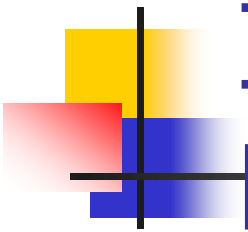
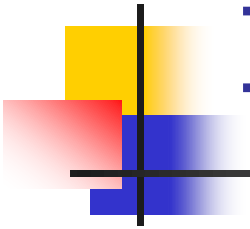


COD Ch. 3 Instructions: Language of the Machine



Instructions: Overview

- Language of the machine
- More primitive than higher level languages, e.g., no sophisticated control flow such as *while* or *for* loops
- Very restrictive
  - e.g., MIPS arithmetic instructions
- We'll be working with the MIPS instruction set architecture
  - inspired most architectures developed since the 80's
  - used by NEC, Nintendo, Silicon Graphics, Sony
  - the name is just not related to *millions of instructions per second* !
  - it stands for **m***icrocomputer without* **i***nterlocked* **p***ipeline* **s***tages* !
- Design goals: *maximize performance* and *minimize cost* and *reduce design time*

MIPS Arithmetic

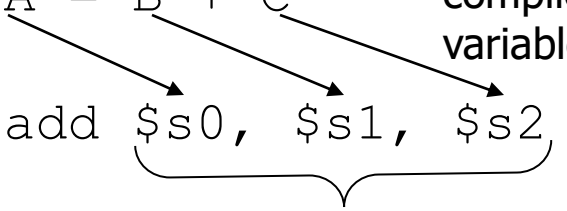
- All MIPS arithmetic instructions have 3 operands
- Operand order is fixed (e.g., destination first)

- *Example*:

C code: `A = B + C`

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

compiler's job to associate variables with registers



MIPS Arithmetic

- Design Principle 1: *simplicity favors regularity*.
  Translation: *Regular instructions make for simple hardware*!
- *Simpler hardware reduces design time and manufacturing cost.*
- Of course this complicates some things...

C code:

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

MIPS code (arithmetic):

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

Allowing variable number of operands would simplify the assembly code but complicate the hardware.

- Performance penalty: high-level code translates to denser machine code.

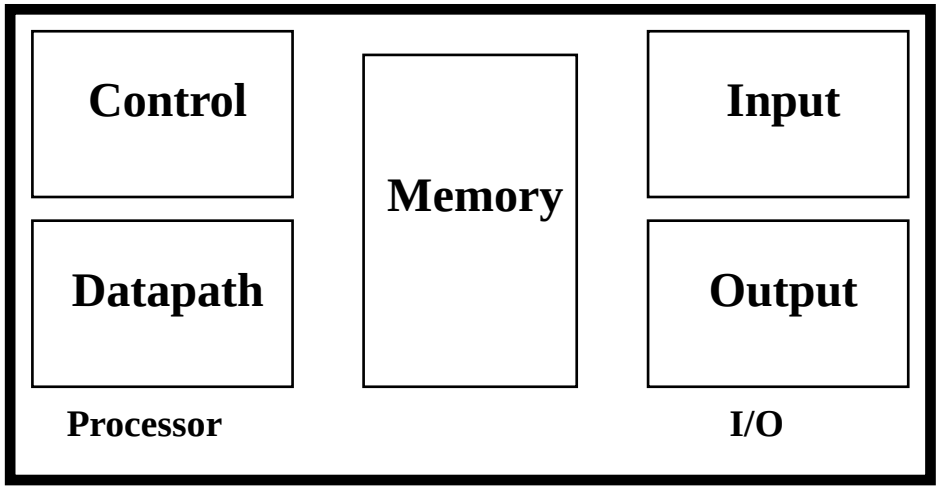
MIPS Arithmetic

- *Operands must be in registers* – only 32 registers provided (which require 5 bits to select one register). Reason for small number of registers:

- Design Principle 2: *smaller is faster*. *Why?*
  - *Electronic signals have to travel further on a physically larger chip increasing clock cycle time.*
  - *Smaller is also cheaper!*

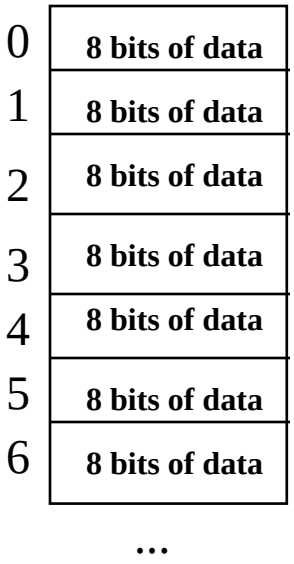
Registers vs. Memory

- Arithmetic instructions operands must be in registers
  - MIPS has 32 registers
- Compiler associates variables with registers
- What about programs with lots of variables (arrays, etc.)? Use *memory*, *load/store* operations to transfer data from memory to register – if not enough registers *spill registers* to memory
- *MIPS is a load/store architecture*



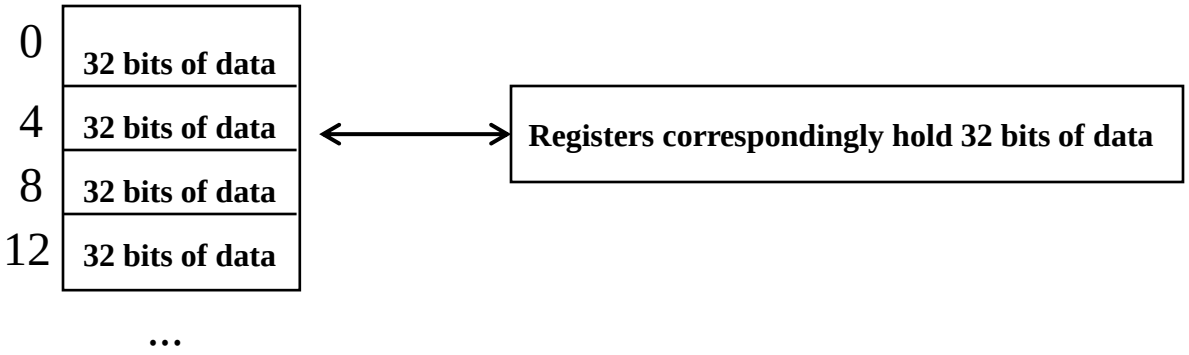
Memory Organization

- Viewed as a large single-dimension array with access by *address*
- A memory address is an *index* into the memory array
- *Byte addressing* means that the index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte



Memory Organization

- Bytes are load/store units, but most data items use larger *words*
- For MIPS, a word is 32 bits or 4 bytes.

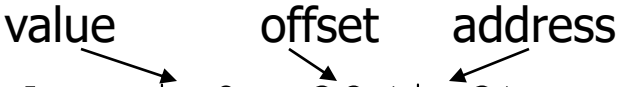


- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
  - i.e., words are *aligned*
  - *what are the least 2 significant bits of a word address?*

Load/Store Instructions

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

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




```
MIPS code   (load):   lw  $t0, 32($s3)
        (arithmetic):  add $t0, $s2, $t0
             (store):  sw  $t0, 32($s3)
```

- Load word has destination first, store has destination last
- Remember MIPS arithmetic operands are registers, not memory locations
  - therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory