Machine Language

Of course, the hardware doesn't really execute MIPS assembly language code.

The hardware can only store bits, and so the instructions it executes must be expressed in a suitable binary format.

We call the language made up of those instructions the *machine language*.

Different families of processors typically support different machine languages.

In the beginning, all programming was done in machine language… very ugly…

Assembly languages were created to make the programming process more human-centric.

Assembly language code is translated into machine language by an *assembler*.

Alas, there is no universal assembly language. In practice, assembly languages are tightly coupled with the underlying machine language and hardware.

Assembly Language vs. Machine Language

Assembly provides convenient symbolic representation

- much easier than writing down numbers
- e.g., destination first

Machine language is the underlying reality

- e.g., destination is no longer first

Assembly can provide 'pseudoinstructions'

- e.g., "`move $t0, $t1`" exists only as an extension to assembly
- would be translated to "`add $t0,$t1,$zero`" by the assembler

When considering performance you should count real instructions

Classifying the Assembly Instructions

Examining the (basic) MIPS assembly instructions, we can easily identify three fundamentally different categories, according to the parameters they take:

- instructions that take 3 registers

```
add   $s0, $s1, $s2
or    $t1, $t0, $t1
```

- instructions that take 2 registers and an immediate (offset, number, address, etc.)

```
addi  $t7, $s5, 42
lw    $s3, 12($t4)
beq   $t1, $t3, Label01
```

- instructions that take an immediate

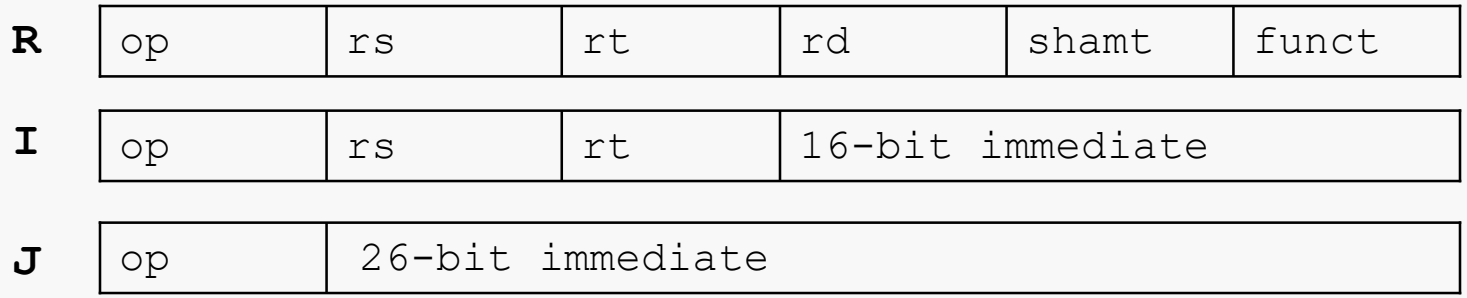
```
j     Label01
```

Overview of MIPS Machine Language

Simple instructions, all 32 bits wide

Very structured, no unnecessary baggage

Only three* instruction formats with strictly-defined fields:



R-format: basic arithmetical-logical instructions
I-format load/store/conditional branch instructions
J-format: jump/unconditional branch instructions

* Well… not really…

Overview of MIPS Machine Language

Design principles

1. Simplicity favors regularity
2. Smaller is faster
3. Make the common case fast
4. Good design demands good compromises

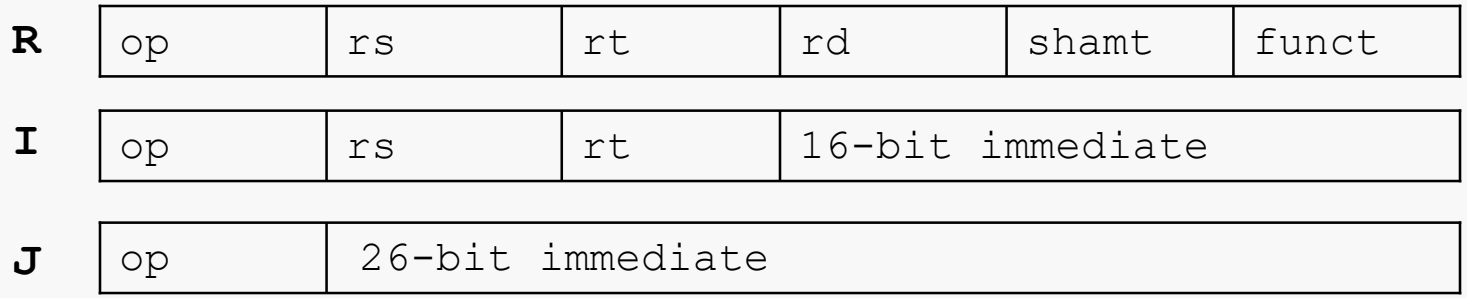
Overview of MIPS Machine Language

In MIPS32 Release 2, there are over 200 basic MIPS machine instructions.

In order to specify that many different instructions, we could use a field of 7 or more bits in every machine instruction.

