ECS404U: Computer Systems & Networks

Week 7: Instruction Set Architecture, MIPS Assembly (1)

Prof Edmund Robinson, Dr Arman Khouzani

November 02, 2020

EECS, QMUL

Table of contents

Who Am I?

Education:
- BSc in EE: Sharif U of Tech, Tehran (06)
- MSc and PhD in ESE, UPenn (08, 11)

Research:
- ▷ Postdoc at OSU, USC, RHUL, QMUL
- ▷ optimisation, information theory, security

Lectureship at QMUL: since 2016
- ▶ Cloud Computing (grad), Security Engineering (final year), Data Mining (final year), CSN (year 1)

Where to find me:
- ▷ During labs (Mondays & Wednesdays)
- ▷ Teams (`eex008`)
- ▷ Email: `arman.khouzani@qmul.ac.uk`
- ▷ On-campus Tutorial sessions (Thursdays 3-4PM, Law 1.00) – through booking on QM+, limited to 15 students, masks mandatory!

Agenda/Objectives

- Understanding the concept of Instruction Set Architecture (ISA)
- Getting started with the MIPS Assembly

Recommended Reading:

- Computer Organization Design (Patterson-Hennessy), **Chapter 2, "Instructions: Language of the Computer", sections 2.1–2.6**

Recommended Watching:

- Computer Science Crash Course (on youtube) episodes #7, #8, #9

A Quick Recap (where we are)

Representation of information in Computers

Everything is Binary (because transistors!)

- ▷ Integer numbers (base 2)
- ▷ signed integer numbers (2's complement)
- ▷ characters(ASCII, UNICODE (UTF-8, UTF-16, UTF-32)). Strings are just a sequence of characters.
- ▷ image (e.g. `bitmap`: grid of pixels, each pixel represented by a number, determining its colour)
- ▷ sound (e.g. `wave`: a series of numbers indicating the intensity of time-samples of the wave.)
- ▷ video, a sequence of frames, or a sequence of differences between frames

- ▶ What about **programmes** (i.e., a sequence of instructions)?

Representation of information in Computers

Everything is Binary (because transistors!)

- ▷ Integer numbers (base 2)
- ▷ signed integer numbers (2's complement)
- ▷ characters(ASCII, UNICODE (UTF-8, UTF-16, UTF-32)). Strings are just a sequence of characters.
- ▷ image (e.g. `bitmap`: grid of pixels, each pixel represented by a number, determining its colour)
- ▷ sound (e.g. `wave`: a series of numbers indicating the intensity of time-samples of the wave.)
- ▷ video, a sequence of frames, or a sequence of differences between frames

- ▶ What about **programmes** (i.e., a sequence of instructions)? Binary too! as we will find out today.

Building a Computer: Bottom-up

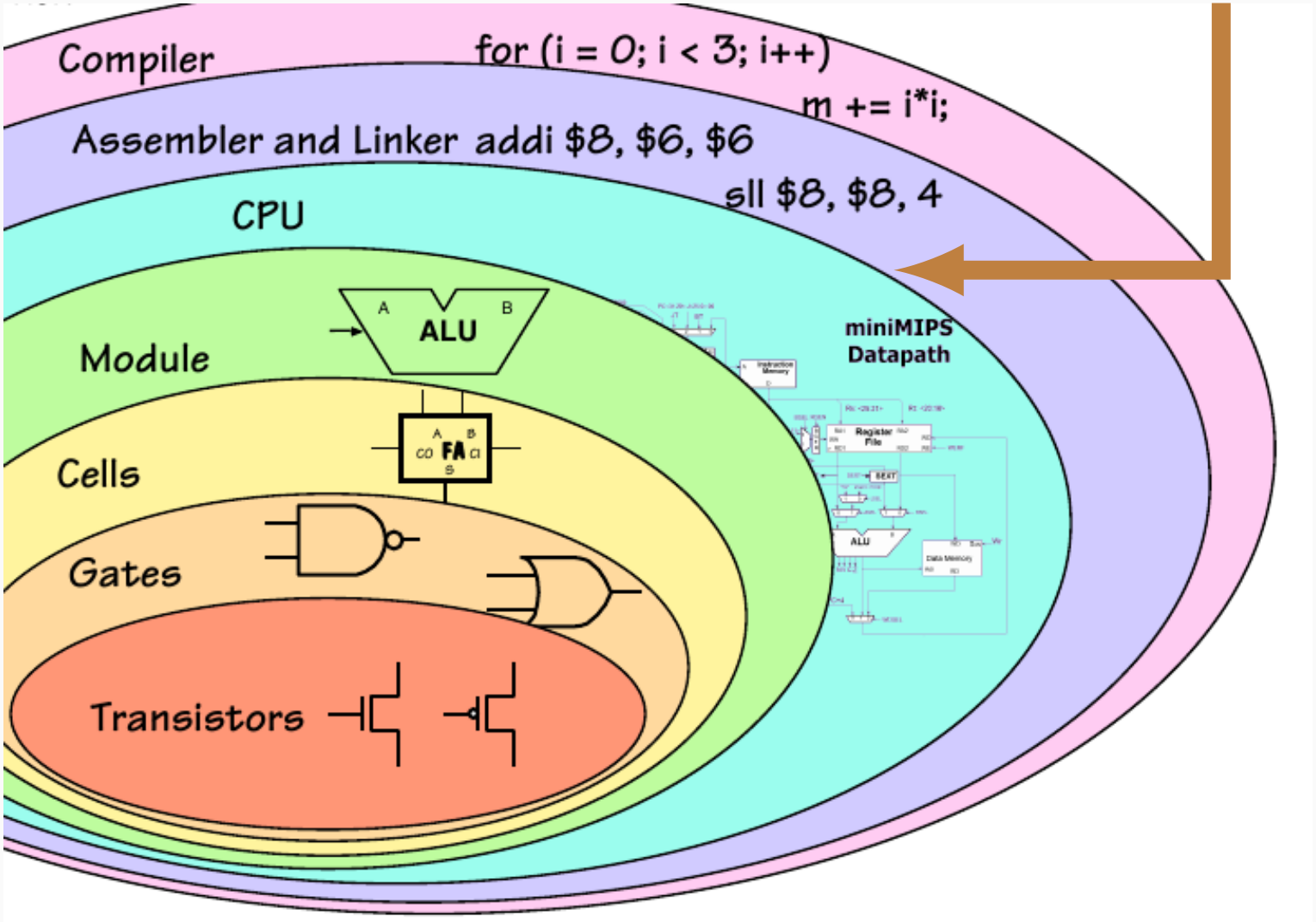
Our computers are made up of an arrangement of transistors and wires connected to electricity!

