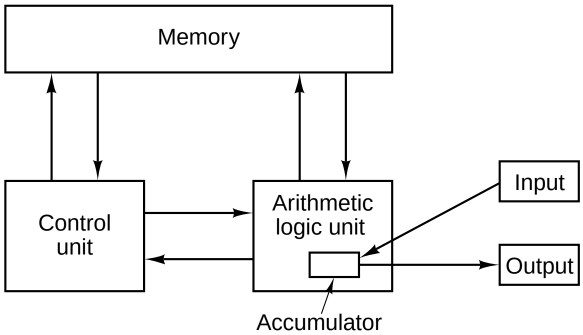
Slide 1



Key ideas :-

- Programs and data are stored in memory (More on this later).

Slide 2

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Fetch any data from memory
4. Execute the instruction.
5. Work out the address of the next instruction
6. Go back to step 1.

Questions

- How do you represent instructions?
- How do you represent data?

- Think of assembly language as a very low level programming language.
- There are very few instructions.
- Many things that you can do in one step in high level languages you have to do in many steps.

Slide 3

Why learn Assembly Language?

- It is the language of the machine. Computers don't understand C or Java directly.
- We'll see how we can implement assembly language.
- It helps you to understand how compilers work.

Slide 4

- The MIPS processor has 32 special variables called registers. These registers can hold 32 bits (4 Bytes).
- Some of the registers have special uses. We will find out as we go along.
- The registers have the names \$0-\$31, they also have other names.
- In the next few lectures we will be concerned with the following registers (the meaning of saved and temporary will become clear later) :

Name	Number	Usage
`$t0-$t7`	8-15	Temporaries
`$s0-$s7`	16-23	Saved
`$t8-$t9`	24-25	more temporaries

Pseudo C code

```
$s0 = $s5 + $t0
```

Assembly language:

Slide 5

```
add $s0,$s5,$t0
```

Arithmetic instructions have three arguments the first two must be registers and the last is a register or a small constant (more later).

Arithmetic instructions, first argument is the destination.

Important Arithmetic instructions can only have 3 arguments.

Pseudo C code

```
$s0 = $s1 + $s2 + $s4 + 2*$s5
```

Assembly language:

Slide 6

```
add $t0,$s1,$s2
add $t0,$s4,$t0
add $t1,$s5,$s5
add $s0,$t0,$t1
```

The add instruction does not get confused if the destination register is the same as a source register.

- What about constants?
- How do we do do things like `$t0 = $t0 + 1`?

We can't just magic the values into registers we have to load values in there.

Slide 7

The last operand of an `add` instruction can be a small constant (a 16bit number). The new form of the `instruction` is called `addi`, the `i` stands for immediate.

- `addi $t0,$t0,1`

When you are writing your assembly language programs, a $ means that there is a register.

While the assembler can often guess what you mean it is better to write what you mean.

Slide 8

The above instruction could be rewritten as

- `addi $8,$8,1`

If you wrote

- `add 8,8,1`

The assembler would have a hard job of guessing what you mean.

Slide 9

- How do we put a value in a register?
- The MIPS processor has a special register, number 0, which is hardwired to be the value 0. No matter what you do to that register it stays at that value.
- This register is called `$0` or `$zero` .
- How do I set `$s0` to be 34?
- `add $s0,$zero,34`

Slide 10

There is no direct way of loading large constants into a register. It must be done in two steps.

For example to load the value `0x0fff0123` into the register `$s0` we have to do the following:

- `lui $s0,0xfff0` This places the value `0xfff00000` into the register `$s0`. `lui` stands for load upper immediate.
- `add $s0,$s0,0x0123`

If you are not completely happy with hexadecimal (base 16) numbers, revise them now!.

Slide 11

The assembler provides a number of pseudo instructions, that is instructions that look like atomic instructions that get turned into a sequence of instructions.

One of them is `li` which allows you to load large constants into registers.

The instructions on the previous slide can be abbreviated to

- `li $s0,0xfff00123`

Slide 12

- `sub` and `subi` same format as `add`.
- `mul` multiply.

Slide 13

- 32 registers, each can hold 32 bit integers.
- register `$0` is fixed at the value 0.
- Arithmetic instructions, very limited format, all ways three arguments. Destination is the first register.