But MIPS machine language uses a 6-bit *opcode* field and a variety of special cases.

For R-format instructions, the opcode field is set to `000000`, and the last 6 bits specify exactly which arithmetic/logical instruction is to be performed.



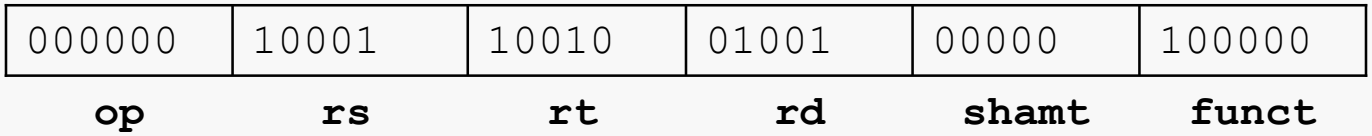
Arithmetic/Logical Instructions

Instructions, like registers and words of data, are also 32 bits long

Example: `add $t1, $s1, $s2`

registers have numbers, `$t1` = 9, `$s1` = 17, `$s2` =18

Machine language basic arithmetic/logic instruction format:

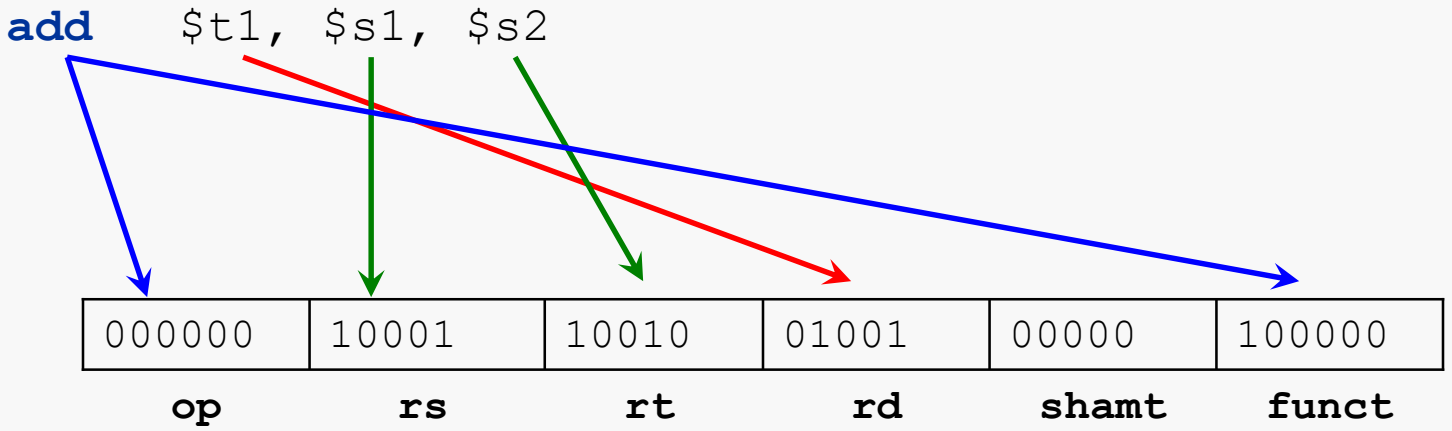


Can you guess what the field names stand for?

op	**operation code (opcode)**
rs	**1st source register**
rt	**2nd source register**
rd	**destination register**
shamt	**shift amount**
funct	**opcode variant selector**

Mapping Assembly to Machine Language

Note how the assembly instruction maps into the machine representation:



The three register fields are each 5 bits wide. Why?

For arithmetic-logical instructions, both the **op** field and the **funct** field are used to specify the particular operation that is to be performed.

If you view memory contents, this would appear as 0x02324820.

Load Instructions

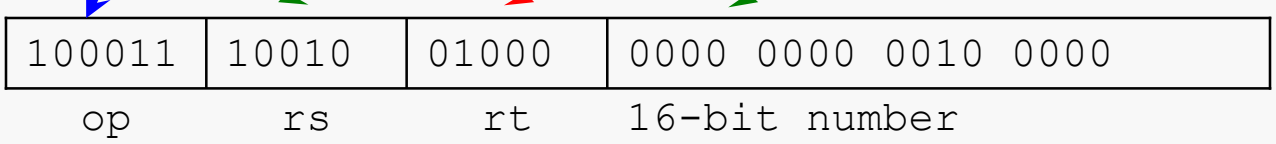
Consider the load-word and store-word instructions:

- what would the regularity principle have us do?
- new principle: Good design demands a compromise

We need a different type of machine language instruction format for these:

- I-type for data transfer instructions
- other format was R-type for register

Example: `lw $t0, 32($s2)`



Where's the compromise?

Jump Instructions

Consider the jump instruction: `j Label01`

We need a different type of machine language instruction format for this as well.

Example: `j exit # assume exit is a statement label`

Anticipating Execution

What will be involved in executing a machine language instruction?

Consider an I-type instruction, say a **`lw`** instruction: