MIPS History

- MIPS is a computer family
  - R2000/R3000 (32-bit); R4000/4400 (64-bit); R10000 (64-bit) etc.
- MIPS originated as a Stanford research project under the direction of John Hennessy
  - *M*icroprocessor without *I*nterlocked *P*ipe *S*tages
- MIPS Co. bought by SGI
- MIPS used in previous generations of DEC (now Compaq) workstations
- Now MIPS Technologies is the embedded systems market
- **MIPS is a RISC**

ISA MIPS Registers

- Thirty-two 32-bit registers \$0,\$1,…,\$31 used for
  - integer arithmetic; address calculation; temporaries; special-purpose functions (stack pointer etc.)
- A 32-bit Program Counter (PC)
- Two 32-bit registers (HI, LO) used for mult. and division
- Thirty-two 32-bit registers \$f0, \$f1,…,\$f31 used for floating-point arithmetic
  - Often used in pairs: 16 64-bit registers
- Registers are a major part of the "state" of a process

MIPS Register names and conventions

Register	Name	Function	Comment
$0	Zero	Always 0	No-op on write
$1	$at	Reserved for assembler	Don’t use it
$2-3	$v0-v1	Expr. Eval/funct. Return	
$4-7	$a0-a3	Proc./func. Call parameters	
$8-15	$t0-t7	Temporaries; volatile	Not saved on proc. Calls
$16-23	$s0-s7	Temporaries	Should be saved on calls
$24-25	$t8-t9	Temporaries; volatile	Not saved on proc. Calls
$26-27	$k0-k1	Reserved for O.S.	Don’t use them
$28	$gp	Pointer to global static memory	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Proc./funct return address	

MIPS = RISC = Load-Store architecture

- Every operand must be in a register
  - Except for some small integer constants that can be in the instruction itself (see later)
- Variables have to be **loaded** in registers
- Results have to be **stored** in memory
- Explicit Load and Store instructions are needed because there are many more variables than the number of registers

Example

- The HLL statements

  a = b + c

  d = a + b

- will be "translated" into assembly language as:

  load b in register rx

  load c in register ry

  rz <- rx + ry

  store rz in a

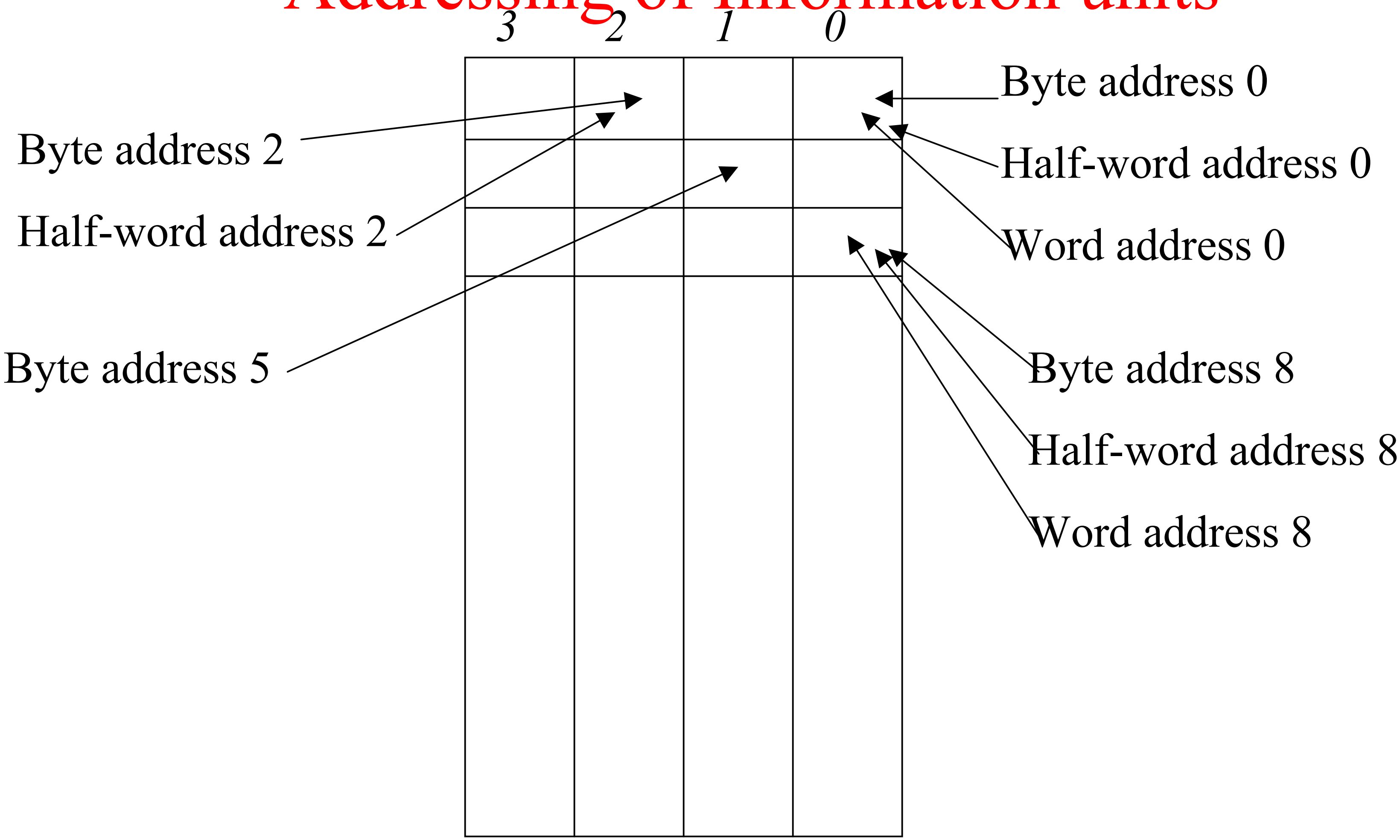
  rt <- rz + rx

  store rt in d

MIPS Information units

- Data types and size:
  - Byte
  - Half-word (2 bytes)
  - Word (4 bytes)
  - Float (4 bytes; single precision format)
  - Double (8 bytes; double-precision format)
- Memory is **byte-addressable**
- A data type must start at an address evenly divisible by its size (in bytes)
- In little-endian environment, the address of a data type is the address of its lowest byte

Addressing of Information units



SPIM Convention

Words listed from left to right but little endians within words

[0x7fffebd0]   0x00400018   0x00000001   0x00000005   0x00010aff

Byte 7fffebd2   Word 7fffebd4   Half-word 7fffebde

Assembly Language programming or How to be nice to your TAs

- Use lots of detailed comments
- Don’t be too fancy
- Use lots of detailed comments
- Use words (rather than bytes) whenever possible
- Use lots of detailed comments
- Remember: The address of a word is evenly divisible by 4
- Use lots of detailed comments
- The word following the word at address i is at address $i+4$
- Use lots of detailed comments

MIPS Instruction types

- Few of them (RISC philosophy)
- Arithmetic
  - Integer (signed and unsigned); Floating-point
- Logical and Shift
  - work on bit strings
- Load and Store
  - for various data types (bytes, words,…)
- Compare (of values in registers)
- Branch and jumps (flow of control)
  - Includes procedure/function calls and returns

Notation for SPIM instructions

- Opcode rd, rs, rt
- Opcode rt, rs, immed
- where
  - rd is always a destination register (result)
  - rs is always a source register (read-only)
  - rt can be either a source or a destination (depends on the opcode)
  - immed is a 16-bit constant (signed or unsigned)

Arithmetic instructions in SPIM

- Don’t confuse the SPIM format with the “encoding” of instructions that we’ll see soon

Opcode	Operands	Comments
Add	rd,rs,rt	#rd = rs + rt
Addi	rt,rs,immed	#rt = rs + immed
Sub	rd,rs,rt	#rd = rs - rt

Examples

```
Add   $8,$9,$10     #$8=$9+$10
Add   $t0,$t1,$t2   #$t0=$t1+$t2
Sub   $s2,$s1,$s0   #$s2=$s1-$s0

Addi  $a0,$t0,20    #$a0=$t0+20
Addi  $a0,$t0,-20   #$a0=$t0-20

Addi  $t0,$0,0      #clear $t0
Sub   $t5,$0,$t5    #$t5 = -$t5
```

Integer arithmetic

- Numbers can be *signed* or *unsigned*
- Arithmetic instructions (+,-,*,/) exist for both signed and unsigned numbers (differentiated by Opcode)
  - Example: Add and Addu
    Addi and Addiu
    Mult and Multu
- Signed numbers are represented in 2's complement
- For Add and Subtract, computation is the same but
  - Add, Sub, Addi cause exceptions in case of *overflow*
  - Addu, Subu, Addiu don't

How does the CPU know if the numbers are signed or unsigned?

- It does not!
- **You do** (or the compiler does)
- You have to tell the machine by using the right instruction (e.g. Add or Addu)