- ▷ low/high voltage $\rightarrow$ zero/one
- ▷ semiconductors + doping $\rightarrow$ a transistor (a switch, controllable by HIGH/LOW voltage)
- ▷ a careful arrangement of transistors + some wiring $\rightarrow$ logic gates (NOT, AND, OR, NAND, NOR, XOR, etc)
- ▷ a careful arrangement of these transistors + some wiring $\rightarrow$ one bit of memory (SRAM) $\rightarrow$ Registers
- ▷ a careful arrangement of the logical gates $\rightarrow$ arithmetic (adding, multiplying)
- ▷ packaging up: Arithmetic & Logic Unit, Control Unit
- ▷ and a clock pulse!

Instruction Set Architecture

Machine Codes, Instruction Set Architecture

ISA: *the interface between hardware and software*



ISA

ISA: Instruction Set Architecture

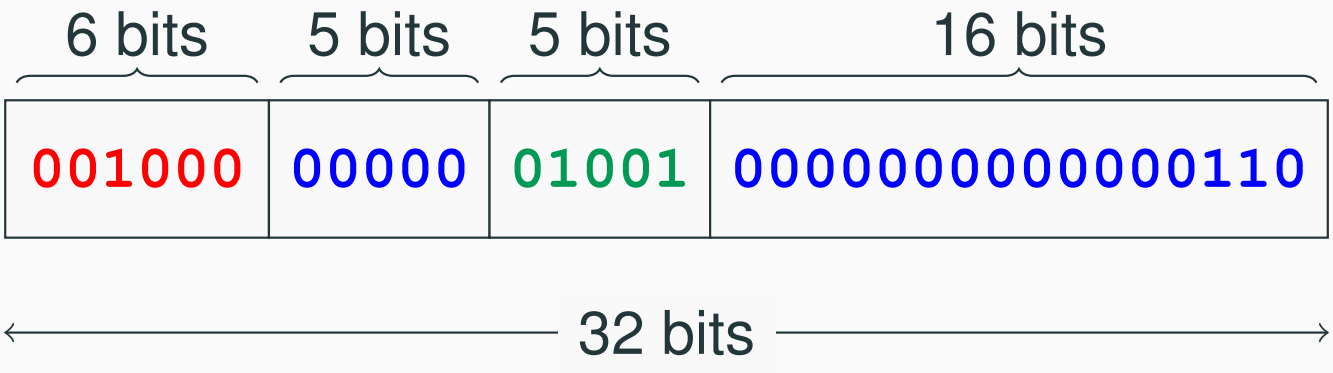
- *the interface between hardware and software*

What does this mean?

- Each instruction in ISA is implemented in the hardware (using e.g. transistors and wires);
- ISA details everything a programmer needs to know in order to write codes that can be directly executed on a processor;
- So the programmer does not need to know the detail of how the instructions are actually implemented (e.g. how the adder is designed, what kind of transistors are used, etc.) i.e., it provides an *abstract computer*.

Machine Codes, Instruction Set Architecture

- ▷ Each of these instructions has a binary encoding for that processor; These binary instructions are referred to as the **machine code** (or **machine instructions**).
- ▷ In a 32-bit processor, each machine instruction is a 32-bit binary sequence (4 Bytes), in a 64-bit processor, it is a 64-bit binary sequence.



Example of a machine instruction for a MIPS processor

Machine Codes, Instruction Set Architecture

- Since it is difficult to work with binaries **for humans**, we introduce symbols (*mnemonics*)[1] for each of them that is easier for us to remember. This constitutes the **Assembly language**, e.g.: `addi  $t1, $zero, 6`
- So note that each instruction set architecture will have a separate Assembly language. For instance, `MIPS` Assembly language will be different from `ARM` or `x86` Assembly languages.

[1] `mnemonics`:/nɪˈmɒn.ɪk/: "*something such as a very short poem or a special word used to help a person remember something*"

ISA

Typical Instructions:

- **Arithmetic**: adding, subtracting, multiplying, ...
- **Logic**: bitwise and, or, xor, nor, ...
- **Comparison**: check whether two values are equal, one is less than the other, one less than or equal to the other, ...
- **Control**: jumps, branching (which next instruction should be computed based on a condition), repetition, ...
- **Memory Access**: save the data in a register to memory, load the data from memory to a register, ...

What does a processor do?

- *Fetch*: Loads an instruction from memory;
- *Decode*:
  - Figure out what operation it should do;
  - Figure out which data to use;
- *Execute*: Carry out the computation;
- Find out which next instruction to fetch;
- Repeat the cycle!

RISC vs CISC

RISC: Reduced Instruction Set Computer ($\sim$ tens)

- ▷ Simplified design for instructions and processor;
- ▷ More instructions are needed to accomplish the same high level task;
- ▷ But each instruction will run much faster (takes one or a few actual clock cycles);
  - less time necessary to decode each instruction;
  - focuses on making these few instructions efficient;
  - leaves the smart optimisations to the compiler;
- ▷ Notable examples: ARM, Power, SPARC, MIPS

CISC Complex instruction set computer ($\sim$ hundreds)

- ▷ Notable example: the x86 family (intel, AMD)

RISC vs CISC

For a given Program of interest (with no I/O, interrupts):

$$\text{CPU execution time} = \frac{\text{\# instructions}}{\text{Program}} \times \frac{\text{\# Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

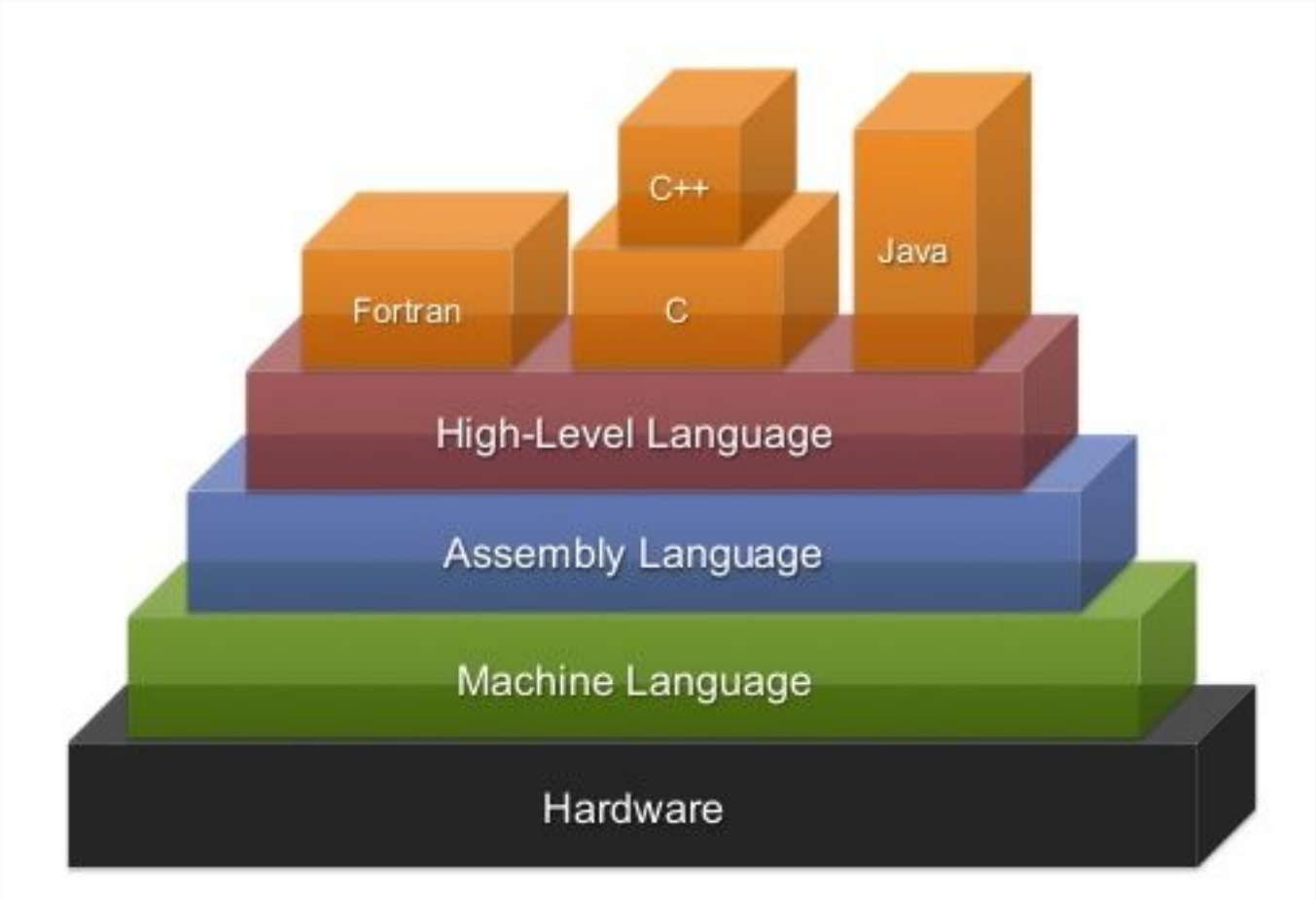
To increase efficiency (how fast programs are executed):

- RISC focuses on reducing $\frac{\text{\# Clock cycles}}{\text{Instruction}}$
- CISC focuses on reducing $\frac{\text{\# instructions}}{\text{Program}}$

Compiling

- **Compiler** Takes a (source) code written in **High Level** language like C/C++, and converts it to a low level language like (Assembly) which then the **Assembler** turns into machine level instructions (binaries) determined by a processor's ISA

Machine Codes, Instruction Set Architecture



Any high level code (Java, C++, Pascal, Ada,. . .) needs to be converted (translated) to the machine instructions of the processor it wants to run on. This task is done by a programme called **compiler**.

Compiling (the peculiar case of Java)

- Word of caution, **Java** is a bit different:
  - Java compiler turns the java (source) code into “Bytecodes” (Bytecode Executable Classes), which is in the ISA of the Java Virtual Machine (JVM).
  - The Java Runtime System then does a “Just-in-Time” compiling into the ISA of the actual processor it is running on (native code).

MIPS Processor

What is MIPS?

- **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages.
- Developed in the 80s under supervision of John Hennessy, and had a boom period in the 90s.
- Still in some use today (though losing the competition to other RISC architectures like “Power”, “ARM”)
  - Residential gateways & routers (e.g. some Cisco/Linksys)
  - Cable boxes
  - Gaming consoles (e.g. Nintendo 64, Playstation, Playstation 2, PSP)
- Good candidate for academic learning.

MIPS is a load/store architecture

MIPS is an example of **load/store** architecture:

- this means the operations (arithmetic, logic, comparison) can only be performed on **registers**; e.g.:

```
add  $t0, $s1, $s2    # t0 = s1 + s2
addi $s0, $s1, -2     # s0 = s1 - 2
```

- so the instruction can be classified into two groups:
  - Memory access (to load or store data between RAM and registers), e.g. `lw`, `sw`
  - ALU operations (add/subtract, bitwise and/or,. . .)

A **load/store** architecture is in contrast to a **register–memory** architecture, whose ISA allows operations directly on memory as well as registers (e.g. in *Intel x86*)

MIPS: word size

MIPS was originally a 32-bit architecture:

- each register holds 32 bits;
- each instruction is 32 bits long;
- its ALU takes two 32-bit long inputs (the ALU operations are done on groups of 32 bits at a time).
- This parameter so important we have given it a name: a “**word**” is the size of data that the ALU operates on.

Although a 64-bit MIPS is also developed, we only focus on the 32-bit architecture.

MIPS: integer registers

MIPS has 32 integer registers, each holding 32 bits. The ones we typically use are the following:

- `$zero`: Constant value of zeros (cannot be overwritten!)
- `$t0` $\cdots$ `$t9`: general purpose (temporary registers)
- `$s0` $\cdots$ `$s7`: general purpose (save registers)

MIPS: registers

Reg. No.	Reg. Name	purpose (usage)
$0	`$zero`	always holds 32 bits of zeros (no overwrite)
$1	`$at`	Reserved to be used by the assembler
$2 · · · $3	`$v0`, `$v1`	Return values from subroutines
$4 · · · $7	`$a0` · · · `$a3`	Arguments to subroutines
$8 · · · $15	`$t0` · · · `$t7`	General purpose (temporary registers)
$16 · · · $23	`$s0` · · · `$s7`	General purpose (save registers)
$24 · · · $25	`$t8` · · · `$t9`	More temporary registers
$26 · · · $27	`$k0` · · · `$k1`	Reserved for Kernel (Operating System)
$28	`$gp`	Global Area Pointer
$29	`$sp`	Stack Pointer
$30	`$fp`	Frame Pointer
$31	`$ra`	Return Address

MIPS instruction set architecture: Registers

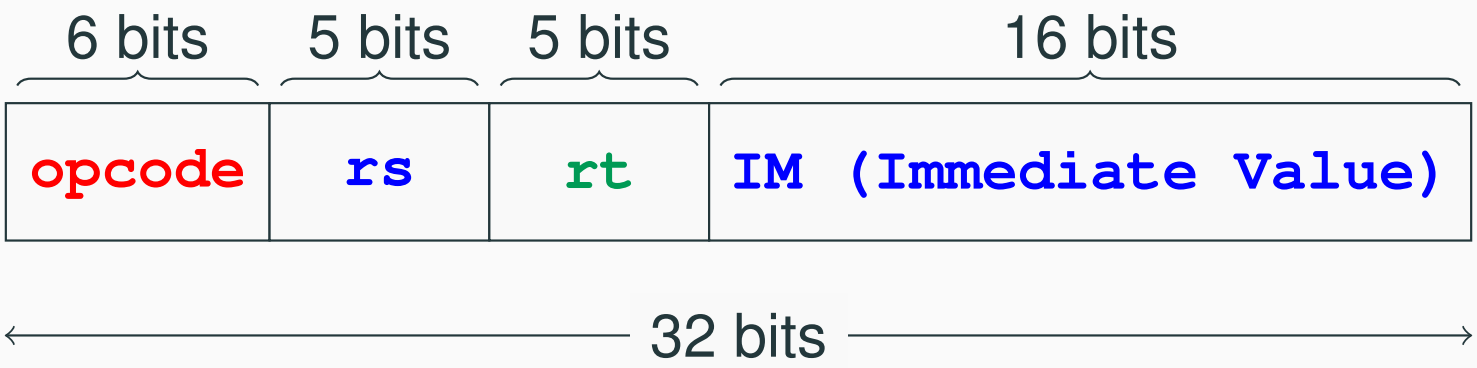
- Besides these general purpose registers, there are a few registers that are crucial for the working of the processor itself. Most notably:[2]
  - ▷ **`PC`**: *Program Counter*, a (32-bit) register: contains the (memory) address of the next instruction to be fetched from memory;
  - ▷ **`IR`**: *Instruction Register*, a (32-bit) register: contains the most recently fetched instruction.

[2] The programmer does not have direct access to these registers.

MIPS instructions

- All MIPS instructions (machine codes) are **32 bits** long (i.e., a “word” length).
- They come in three formats (layouts):
  - the **I**-type or I-format (**I** for **I**mmediate)
  - the **R**-type or R-format (R for Register)
  - the **J**-type or J-format (J for Jump)

MIPS instructions: I-format



I-format

opcode : specifies the operation to be performed;

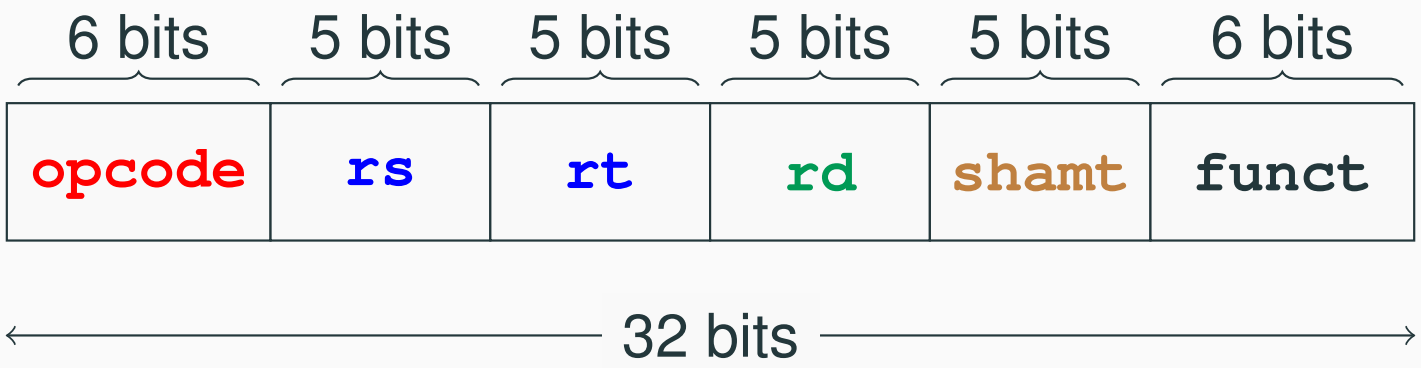
rs : specifies the register that hold one of the operands (what the operation will perform on);

rt : specifies the register to put the result in;

IM : specifies the other operand as an immediate value (a 16 bit constant).

apply the operation specified by **opcode** on the content of reg. **rs** & the **immediate** value, put the result in reg. **rt**.

MIPS instructions: R-format



R-format

opcode : specifies the operation to be performed;

