an unsigned value, which is the correct approach for processing characters in a string.

In a word addressed memory view, the code would look like such:

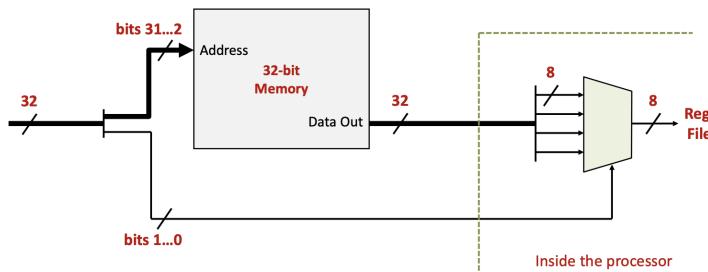
```

1  strlen:
2      li t1, 0          # t1 will hold the length (initialized to 0)
3  next_word:
4      li t2, 4          # t2 will count the bytes in a loaded word (four)
5      lw t3, 0(t0)      # Load four bytes at address t0 into t3
6  next_byte:
7      andi t4, t3, 0xff # Move the "little-end" in t4
8      beq t4, zero, end # If t4 is 0 (null byte), we are done
9      addi t1, t1, 1     # Increment the length counter (t1)
10     srli t3, t3, 8    # Prepare the next byte of the word in the "little-end" (t3)
11     addi t2, t2, -1    # One byte left in the loaded word
12     bneq t2, next_byte # If more bytes in t3, check the next
13     addi a0, a0, 4     # Else point to the next word of characters in the string
14     j next_word        # Repeat the loop
15 end:
16     mv a0, t1          # Move the length (t1) into a0 as the return value
17     ret                # Return to caller

```

3.5.6 Loading Bytes (lb)

Now, one may wonder in what ordering the bytes are stored in memory.



Which Byte Where?

Both ordering of bytes are valid the only thing we have to do is stick to one, the most generally used is little-endian as it's the RISC-V default and the Intel x86/x64 default.



Personal Remark : Mnemotecnic - Little Endian = Little End (The ending memory index takes the smallest(starting) data address), Big Endian = Big End.

Chapter 4

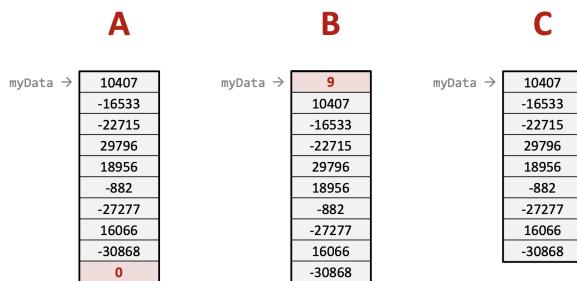
Part I(d) - ISA Arrays and Data Structures - W 2.2

4.1 Arrays

In higher level languages, are written like follows :

```
1 short[\[] myData = \{10407, -16533, -22715, 29796, 18956, \dots\}:
```

4.1.1 Different Ways to Store Arrays



- **A: Storing Arrays with a Null Terminator**

- In this method, the array is stored with each element represented using 16-bit integers.
- A null terminator (the value 0) is used at the end of the array to indicate its termination.
- This method is common when the array size is unknown in advance, and the null terminator acts as a signal to stop reading the data.

- **B: Storing Arrays with a Length Prefix**

- Here, the first element of the array contains the length of the array, stored as a 16-bit integer (in this case, the length is 9).
- The rest of the array is stored in consecutive memory locations, similar to method A.
- This method allows the array size to be known before reading all the data, making it more efficient for some use cases.

- **C: Storing Arrays without a Terminator or Length Prefix**

- In this case, the array is stored without a length prefix or a null terminator.
- The size of the array must be known externally, either through the code or an external mechanism.
- This method is the most compact but requires prior knowledge of the array's size.

4.1.2 Adding Positive Elements

Here we'll write the same program for the different ways of storing arrays.

The program will add all positive elements of an array of signed 16-bit integers. At call time, *a0* points to the array, at return time, *a0* contains the result.

