Overview of MIPS ISA

MIPS is a "computer family": R2000/R3000 (32-bit), R 4000/4400 (64-bit), R8000 (for scientific/graphic), R10000 (64-bit)

MIPS originated as a Stanford project: **M**icroprocessor without **I**nterlocked **P**ipe **S**tages

MIPS company bought by Silicon Graphics (SGI)

MIPS processors in (older) DEC Stations and SGI products
MIPS is a RISC

MIPS Registers

Thirty-two 32-bit registers: \$0, \$1,...\$31 used for:

- integer arithmetic
- address calculations
- special-purpose function defined by convention
- temporaries

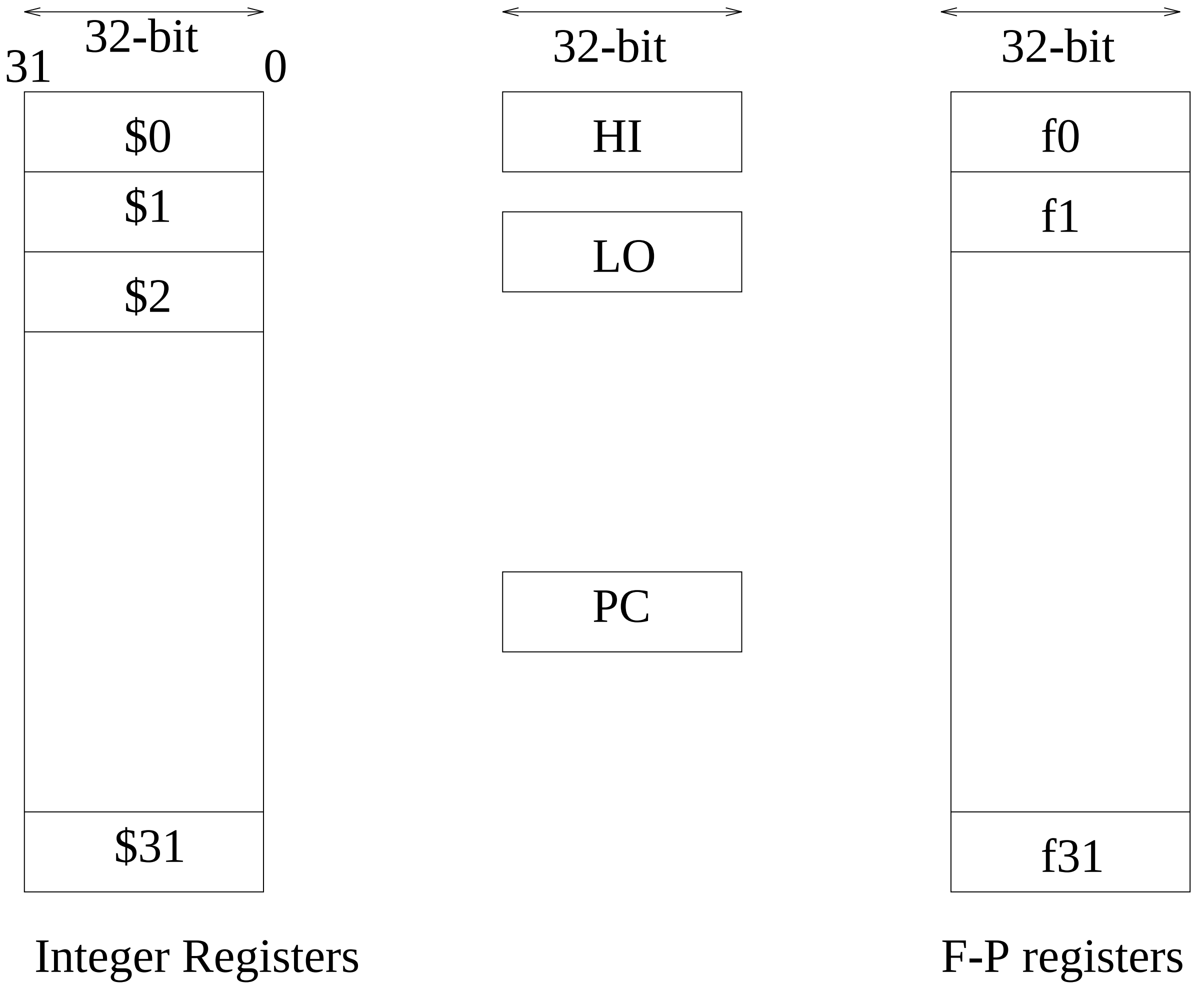
A 32-bit Program Counter (PC)

Two 32-bit registers HI and LO used specifically for Multiply and Divide

Thirty-two 32-bit registers \$f0,\$f1,...,\$f31 used for floating-point arithmetic (often used as 16 64-bit registers)

Other special-purpose registers (see later)

MIPS Registers are part of the "state" of the 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	expression eval./funct. return	
\$4-7	\$a0-a3	proc./funct. call parameters	
\$8-15	\$t0-t7	temporaries;volatile	not saved on proc. call
\$16-23	\$s0-s7	temporaries	should be saved on calls
\$24-25	\$t8-t9	temporaries;volatile	not saved on proc. call
\$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 Load-store architecture

Every operand of a MIPS instruction must be in a register (with a few exceptions)

Variables have to be *load*ed in registers.

Results have to be *store*d in memory

For example, the HLL statements:

```
a = b + c
d = a + b
```

will be "translated" into:

```
Load b in register rx
Load c in register ry
rz <- rx + ry
Store rz in a
rt <- rz + rx
Store rt in d
```

Of course, there are more variables than registers, so we need these explicit load and stores.

MIPS Information units

Data types and size

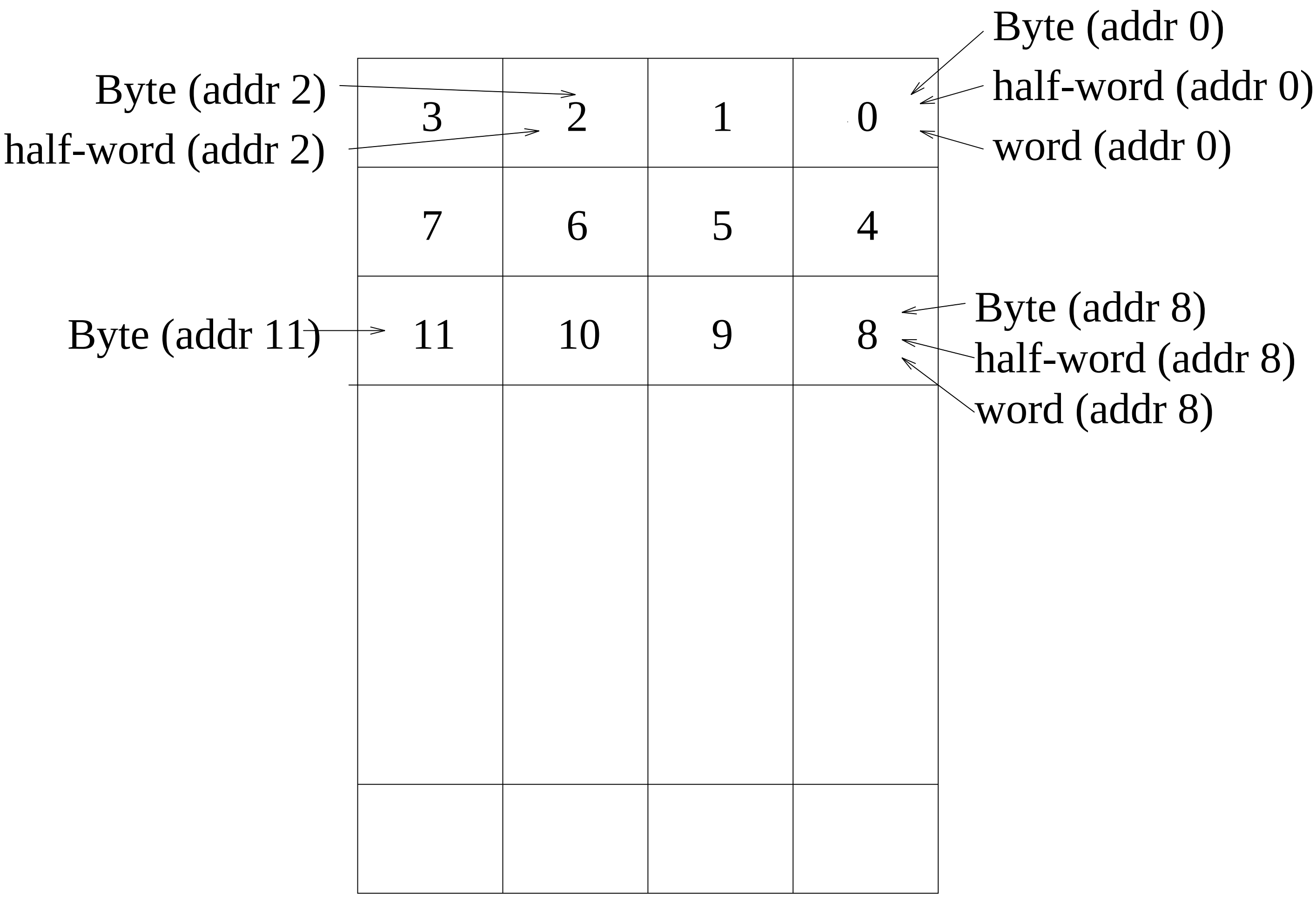
- Byte
- Half-word (2 bytes)
- Word (4 bytes)
- Float (4 bytes using single-precision floating-point format)
- Double (8 bytes using double-precision floating-point format)