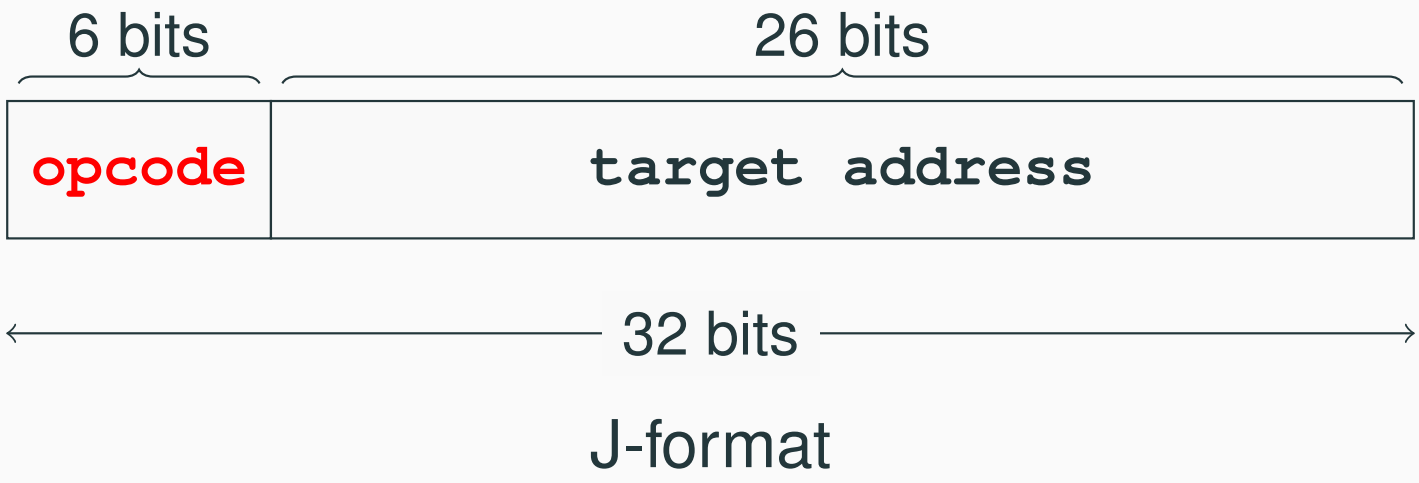
rs, **rt** : specify the two registers that hold the operand (what the operation is performed on);

rd : specifies the register to put the result in;

shamt : shift amount (for the shift instructions);

funct : specifies which variation of the **opcode**, when multiple operations share the same opcode.

MIPS instructions: J-format



J-format

`opcode` : the opcode for jump;

`target address` : the address of the instruction (in the memory) to jump to (to fetch next)

MIPS Assembly

MIPS Assembly Instructions

- ▷ Since it is hard for us (humans) to follow binary numbers, we also have **mnemonics** to represent the machine codes.
- ▷ these constitute the Assembly language (so we have many Assembly languages, one for each ISA!)

The way we write a typical MIPS Assembly Instruction:

`OPERATION DESTINATION, OPERAND1, OPERAND2`

which means:

$\texttt{DESTINATION} \leftarrow \texttt{OPERATION}(\texttt{OPERAND1}, \texttt{OPERAND2})$

- ▷ so pay attention, that we write the destination first (where to save the result of operation);
- ▷ with one exception (which we will see later today!)

Arithmetic Operations: Adding

- `add <destination> <source 1> <source 2>`

  examples:

```
add $t0, $s1, $s2   # t0 = s1 + s2
add $s1, $s1, $s2   # s1 = s1 + s2
```

- `addi <destination> <source 1> <value>`

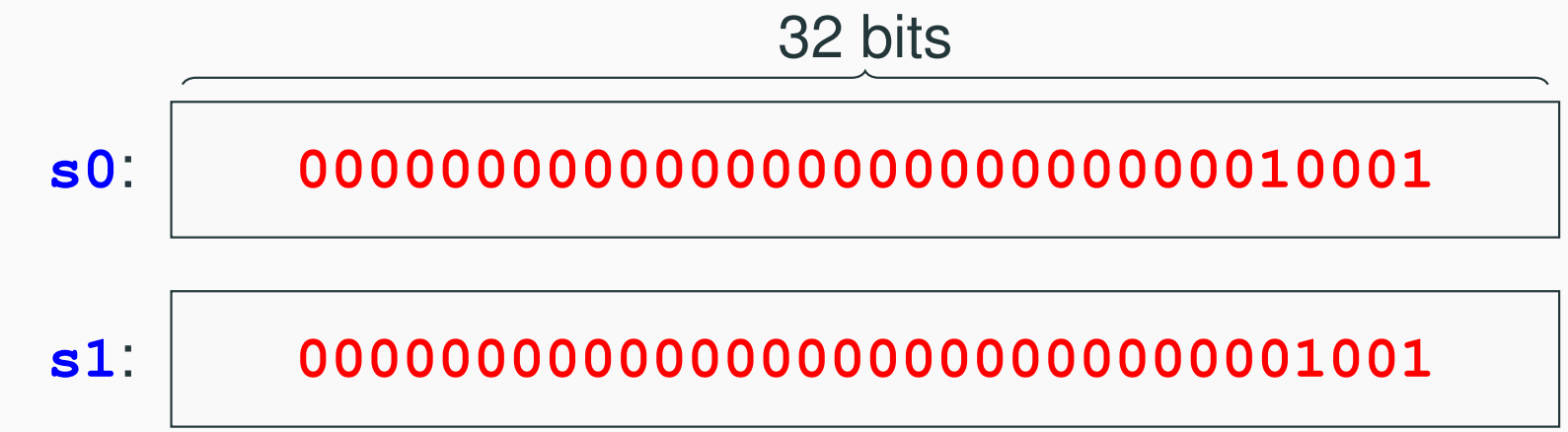
  examples:

```
addi $s0, $s1, -2    # s0 = s1 - 2
addi $s1, $s1, 1     # s1++
addi $t1, $zero, 6   # t1 = 6
```

