Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Instruction Set Architecture

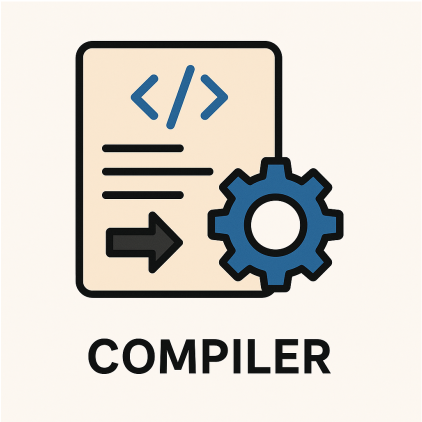
How to talk to a Computer?

- **Computers can be given "instructions"**
- We have a set of instructions for every computer — called **instruction set**
- **When you write a program, you write instructions..**
  - More details later…
- Every instruction some hardware circuit implemented inside the processor to get its job done.
- **Instruction Set Architecture**: specifies the set of instructions a processor understands, their encoding, how they access memory etc…

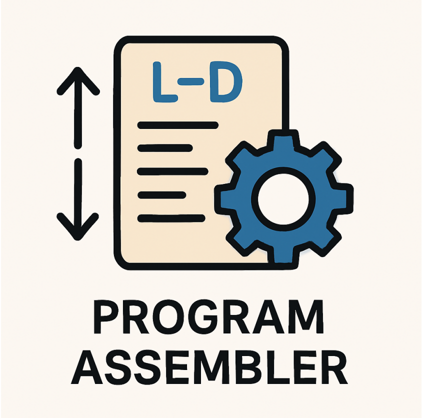
Image generated by ChatGPT

What happens when you write a program

- **Say we write:**
  - a = b + c;



- There is a software program called **compiler**
  - Takes our code and encodes in terms of the instructions available for the computer
  - `add reg1, reg2, reg3`



- There is another program called **assembler** which converts the instruction (sequence) to bits
- 0101110000110101

Image generated by ChatGPT

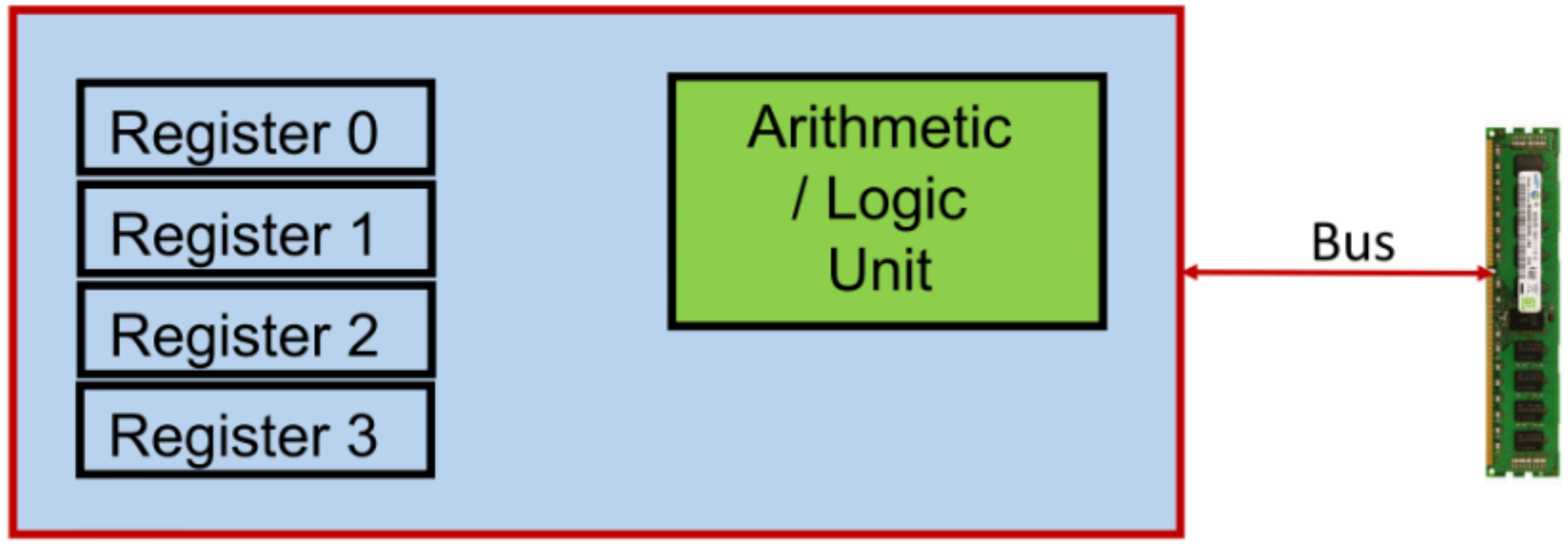
How to talk to a Computer?

- **Instruction Set Architecture**: specifies the set of instructions a processor understands, their encoding, how they access memory etc…
  - **End of the day even your ChatGPT is a sequence of instructions** (many billions or trillions).
- Instruction set is basically an **abstraction layer**
  - **Hides the complexity of hardware from the software designers,**
  - **Interfaces the software and hardware.**

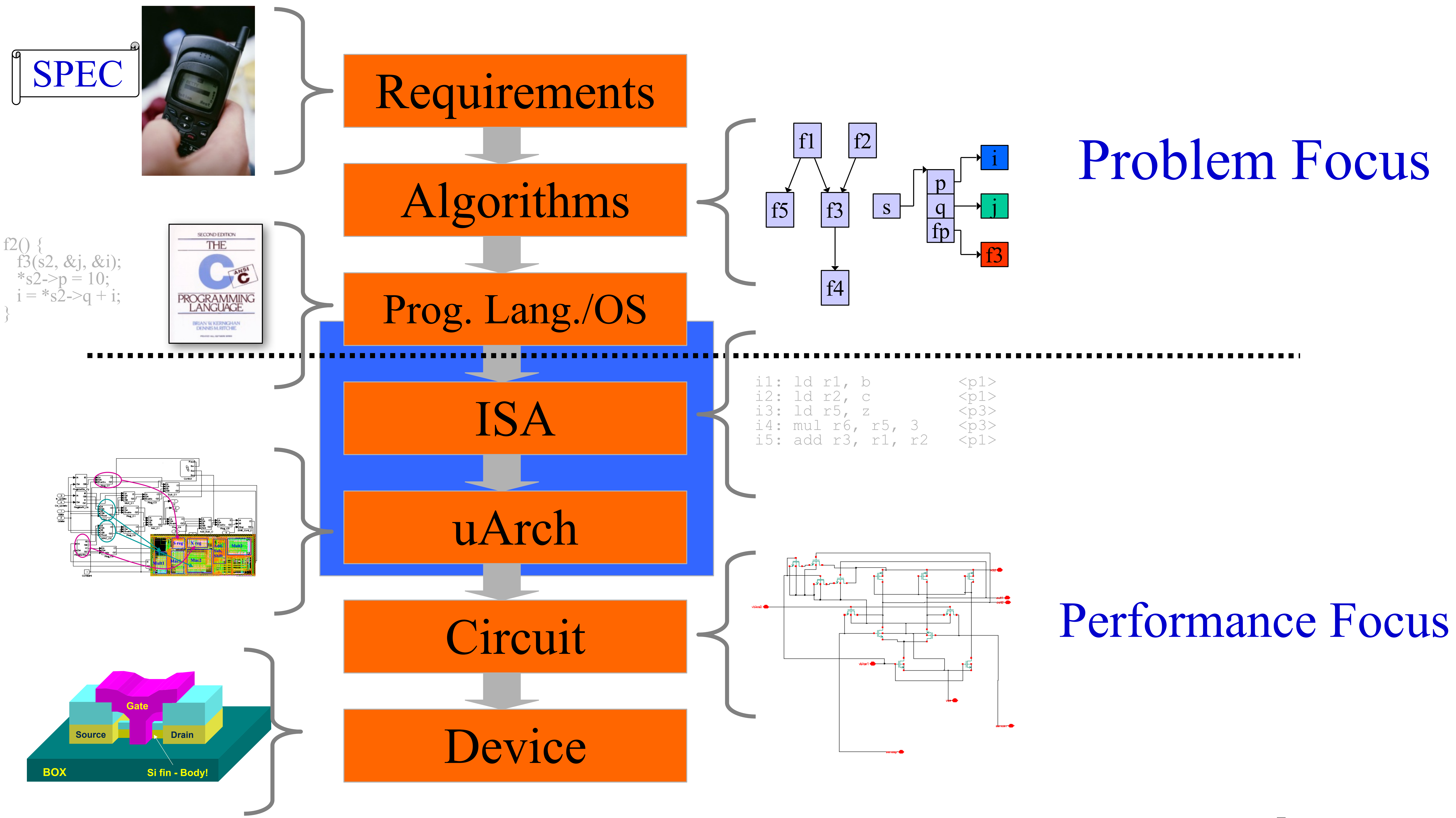
Image generated by ChatGPT

Let’s get into the processor a bit

- It is a sequential circuit with a **limited** number of registers.
- It interacts with an external “memory”.
- Every instruction operates on some **operands** and generate results.
- Results and operands are stored in **registers**.
- **But they can also be in memory as the number of registers are limited**
- Note that typically such memory **(**called **DRAM** or **Dynamic Random Access Memory)** is **off chip —outside the processor**
- **To operate, you have to bring the data from memory and store the results back**

