

Computer Architecture

IN BA3 - Paolo IENNE

September 11, 2024

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.

Contents

Contents	3
1 Part I(a): ISA Reminder, Assembly Language, Compiler - W 1.1	4
1.1 From High Level Languages to Assembly Language	4
1.1.1 High Level Languages	4
1.1.2 Assembly Language	4
1.2 Processors	5
1.3 Joint or Disjoint Program and Data Memories	6
1.4 The Encoding problem	7
1.4.1 The Stupid Solution	7
1.4.2 RISC-V Encoding (The Solution)	7
1.4.3 Automating this process	8
1.5 ISA (Instruction Set Architecture)	9

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 such code because we're able to build a Finite State Machine that can execute it (which cannot be done directly for C code.)

In this language, we get a more rigid format, a sequence of numbered instructions, an *opcode* and predefined variable names. We also get a way to jump between lines of code.

```

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

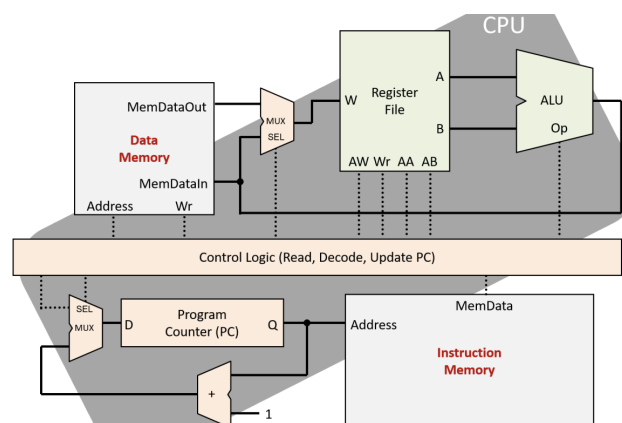
```

1.2 Processors

Last year, we studied Processors for a good reason, they are the closest machines to the code we write.

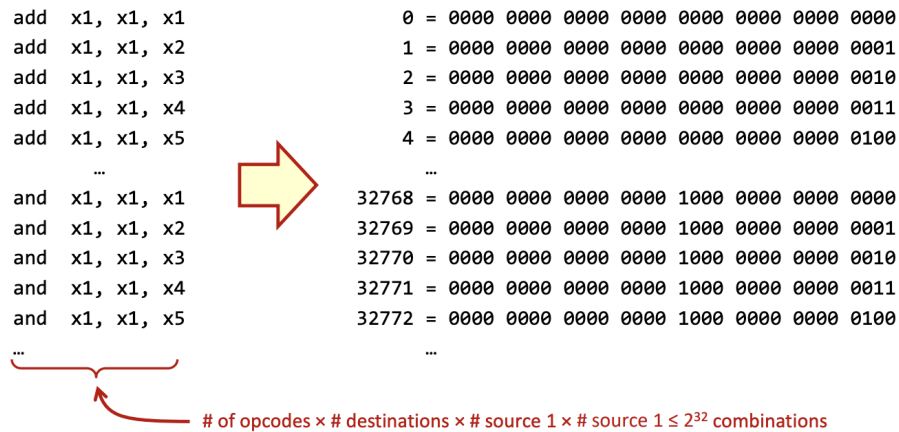
Remember, a processor can be decomposed into five components:

- **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.

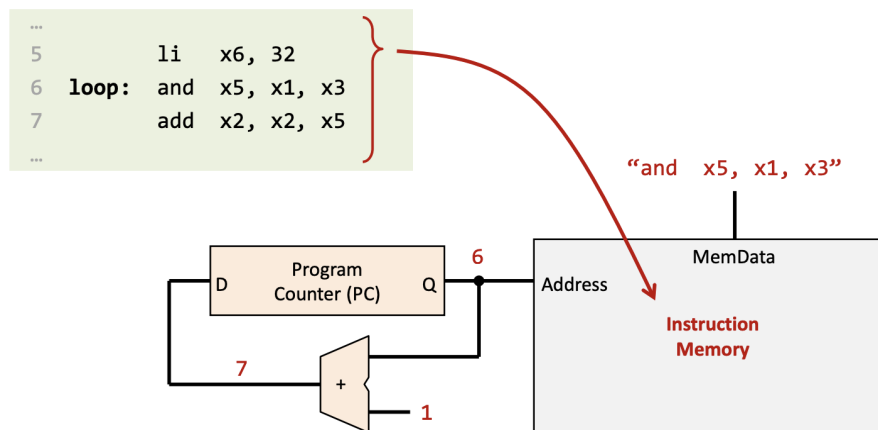


We may distinguish three types of operations:

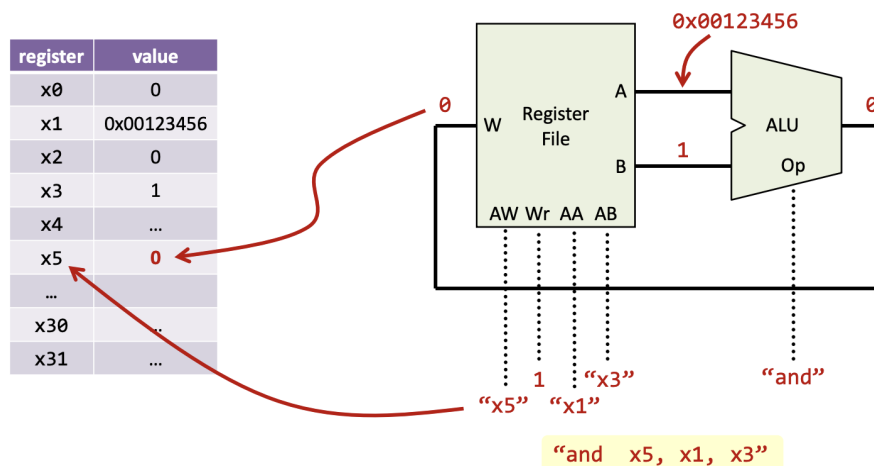
Encoding



Fetching



Executing

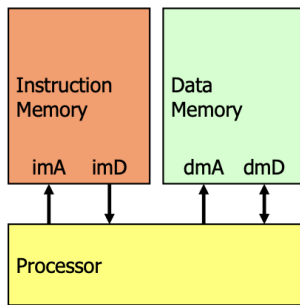


1.3 Joint or Disjoint Program and Data Memories

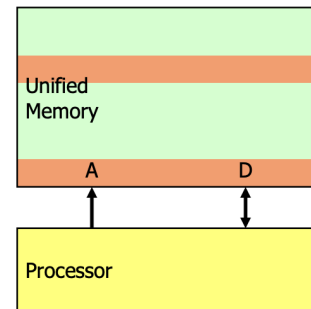
There are two main types of architectures one called the Harvard Architecture (Where the data and the memory are separate) and the other called Unified Architecture (where data is shared with the program)

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	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 × # destinations × # source 1 × # 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. This approach optimizes both memory usage and processing speed by limiting the number of bits required to represent instructions, while still allowing for a large variety of operations.

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
srlui 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
sraui 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	$rd \leftarrow rs1 + sext(imm)$	I		0x0	0x13
sub rd,rs1,rs2	$rd \leftarrow rs1 - rs2$	R	0x20	0x0	0x33
lui rd,imm	$rd \leftarrow imm$				
auipc rd,imm	$rd \leftarrow pc$				

Complete ISA on
Moodle!

Instruction types

	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
S	funct7		imm[4:0]		rs1	funct3		rd		opcode			Register-Immediate Shift
I	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

RISC-V encoding

1.4.3 Automating this process

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	0101 0101 0101 0000 0100 0111 1010 1110
1	li	x2, 0	0001 0100 1001 1101 0011 0000 1100 1001
2	li	x3, 1	1101 1100 1101 0110 0000 1101 0001 0111
3	li	x4, 0	0010 0011 1101 0110 0010 0000 0001 1001
4	li	x5, 0	1100 1010 1011 1010 0111 0100 0000 0110
5	li	x6, 32	1111 0010 1001 0011 1001 1110 1001 1101
6	loop:	and x5, x1, x3	0011 0000 0010 0111 1111 0000 0100 0011
7		add x2, x2, x5	0111 1001 0101 1101 1000 1000 0111 1011
8		srlui x1, x1, 1	1100 1010 1011 0000 0100 0100 0110 0101
9		addi x4, x4, 1	0111 1001 0010 0110 0000 0011 0001 0010
10		bne x4, x6, loop	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. This machine code can then be executed by the processor. The compiler performs various stages, such as lexical analysis, parsing, optimization, and code generation, ensuring that the program runs efficiently on the target hardware.

int data = 0x00123456;	0	li	x1, 0x00123456
int result = 0;	1	li	x2, 0
int mask = 1;	2	li	x3, 1
int count = 0;	3	li	x4, 0
int temp = 0;	4	li	x5, 0
int limit = 32;	5	li	x6, 32
do {	6	loop:	and x5, x1, x3
temp = data & mask;	7		add x2, x2, x5
result = result + temp;	8		srlui x1, x1, 1
data = data >> 1;	9		addi x4, x4, 1
count = count + 1;			bne x4, x6, loop
} while (count != limit);			

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.

