Slide 1

- Labels
- Making Decisions `beq` and `bne`
- Jumps.
- Some example programs

Slide 2

- Programs and Data live in Memory.
- The processor does not understand strings of characters such as `add` for instructions.
- Each instruction has a unique number. For example the in instruction

```
lw $4, 0($29)
```

  has the code `0x8fa40000`.
- It does not concern us at the moment how instructions get turned into numbers, all that matters is that it happens. Later it will become more important how the coding works.
- Every instruction has an address.

Slide 3

- Programs and data are in the same memory. The processor just fetches numbers from memory executes instructions. Almost every combination of bits is an instruction. Thus you could ask the machine to execute a sound file with unpredictable consequences.
- Also, the machine has no way of knowing if a value in memory is data or an instruction. So if you do a `lw` or a `sw` you better know what your reading or writing.
- All this has a positive side, you can write programs that write other programs (Can you think of any?).

Your assembly language programs will fail in very strange ways, you better get used to understanding the code you write.

Slide 4

- Every instruction has an address.
- Sometimes you need to know the address of an instruction.
- Every instruction takes up exactly 4 bytes of memory (on the MIPS), it is possible in theory to work out the address of any instruction if you know the address of the first instruction.

Working out the address of an instruction can be tedious, luckily there is a program called an assembler which works this out.

Slide 5

```
 .data
n1:     .word 10
        .text
        .globl main

main:   la $t0,n1
        lw $s0,0($t0)
        addi $s0,$s0,1
        sw $s0,0($t0)
        jr $31
```

`main` is the address of the first instruction. `n1` is the address of a piece of data.

Slide 6

All the programs we have looked at so far have been linear. We need a way of doing different things depending on the values of registers.

The MIPS processor provides two instructions for making decisions:

- `beq` Branch if Equal
- `bne` Branch if not equal

General format:

- `beq $register1,$register2,Label`

Slide 7

- `beq $register1,$register2,Label` If \$regisiter1 is equal to \$regisiter2 then goto `Label` otherwise execute the next instruction.
- `bne $register1,$register2,Label` If \$regisiter1 is not equal to \$regisiter2 then goto `Label` otherwise execute the next instruction.

Slide 8

Pseudo C code:

```
 if $s1 == $s2 then $s3 = 0 ;
```

Assembly language:

```
      bne $s1,$s2,skip
      add $s3,$0,$0
skip:  Next Instruction
```

Slide 9

Pseudo C code:

```
if $s1 = $s2 then $s3 = 0 else $s3 = $s3 + 1 ;
```

Assembly code.

```
          beq $s1,$s2,set_zero
          addi $s3,$s3,1
          j skip2
set_zero: add $s3,$0,$0
skip2:    Next Instruction
```

Slide 10

On the previous slide we met the jump instruction. This is like an unconditional branch.

- `j label` make the next instruction the label.

Slide 11

There is often more than one way of writing the same piece of code.

For example:

```
      bne $s1,$s2,skip
      add $s3,$0,$0
skip:  Next Instruction
```

Can also be written as:

```
                beq $s1,$s2,settozero
                j skip
settozero:      add $s3,$0,$0
skip:           Next Instruction
```

Although at this stage apart from the number of instructions there does not seem to be much difference between the two pieces of code. We will later discover that for efficiency reasons it is better to avoid too many jumps.

Slide 12

Pseudo C code

```
for($t0 = 0; $t0 != 10 ; $t0 = $t0 +1 ) {
  A[$t0] = 0
}
```

Assume that the base of the integer array A is stored in `$s0`.

```
       addi $t0,$0,0
loop:  addi $t1,$0,10
       beq $t0,$t1,exit
       add $t2,$t0,$t0
       add $t2,$t2,$t2  # $t2 = $t0*4
       add  $t2,$s0,$t2  # $t2 = address of A[$t0]
       sw $0,0($t2)
       addi $t0,$t0,1
       j loop
exit:
```

Look at the loop on the previous slide. How can we make it more efficient?

- We don't have to load 10 in `$t1` each time around the loop because `$t1` does not change (this is called a loop invariant).

Slide 13

- We can do the test to exit the loop at the end of the loop, because we know the loop executes at least once.
- Instead of multiplying by 4 each time around the loop we can add 4 to `$t0` each time and exit when `$t0` is equal to 40.

Exercise rewrite the above code.

- So far we have only compare if registers are equal or different.
- The MIPS provides no branch on less than.
- Instead there is the `slt` instruction.

Slide 14

General format:

- `slt $Rdest,$Rsrc1,$RSrc2`

Set Register Rdest to 1 if register `Rsrc` is less than `Rsrc2`, set to zero otherwise.

Slide 15

Pseudo C code:

```
if $s1 < $s2 then $s3 = 0
```

Assembly:

```
     slt $t0,$s1,$s2
     beq $t0,$zero,skip
     add $s3,$zero,$zero
skip:
```