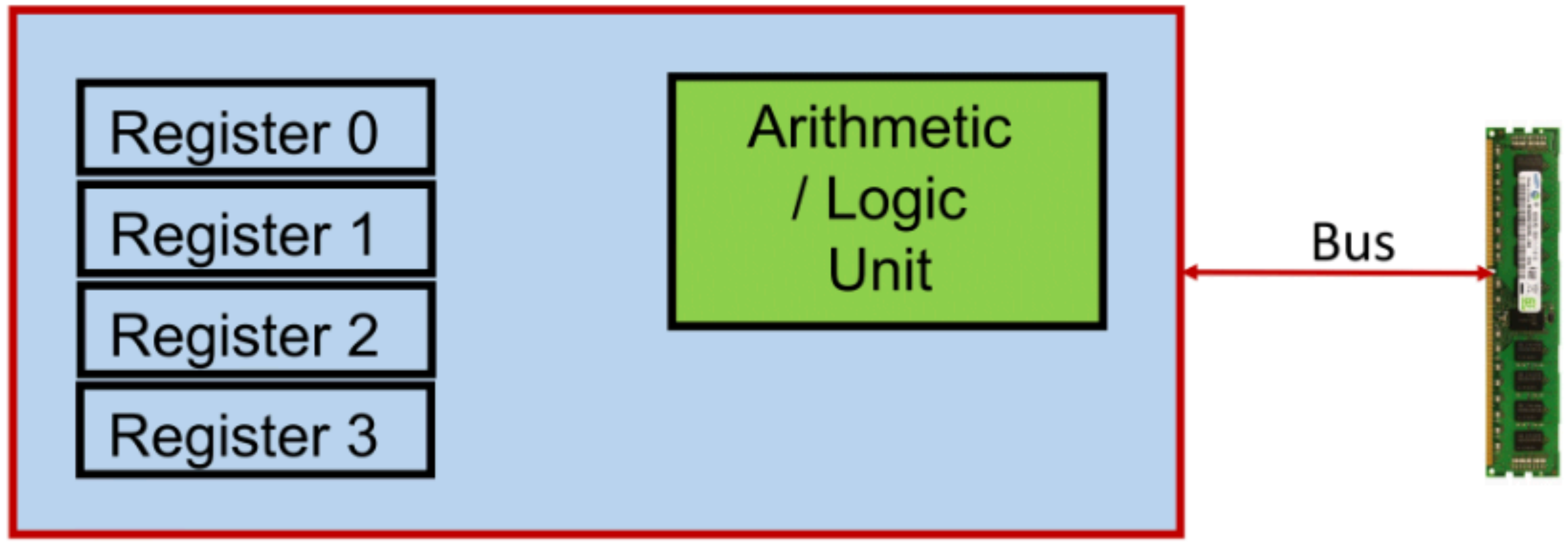


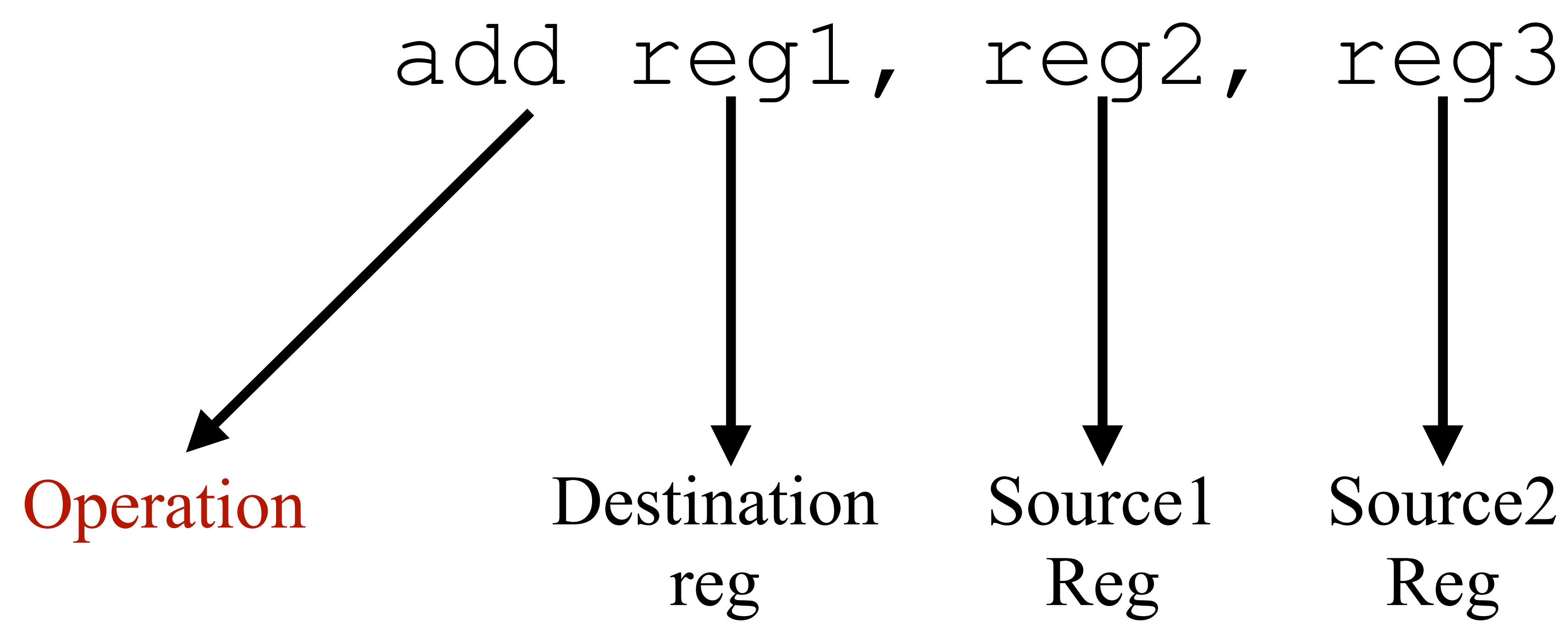
The Big Picture



Dissection of an Instruction

- Let’s focus on the simplistic view of the processor





- Most of the arithmetic/logical instructions can take this form — not all though

Instruction Set Architectures (ISA)

- There are many…
  - Intel uses **X86**
  - Apple uses a version of **AArch64** (ARM)
  - The entire world of embedded processors like ST-Microelectronics uses ARM
  - Now **RISC-V** is becoming a mainstream trend.
  - We shall study MIPS — a simple to understand ISA

