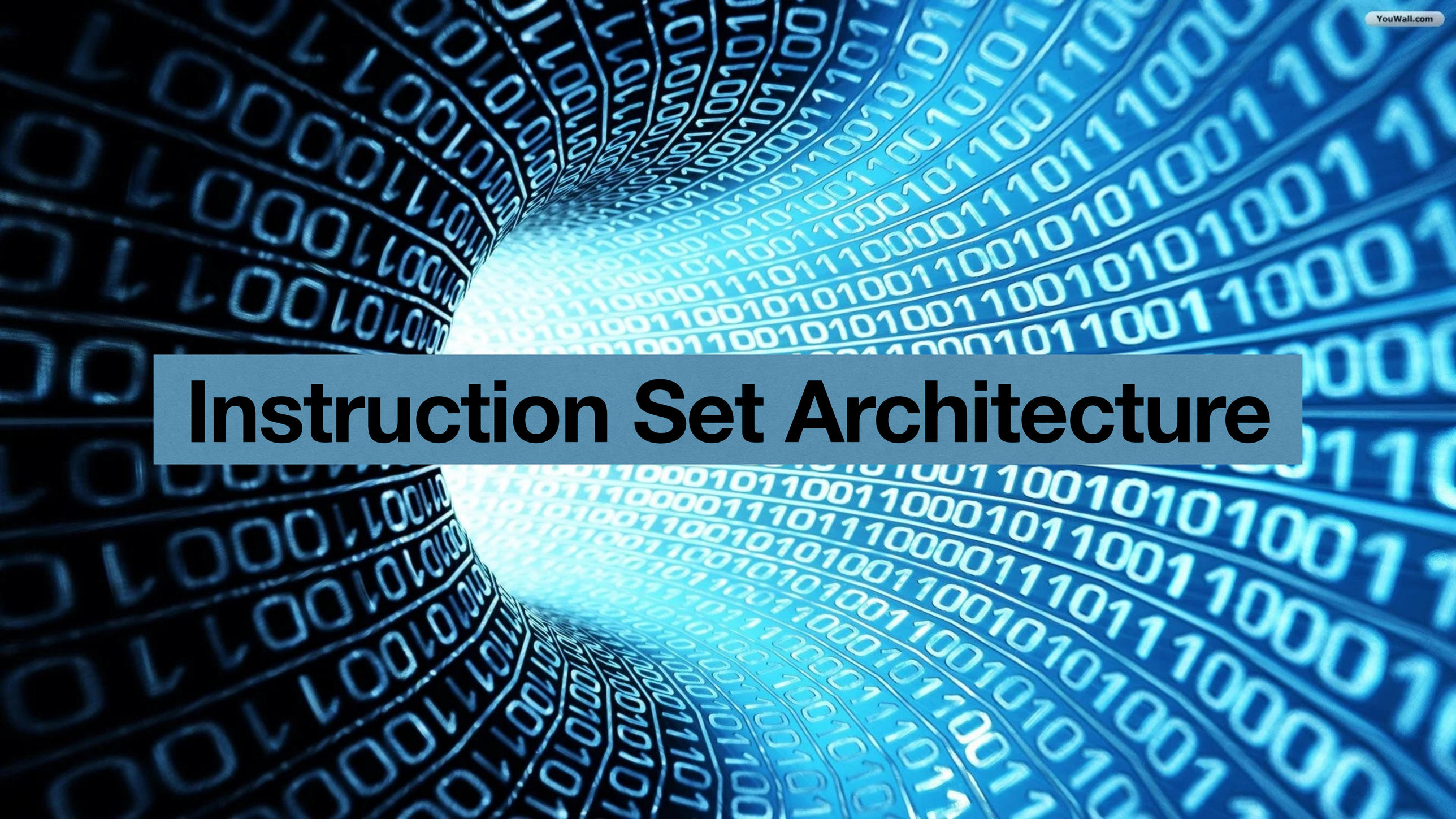
Digital Logic Design + Computer Architecture

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.**

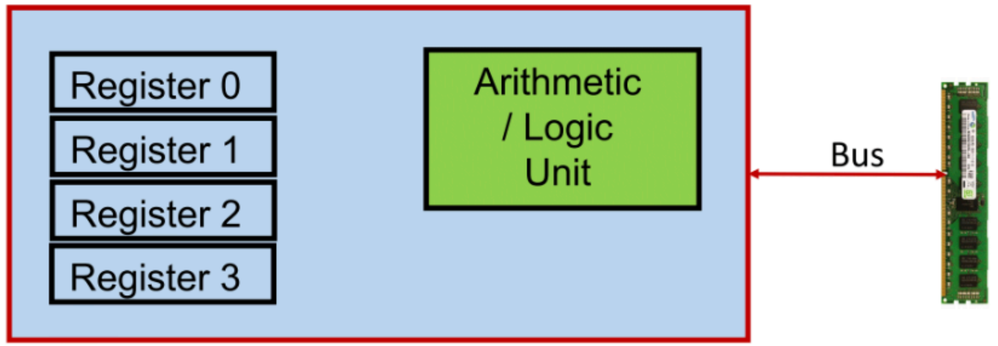
Image generated by ChatGPT

Let’s get into the processor a bit

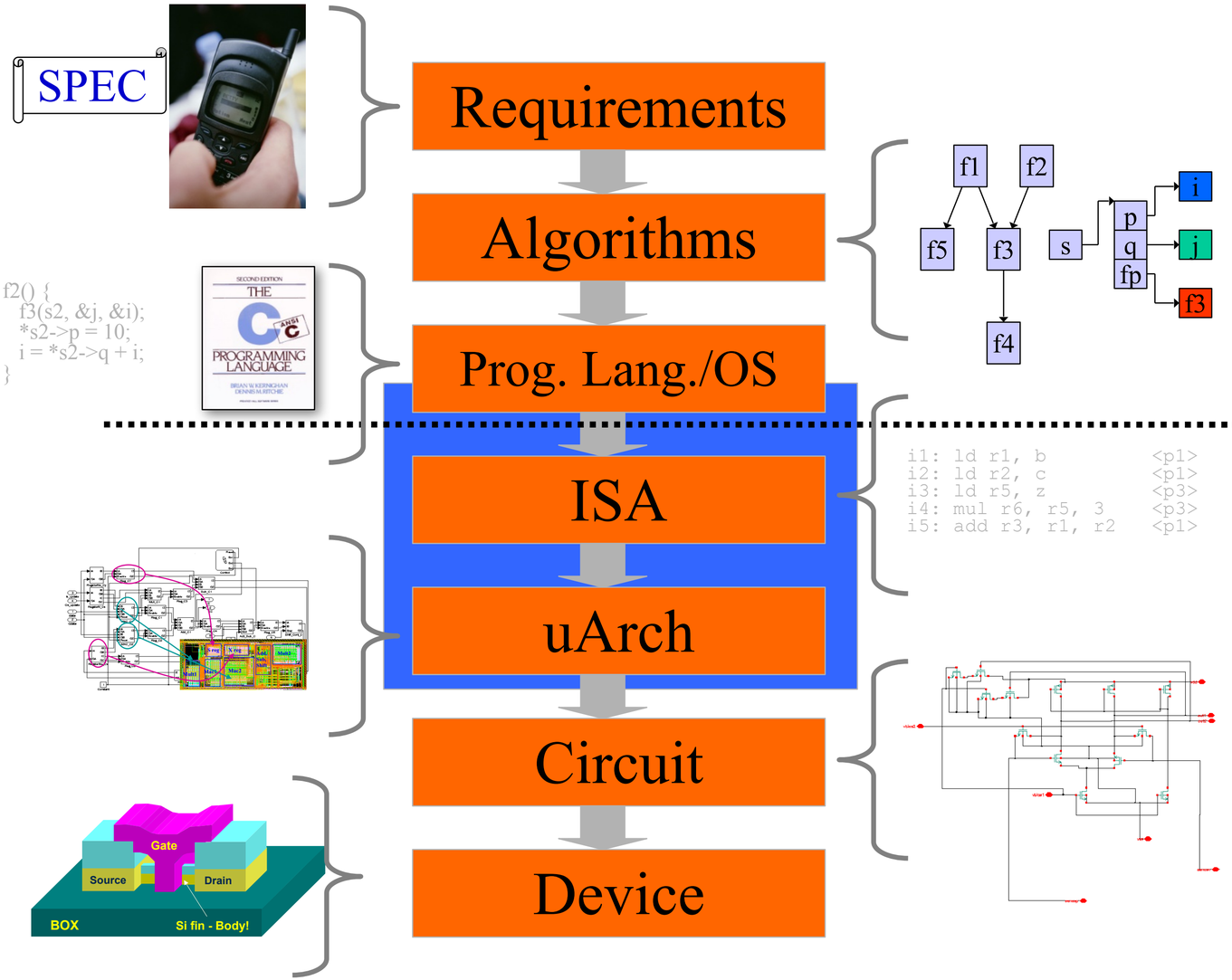
- It is a sequential circuit with a **limited** number of registers.
- It interacts with an external “memory”.
- Every instruction operates on some **operands** and generate results.

- 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

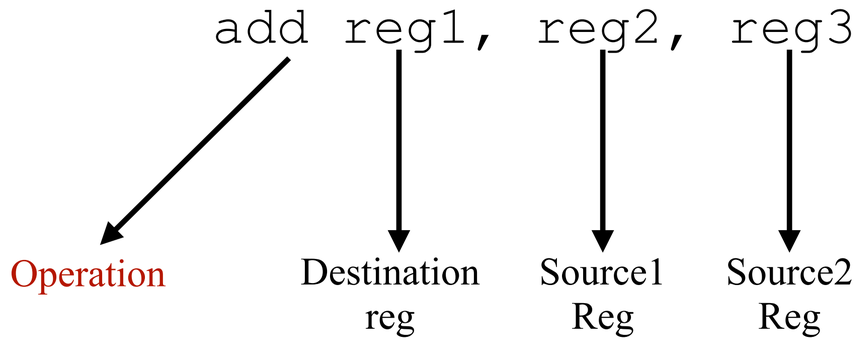


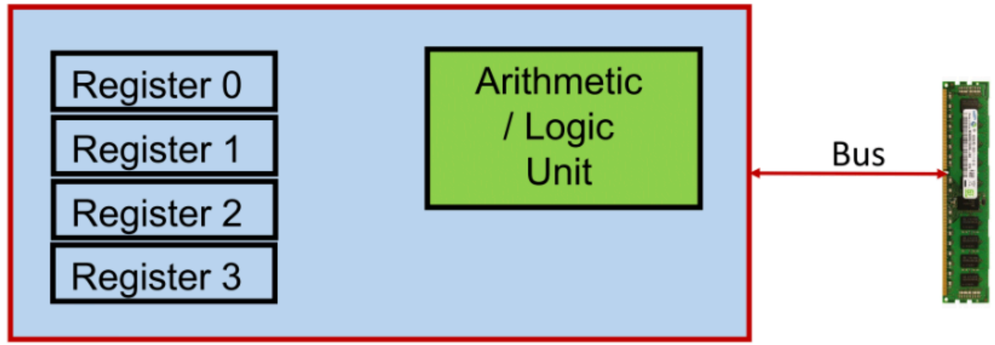
Problem Focus

Performance Focus

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

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

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: <u>Do I need a `subi` instruction??</u>**

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**

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

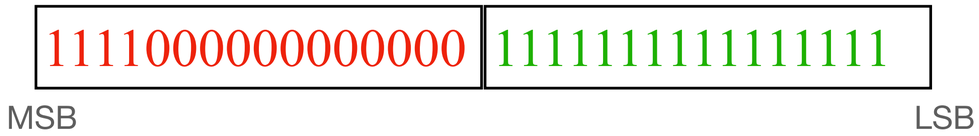
- Remember: **These are bitwise operations…**
  - **Treats the operands as bit strings instead of numbers**

Critical Thinking…

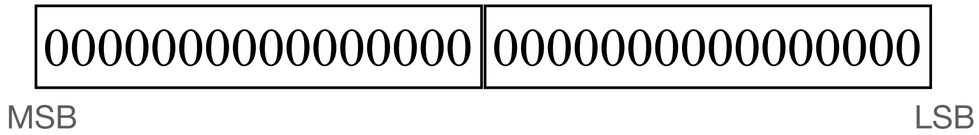
- We have seen that constants are 16 bits…
- But registers are 32-bits…
- How to store a 32-bit constant in a register???
  - Let's say the constant is:
    - 11110000000000001111111111111111
    - In Hex: 0xF000FFFF
- **Info**: You have the following new instruction:
  - **`lui $r0, const`** `// loads cosnt in the upper 16 bits of the register $r0`

Critical Thinking…

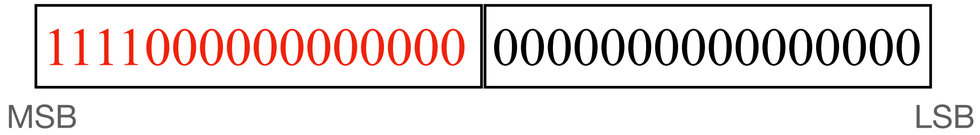
- Think, how the data will be represented inside your register…



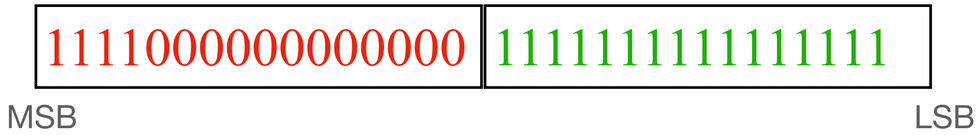
- Initially The register `$r0` is at (simplifying assumption…does not matter)



- Now do: `lui $r0, 0xF000`



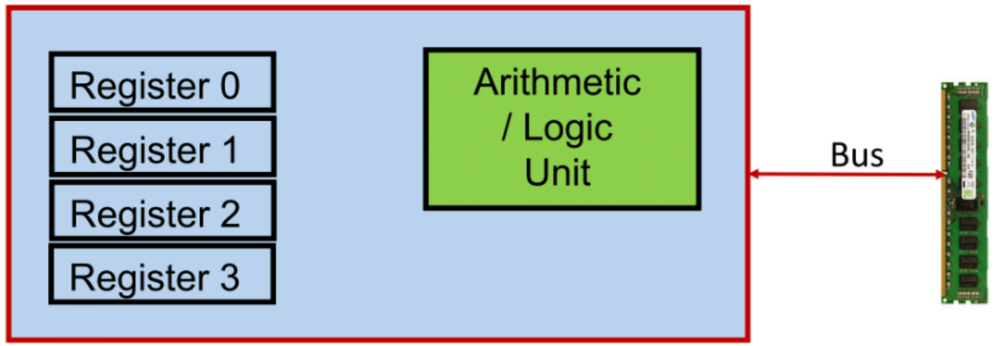
- **Now do,** `addi $r0, 0xFFFF`



- You can also do `ori`

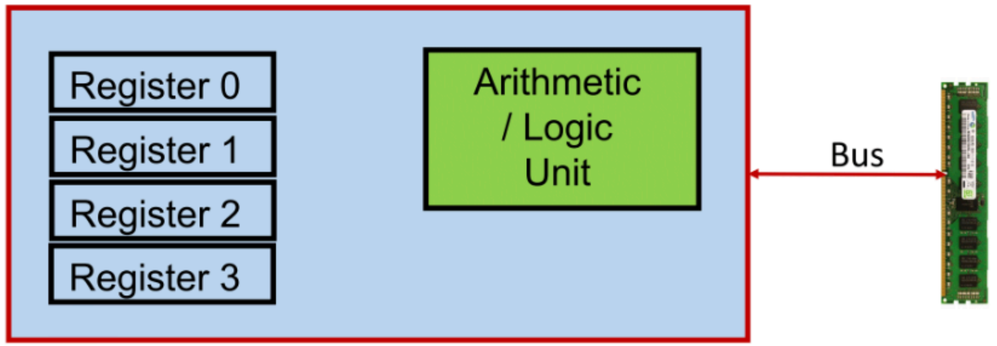
How to Use Your Memory??

- Recall, that MIPS only have 32 registers.
- Have you ever cared about counts while declaring variables in your program? — No way…
- Then how things work?
  - How can every program fits itself in 32 registers?



How to Use Your Memory??

- **Solution**:
  - Just store things in an external memory
  - Fetch the data to registers whenever it is required
  - Store the results after processing.
  - But still something is missing here…What is that??



How to Use Your Memory??

- Name this person?

How to Use Your Memory??

- Name this person?
  - John Luis von Neumann

How to Use Your Memory??

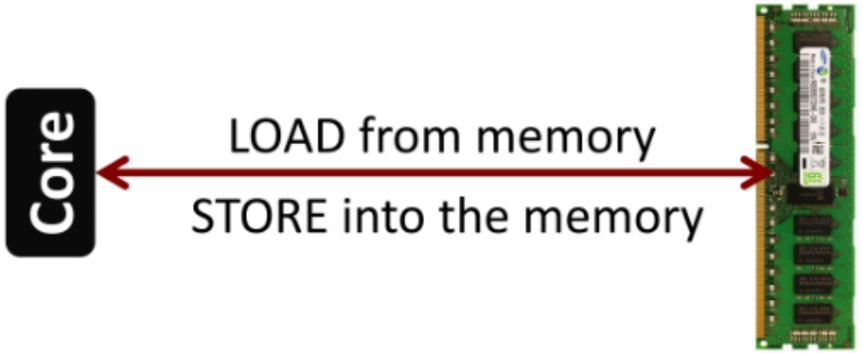
- In the old days, “programming” involved actually changing a machine’s physical configuration:
  - by flipping switches or connecting wires.
  - Memory only stored data that was being operated on.
- Then around 1944, **John von Neumann and others got the idea to encode instructions in a format that could be stored in memory just like data. — Stored program paradigm**
  - The processor interprets and executes instructions from memory

How to Use Your Memory??

- In the old days, “programming” involved actually changing a machine’s physical configuration:
  - by flipping switches or connecting wires.
  - Memory only stored data that was being operated on.
- Then around 1944, **John von Neumann and others got the idea to encode instructions in a format that could be stored in memory just like data. — Stored program paradigm**
  - The processor interprets and executes instructions from memory

Memory Instructions

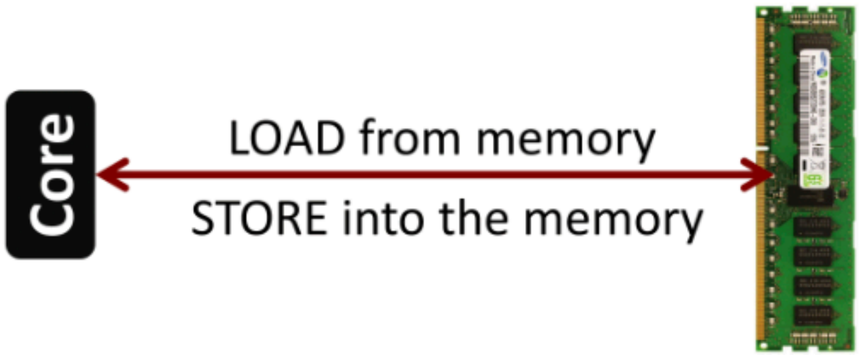
- **Load-Store Architecture**:
  - Load your data to process
  - Store it back…
- Instructions are handled in a slightly different manner….will come to that…




```
lw $t0, 1($a0)     # $t0 = Memory[$a0 + 1]
sw $t0, 1($a0)     # Memory[$a0 + 1] = $t0
```

Memory Instructions

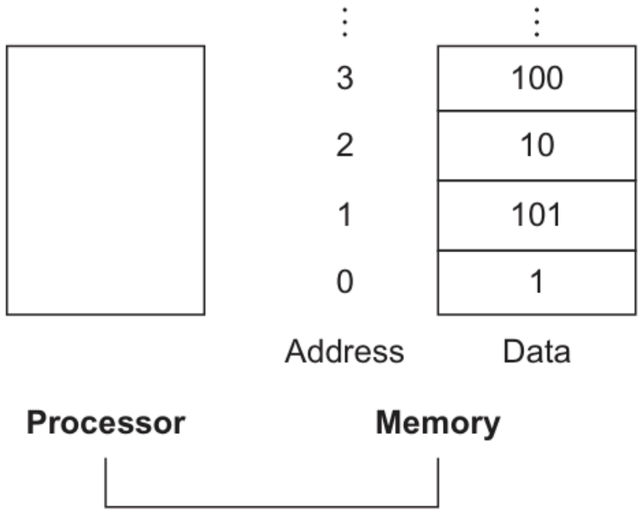
- **Load-Store Architecture**:
  - Load your data to process
  - Store it back…
- Instructions are handled in a slightly different manner….will come to that…
- But, a critical question:
  - How do you know where to find the data inside memory?




```
lw $t0, 1($a0)     # $t0 = Memory[$a0 + 1]
sw $t0, 1($a0)     # Memory[$a0 + 1] = $t0
```

Memory Instructions

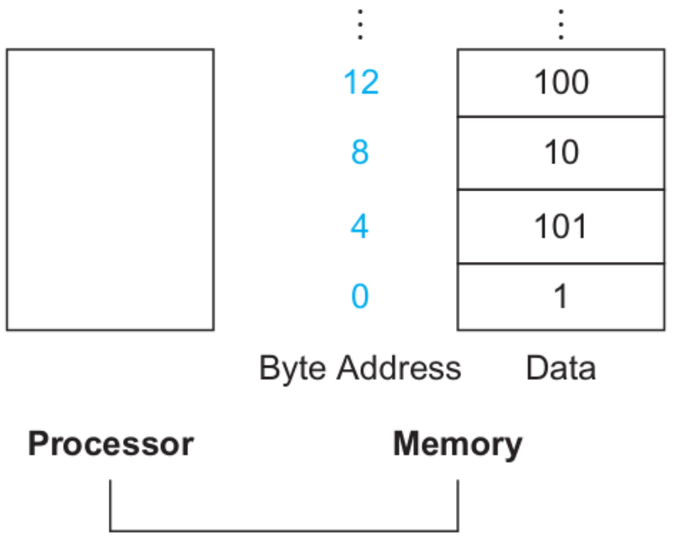
- But, a critical question:
  - How do you know where to find the data inside memory?
    - Memory has addresses
    - Think it like a large contiguous array…
    - **Every byte in memory has an unique address**
      - **Byte-addressable**
  - **BTW, each address is 32-bit in MIPS**



Memory Instructions

```
lw $t0, 1($a0)     # $t0 = Memory[$a0 + 1]
sw $t0, 1($a0)     # Memory[$a0 + 1] = $t0
```

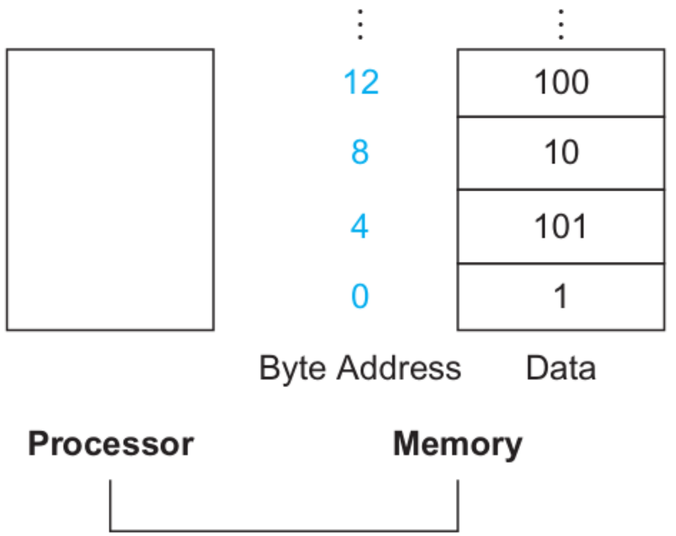
- The `lw` is interpreted as "load word"
  - MIPS also have other variants like "load byte" (`lb`)
- Data comes in `$t0`.
- But what is the 1(`$a0`) part signify?
  - `$a0` is the *base address* of the location you want to read from memory
  - 1 is called the *offset*.
- But why don't you read directly?



Memory Instructions

```
lw $t0, 1($a0)     # $t0 = Memory[$a0 + 1]
sw $t0, 1($a0)     # Memory[$a0 + 1] = $t0
```

- The `lw` is interpreted as "load word"
  - MIPS also have other variants like "load byte" (`lb`)
- Data comes in `$t0`.
- But what is the 1(`$a0`) part signify?
  - `$a0` is the *base address* of the location you want to read from memory
  - 1 is called the *offset*.
- But why don't you read directly?
  - Again a design choice, to ease compilation, programming, and hardware design…

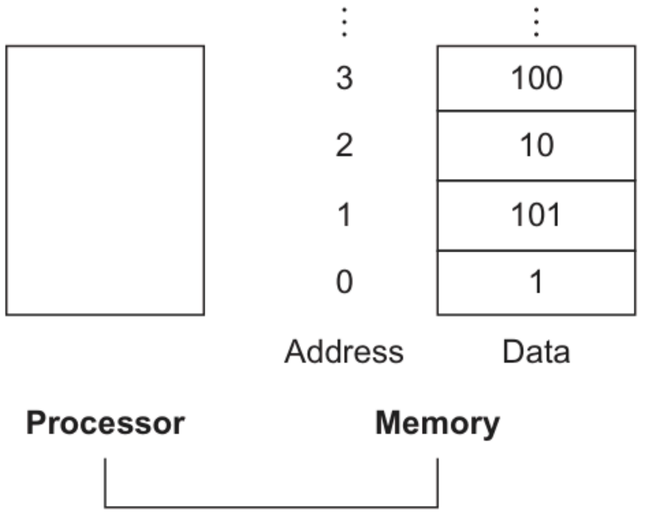


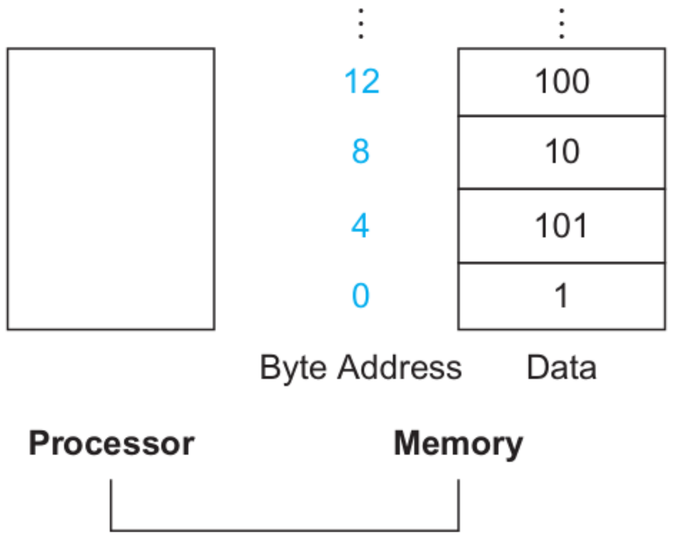
Memory Instructions: Word vs. Byte

```
lw $t0, 1($s0)        lb $t0, 1($s0)
```

- `lw` is interpreted as "load word"
- `lb` is "load byte"
- For the lw, we need the base+offset ($s0 + 1) to be **always divisible by 4 — word alignment**
- Why?
- Nothing such for lb

- **What Lies Beneath?**
  - `lb` just read the byte in the calculated address
  - `lw` reads four consecutive bytes starting from the calculated address.
- Why word alignment — again, it simplifies hardware OS, compiler….



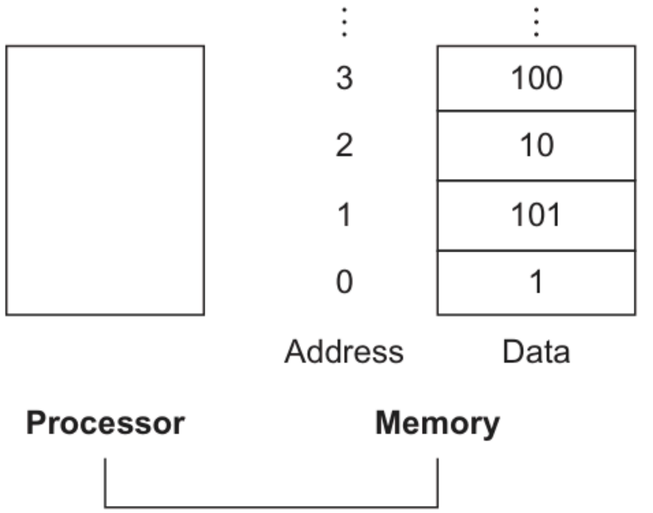


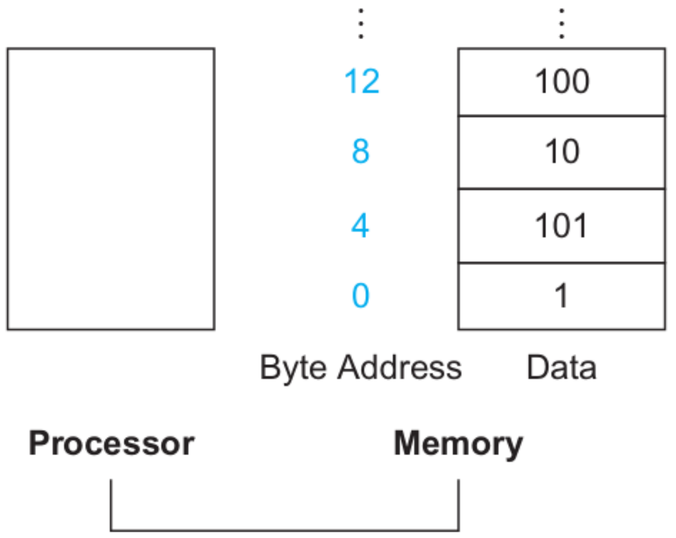
Memory Instructions: Word vs. Byte

```
lw $t0, 1($s0)        lb $t0, 1($s0)
```

- `lw` is interpreted as "load word"
- `lb` is "load byte"
- For the lw, we need the base+offset ($s0 + 1) to be **always divisible by 4 — word alignment**
- Why?
- Nothing such for lb

- **What Lies Beneath?**
  - `lb` just read the byte in the calculated address
  - `lw` reads four consecutive bytes starting from the calculated address.
- Why word alignment — again, it simplifies hardware OS, compiler….