  - the immediate value can be positive or negative

Arithmetic Operations: Adding, example

Example: suppose the contents of the registers **s0** and **s1** is as follows:

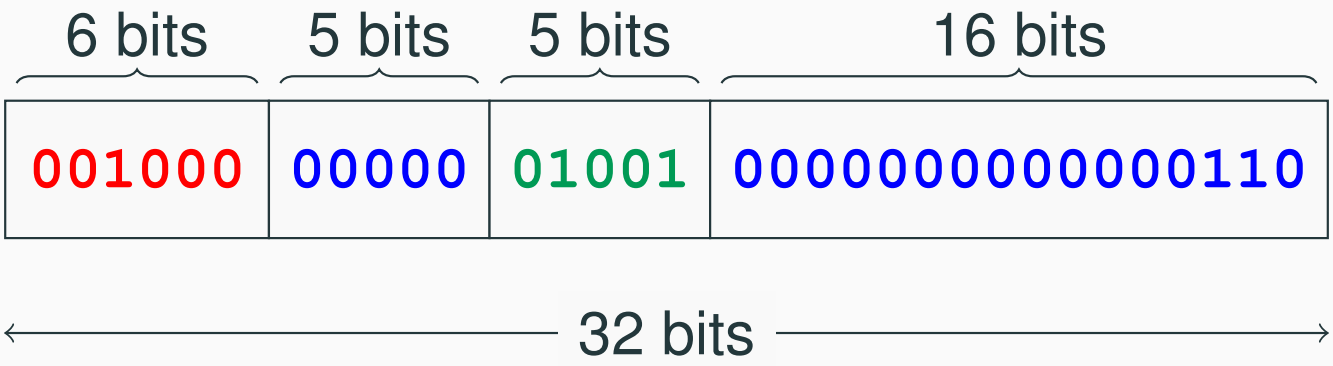


Then, upon execution of: "**add $t0, $s0, $s1**"
the content of register **t0** will be:

Arithmetic Operations: Adding

Example: converting a MIPS assembly instruction to a MIPS instruction (machine code):

▷ `addi $t1, $zero, 6` *# t1 = 6*



The opcode for `addi` is 8, which in 6-bits binary is `001000`, the number for register `$zero` is 0, which in 5-bits binary is `00000`, the number for register `$t1` is 9, which in 5-bits binary is `01001`, and finally, the immediate value 6 in 16-bits binary is `0000000000000110`.

MIPS Instructions: some observations

- Question: in the binary encoding of the MIPS instructions (the R and I formats), why do we need 5 bits to designate a register?

MIPS Instructions: some observations

- Question: in the binary encoding of the MIPS instructions (the R and I formats), why do we need 5 bits to designate a register?
- ▷ Answer: Because we have 32 different registers, and with 5 bits, you can identify $2^5 = 32$ different items (so, if we had a microprocessor that had only 16 register, 4 bits would have sufficed to designate a register, and so on.)

MIPS Instructions: some observations

- Question: in the binary encoding of the MIPS instructions (the R and I formats), why do we need 5 bits to designate a register?

MIPS Instructions: some observations

- Question: in the binary encoding of the MIPS instructions (the R and I formats), why do we need 5 bits to designate a register?
- ▷ Answer: Because we have 32 different registers, and with 5 bits, you can identify $2^5 = 32$ different items (so, if we had a microprocessor that had only 16 register, 4 bits would have sufficed to designate a register, and so on.)

MIPS Instructions: some observations

- Question: what is the largest and smallest number that we can pass as an immediate value to **`addi`** instruction? (recall that the immediate value is signed, represented in 2's complement format.)

MIPS Instructions: some observations

- Question: what is the largest and smallest number that we can pass as an immediate value to **`addi`** instruction? (recall that the immediate value is signed, represented in 2's complement format.)
- Answer:
  - smallest number in 16 bits 2's complement: 1000 0000 0000 0000, which is -2^{15} in decimal
  - largest number in 16 bits 2's complement: 0111 1111 1111 1111, which is $2^{15} - 1$ in decimal

MIPS Instructions: some observations

- Question: in the **`addi`** instruction, the immediate value is 16 bits long, but our ALU takes two 32-bit long inputs. One of the operands is the content of a register, so it is 32 bits long already. How is the 16-bit immediate value converted to a 32-bit value, so that the number it represents does not change?

MIPS Instructions: some observations

- Question: in the **`addi`** instruction, the immediate value is 16 bits long, but our ALU takes two 32-bit long inputs. One of the operands is the content of a register, so it is 32 bits long already. How is the 16-bit immediate value converted to a 32-bit value, so that the number it represents does not change?
- Answer: it is **sign-extended** to 32 bits: the leftmost bit (the MSB) is replicated 16 times to the left: if the immediate value is positive, the 16th bit is zero, and adding 16 more zeros behind it does not change its value. If the immediate value is negative, the 16th bit is one, and adding 16 more ones behind it does not change its value (another beauty of 2's complement!)

Arithmetic Operations: Subtracting

- **sub <destination> <source 1> <source 2>**
  example:

```
sub $t0, $s1, $s2   # t0 = s1 - s2
sub $s1, $s1, $s2   # s1 = s1 - s2
```

  - note that the order of the two operands (**<source 1>** and **<source 2>**) is important.
  - there is no **subi** instruction, but you wouldn't need it, because **addi** can take negative numbers as well.

Bitwise Logic Instructions

Some of the logic instructions in MIPS:

`and` : bitwise AND

`or` : bitwise OR

`nor` : bitwise NOR (bitwise OR, negated)

`xor` : bitwise EXCLUSIVE OR

- Note that these are R-type instructions, e.g.:

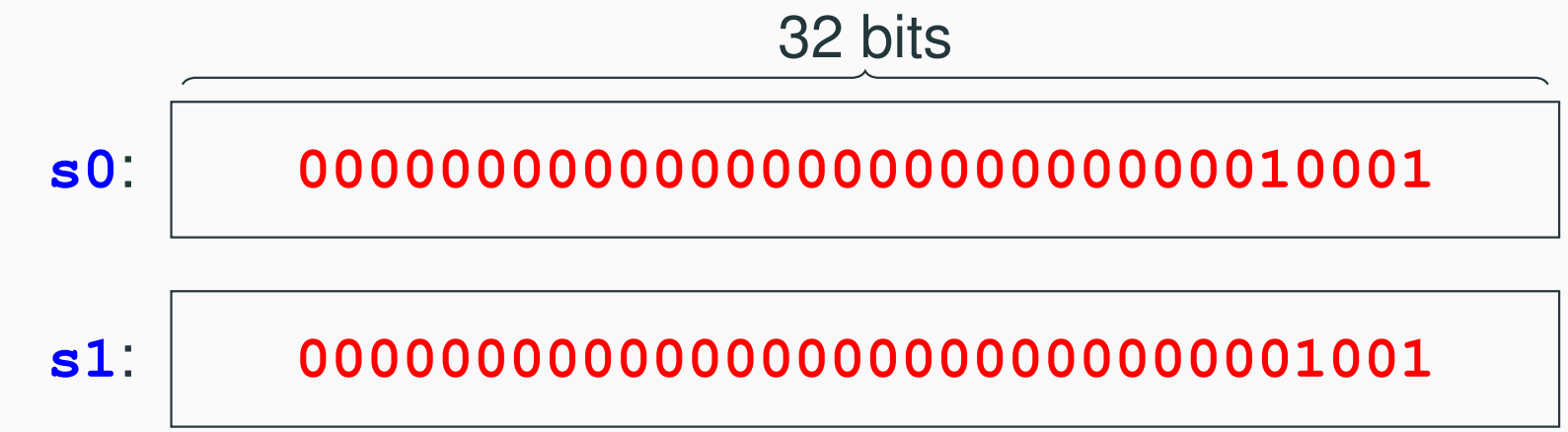
  ```
  and $t0, $s1, $s2
  ```

  means:

  the ALU takes the values in registers **`$s1`** and **`$s2`** (each 32 bits long); bitwise AND them; and puts the result in **`$t0`**, i.e., the first bit in register **`$t0`** will be the AND of the first bit in register **`$s1`** and the first bit in register **`$s2`**, and so on and so forth.

Bitwise Logic Operations: example

Example: suppose the contents of the registers **s0** and **s1** is as follows:



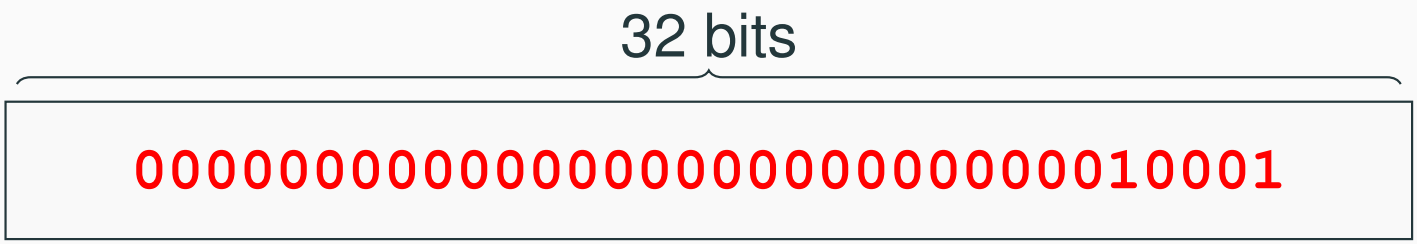
Then, upon execution of: "`xor $t0, $s0, $s1`"
the content of register **t0** will be:

Bitwise Logic Instructions: Immediate version

- Each of these R-format instruction have a corresponding I-format as well: **`andi`**, **`ori`**, **`nori`**, **`xori`**: the operation is performed (again, bitwise) between the content of a register and the binary representation of the immediate value.
- The 16-bit immediate value is **Zero-Extended** to get a 32 bits long operand.

Bitwise Logic Instructions: Immediate version

Example: suppose the content of the register **`s0`** is:



Then, the content of register **`s0`** upon execution of: "**`xori $s0, $s0, 3`**", will be:

(that is, the last two bits of **`$s0`** gets flipped.)