Instruction Set Architectures (ISA)

- We shall study MIPS — a simple to understand ISA
    - Great for beginning…
    - Similar to ARM
    - Still in use in the embedded devices
        - Your smart card
        - Modems
        - Bitcoin-wallets

Now let's write some MIPS

- We shall name the registers as `$0`, `$1`, or `$a0`, `$g1` etc…
- Now we shall try something a bit more complex…

```
add reg1, reg2, reg3
```

```
add $0, $1, $2
```

Now let's write some MIPS

- Let's compute: `a = b+c-d`
- No idea? — get idea :P

```
add reg1, reg2, reg3
```

```
add $0, $1, $2
```

Now let's write some MIPS

- Let's compute: `a = b+c-d`
- Assume we have add and sub instructions taking two sources and one destination register

```
add $0, $1, $2
sub $0, $1, $2
```

Now let's write some MIPS

- Let's compute: `a = b+c-d`
- Assume we have add and sub instructions taking two sources and one destination register

```
add $0, $1, $2

sub $0, $1, $2
```

- First' let's simplify :
  - `t = b+c`
  - `a = t-d`

- **Observe**: I use a temporary register…

- Now, I can map to instructions..
  - `add $r0, $r1, $r2 //t = b+c`
  - `sub $d0, $r0, $r3 //a = t-d`

Now let's write some MIPS

- Let's try: `f = (g+h)-(i+j)`

Now let's write some MIPS

- Let's try: `f = (g+h)-(i+j)`

  - `add $r0, $r1, $r2 //x = g+h`
  - `add $r3, $r4, $r5 //y = i+j`
  - `Sub $r0, $r0, $r3 //f = x-y`

- **Food of thought**: Well, do I really need to reuse registers???

Ok…A Few MIPS Details…

- We have 32 registers in the processor
  - So we have to reuse registers, no other option…
  - Typically, registers are 32-bits…

- But why don't we have infinite number of registers
  - Well, every piece of register is a real hardware…

- **But**: Why 32??

Ok…A Few MIPS Details…

- We have 32 registers in the processor
  - So we have to reuse registers, no other option…
  - Typically, registers are 32-bits…
  - Each instruction also encoded in 32 bits

- **But**: Why 32??

- But why don't we have infinite number of registers
  - Well, every piece of register is a real hardware…

The choice depends on several factors, like the speed of the execution, the usage and size of memory, the size of code, the encoding and decoding of instructions….**It's not a random choice**…

Immediate Instructions…

- b = a + 7

```
addi $r0, $r1, 7
```

- We don't need a register for the constant…
  - Can you tell me why?? Just guess…

Immediate Instructions…

- `b = a + 7`

```
addi $r0, $r1, 7
```

- We don't need a register for the constant…
  - Can you tell me why?? Just guess…

- `i` stands for 'immediate'
- The constant is in **2's complement form and of 16 bits.**
- **Question: Do I need a `subi` instruction??**

Zero Is Very Special in Our Life…

- MIPS has a register which is called `$zero`
  - It stores 0
  - What is the purpose?
    - Well, a lot…you will see
  - A simple use of `$zero`

```
add $r1, $r0, $zero  // a = b
```

- But again, why???

Zero Is Very Special in Our Life…

- MIPS has a register which is called `$zero`
  - It stores 0
  - What is the purpose?
    - Well, a lot…you will see
  - A simple use of `$zero`

```
add $r1, $r0, $zero  // a = b
```

- But again, why??? — just not needed

a=b….The Pseudo-Instructions

- You can still write…

```
move $r1, $r0  // a = b
```

- But it is a pseudo-instruction
- Internally it converts to `add`
- Once again an engineering choice
- There are many such pseudo-instructions. See:

https://en.wikibooks.org/wiki/MIPS_Assembly/Pseudoinstructions

Logical Instructions

- Your good old Boolean algebra

sll, srl, and, or, nor, andi, ori etc

No not instruction ☺, well not is nor with one operand=0

- Remember: **These are bitwise operations…**
  - **Treats the operands as bit strings instead of numbers**