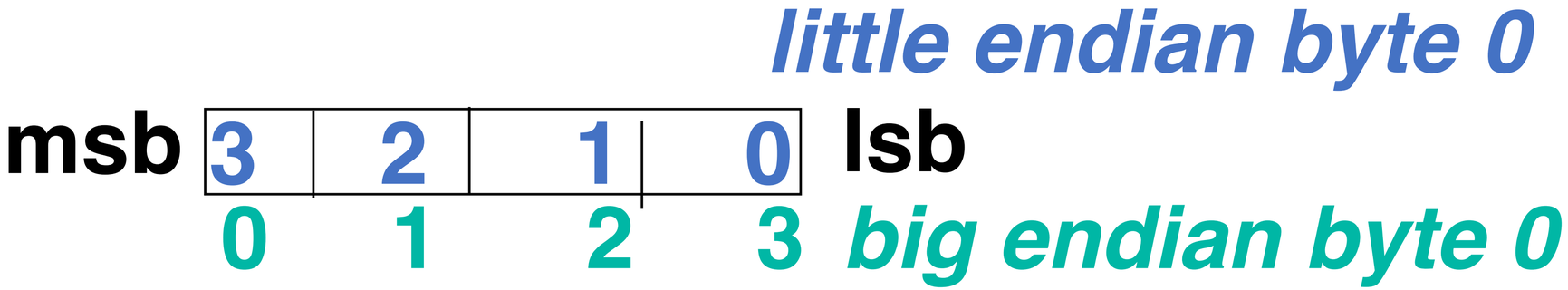




Endianness (Byte ordering within a word)

- Big Endian: address of most significant byte = word address
  (xx00 = Big end of word), MIPS

- Little Endian: address of least significant byte = word address
  (xx00 = Little end of word), x86



Just for an example, do not take it for granted …

```
unsigned int i = 1;
char *c = (char*)&i;  // reading the LSB
Printf (“%d”, *c);
```

```
unsigned int i = 12345678;
char *c = (char*)&i;
Printf (“%d”, *c);
```

```
unsigned int i = 1;
char *c = (char*)&i;  // reading the LSB
Printf (“%d”, *c);
```

Little endian: 1

Big endian: 0

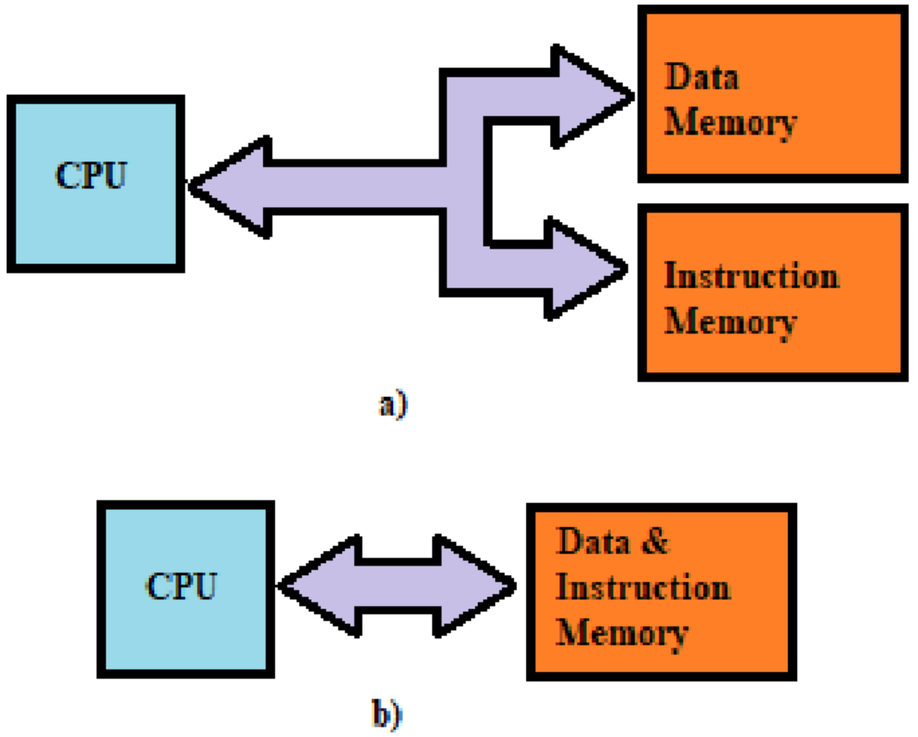
```
unsigned int i = 12345678;
char *c = (char*)&i;
Printf (“%d”, *c);
```

Little endian: 78

Big endian: 12

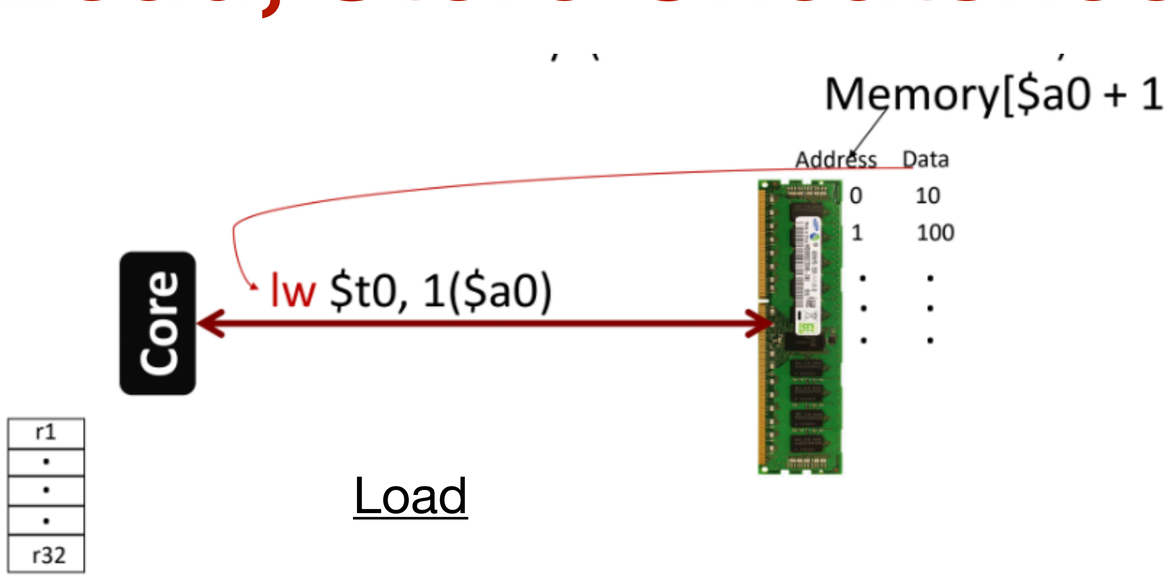
Another Important Point…

- Ok, **Von Neumann** said, data and code both are stored in the same memory.
  - In practice, this may lead to an issue — at a specific interval of time, **you can either fetch a data or an instruction**.
    - Affects parallelisation
- **What if you separate the data and instruction memory and buses?**
  - That is called **Harvard Architecture.**
  - Modern commercial systems use a combination of both
    - RAM stores both instruction and data
    - But there are other intermediate memory (**caches**) which are separated for instruction and data

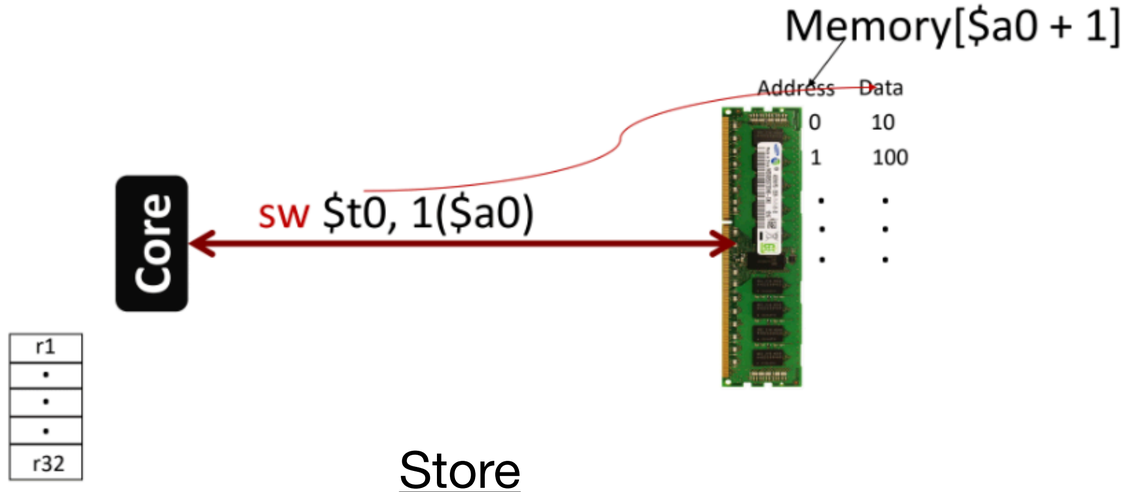


Source: Internet

Load, Store Cheatsheet



Load



Store

Program Counter

Points to the next instruction in the memory to be fetched

```
g = h + A [8];

PCX: lw $t0, 8($3)        # A[8]
PCY: add $s1, $s2, $t0    # g = h + t0

PCY = PCX+4
```

Load+Store+Instruction-fetch

Summary…

- Data and instructions at the same place
- Registers are limited — 32 bit wide
- Instructions are 32 bit wide
- Registers are accessed by names
- Memory is accessed by addresses