A: Storing Arrays with a Null Terminator

A

myData →	10407
	-16533
	-22715
	29796
	18956
	-882
	-27277
	16066
	-30868
	0

```

1 add_pos: li t0, 0           # Initialize t0 to 0
2     lh t1, 0(a0)          # Load halfword from memory address a0 into t1
3     beqz t1, end          # Branch to 'end' if t1 equals zero
4     blez t1, donothing   # Branch to 'donothing' if t1 is less than or equal to
      zero
5     add t0, t0, t1        # Add t1 to t0 (only if t1 is positive)
6 doNothing:                 # This block does nothing for negative or zero values
7 # You can put other operations here if needed
8     j add_pos            # Jump back to the beginning of add_pos to check the next
      value
9 end:                      # Label 'end' for the program termination

```

B: Storing Arrays with a Length Prefix

B

myData →	9
	10407
	-16533
	-22715
	29796
	18956
	-882
	-27277
	16066
	-30868

```

1 add_pos_b:
2     lh t2, 0(a0)          # Load the length of the array into t2
3     addi a0, a0, 2        # Move to the first element of the array (skip the length
      prefix)
4     li t0, 0              # Initialize t0 to 0 for storing the sum
5 loop_b:
6     beqz t2, end_b        # If the length (t2) is zero, branch to 'end_b'
7     lh t1, 0(a0)          # Load the current array element into t1
8     blez t1, skip_b       # If t1 is less than or equal to zero, skip the addition
9     add t0, t0, t1        # Add t1 to t0 (only if t1 is positive)
10 skip_b:
11     addi a0, a0, 2        # Move to the next element in the array
12     addi t2, t2, -1       # Decrease the length counter
13     j loop_b             # Jump back to loop_b
14 end_b:                  # End label

```

C: Storing Arrays without a Terminator or Length Prefix

C

myData →	10407
	-16533
	-22715
	29796
	18956
	-882
	-27277
	16066
	-30868

 16-bit

```

1 add_pos_c:
2     li t0, 0           # Initialize t0 to 0 for storing the sum
3 loop_c:
4     beqz t2, end_c    # If the array size (t2) is zero, branch to 'end_c'
5     lh t1, 0(a0)       # Load the current array element into t1
6     blez t1, skip_c   # If t1 is less than or equal to zero, skip the addition
7     add t0, t0, t1     # Add t1 to t0 (only if t1 is positive)
8 skip_c:
9     addi a0, a0, 2     # Move to the next element in the array
10    addi t2, t2, -1    # Decrease the array size counter
11    j loop_c          # Jump back to loop_c
12 end_c:                # End label

```

4.1.3 Pointer to Memory vs Index in Array

Now we're wondering which one of these two ways of accessing the array is better.

Obviously the less instructions the better (not always true actually but well),
Pointer to Memory.

```

1 add_positive:
2     li t0, 0
3     mv t1, a1
4 next_short:
5     beq t1, zero, end
6     lh t2, 0(a0)
7     bltz t2, negative
8     add t0, t0, t2
9 negative:
10    addi a0, a0, 2
11    addi t1, t1, -1
12    j next_short
13 end:
14    mv a0, t0
15 ret

```

```

1 add_positive:
2     li t0, 0
3     li t1, 0
4 next_index:
5     bge t1, a1, end
6     slli t2, t1, 1
7     add t2, a0, t2
8     lh t3, 0(t2)
9     bltz t3, negative
10    add t0, t0, t3
11 negative:
12    addi t1, t1, 1
13    j next_index
14 end:
15    mv a0, t0
16 ret

```

In C

Writing this in C for better understanding. Again, which one is better?

Obviously the less instructions the better (again not always true but ah), Index in array
Pointer to memory

```

1 short sum = 0;
2 short *ptr = myData;
3 short *end = myData + N;
4 while (ptr < end) {
5     if (*ptr > 0) {
6         sum += *ptr;
7     }
8     ptr++;
9 }
```

Index in array

```
1 short sum = 0;
2 int i;
3 for (i = 0; i < N; i++) {
4     if (myData[i] > 0) {
5         sum += myData[i];
6     }
7 }
```

We need a good compiler

Seeing this, the idea would be to have a sufficiently good **compiler** (check I.4.3.2 if needed) such that we write our C code in *Index* in array, and we get Pointer to memory code in assembly. Thus writing better code but also getting better performance.

Another type of collection we could've used to store the data is a *Linked List*.

Linked lists are useful for efficiently inserting and deleting elements, especially in the middle of the list.

Each 32-bit element in a linked list contains 16 bits for the value and 16 bits for the address of the next element, enabling efficient insertions but slower sequential access compared to arrays.

