Chapter 3

Instructions:

- **Language of the Machine**
- **More primitive than higher level languages**
  e.g., no sophisticated control flow
- **Very restrictive**
  e.g., MIPS Arithmetic Instructions

- **We'll be working with the MIPS instruction set architecture**
  - **similar to other architectures developed since the 1980's**
  - **used by NEC, Nintendo, Silicon Graphics, Sony**

Design goals: maximize performance and minimize cost, reduce design time

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

- Design Principle: simplicity favors regularity. Why?
- Of course this complicates some things...

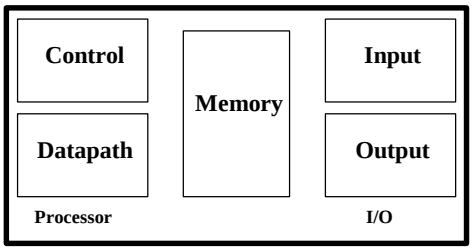
C code:
```
A = B + C + D;
E = F - A;
```

MIPS code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

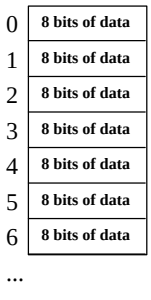
Registers vs. Memory

- **Arithmetic instructions operands must be registers,**
  — only 32 registers provided
- **Compiler associates variables with registers**
- **What about programs with lots of variables**



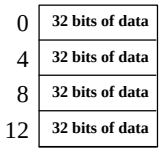
Memory Organization

- **Viewed as a large, single-dimension array, with an address.**
- **A memory address is an index into the array**
- **"Byte addressing" means that the index points to a byte of memory.**



Memory Organization

- **Bytes are nice, but most data items use larger "words"**
- **For MIPS, a word is 32 bits or 4 bytes.**



Registers hold 32 bits of data

- **2^{32} bytes with byte addresses from 0 to 2^{32}-1**
- **2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4**
- **Words are aligned**
  i.e., what are the least 2 significant bits of a word address?

Instructions

- **Load and store instructions**
- **Example:**

C code: `A[8] = h + A[8];`

MIPS code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

- **Store word has destination last**
- **Remember arithmetic operands are registers, not memory!**

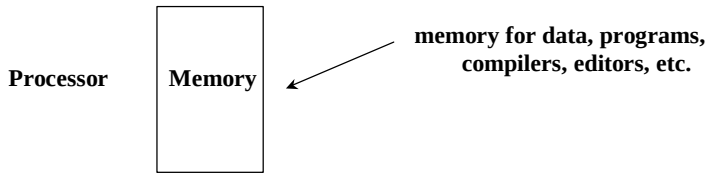
So far we've learned:

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  - — to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the "next" instruction and continue

Control

- **Decision making instructions**
  - **alter the control flow,**
  - **i.e., change the "next" instruction to be executed**
- **MIPS conditional branch instructions:**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- **Example:** **if (i==j) h = i + j;**

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```


Control

- **MIPS unconditional branch instructions:**

```
j  label
```

- **Example:**

```
if (i!=j)           beq $s4, $s5, Lab1
    h=i+j;          add $s3, $s4, $s5
else                j Lab2
    h=i-j;          Lab1:  sub $s3, $s4, $s5
                    Lab2:  ...
```

- ***Can you build a simple for loop?***

Control Flow

- **We have: beq, bne, what about Branch-if-less-than?**
- **New instruction:**

```
                                  if  $s1 < $s2 then
                                     $t0 = 1
      slt $t0, $s1, $s2           else
                                     $t0 = 0
```

- **Can use this instruction to build "`blt $s1, $s2, Label`"**
  — can now build general control structures
- **Note that the assembler needs a register to do this,**
  — there are policy of use conventions for registers

Policy of Use Conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

- How do we make this work?

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 in Assembly
  - would be implemented using "add $t0,$t1,$zero"
- When considering performance you should count real instructions