Memory is byte addressable.

A data type must start on an address evenly divisible by its size (in bytes)

In a little endian environment like the one used in this class, the address of a data type is the address of its lowest byte.

Addressing revisited



SPIM Convention

Words listed from left to right but little endian within words

[0x7fffebdo] 0x00400018 0x00000001 0x00000005 0x00010aff

byte 7fffebd2 word 7fffebd4 half-word 7fffebde

Assembly Language programming
or
How to be nice to your local TA

- use lots of detailed comments
- Don't be too fancy
- use lots of detailed comments
- Use words 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 adress i is at address $i + 4$

MIPS Instruction types

There are few of them:

1. Arithmetic (integer and floating-point)
2. Logical
3. Load and store
4. Compare
5. Branch and jump (flow of control)

Notation

Opcode rd, rs, rt

Opcode rt, rs, immed

- rd: destination register (result)
- rs: source register (read-only)
- rt: source/destination register (read or read + modified)
- Immed: a 16-bit value (constant)

Arithmetic instructions in SPIM

(Note: don't confuse the SPIM format with the encoding of instructions; we'll see encoding in a few minutes)

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	$s0,$s1,$s2	#s0 = s1 - s2
ADDI	$a0,$t6,20	#a0 = t6 + 20
ADDI	$a0,$t6,-20	#a0 = t6 - 20
SUB	$t5,$0,$t5	#t5 = -t5

Integer arithmetic

Numbers can be either *signed* or *unsigned*

If the numbers are signed they are represented in *2's complement* notation.

There exists instructions for (+,-,*,/) of signed and unsigned integers, for example:

- ADD and ADDU
- ADDI and ADDIU
- MULT and MULTU

For add and subtract, the computation is the same for both but the machine will signal an *overflow* when there is one for signed numbers.

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

It does not.

YOU should know!

You have to tell the machine by using the right instruction (ADD or ADDU)

Instruction Encoding

The ISA defines the format of an instruction (syntax) and what it does (semantics)

Format = Encoding. Each instruction format has different fields

OPCODE field says what the instruction does (semantics)

OPERAND fields say where to find input(s) and output(s) of the instruction

MIPS Instruction Encoding

MIPS is a RISC $\leftarrow$ few (3+) instruction formats

All instructions are of the same size (32 bits = 1 word)

The formats are consistent with each other (OPCODE in the same place etc...)

I-type format

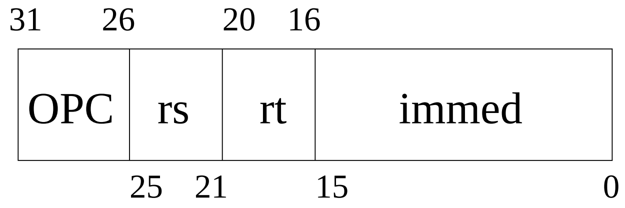
An immediate instruction has the form:

Opcode	Operands	Comments
XXXI	rt,rs,immed	

Recall that we have 32 registers, so we need 5 bits each for rs and rt

We allow 6 bits for opcode (64 instructions...but see later)

Leaves 16 bits for immediate field



I-type format example

Example:

Opcode	Operands	Comments
ADDI	$a0,$12,33	#a0 = $t4 + 33

The ADDI opcode is 08.

31 26		20 16	
08	12	04	33
	25 21		15 0

How would that be in hex?

Sign Extension

Internally, the ALU deals only with 32-bit numbers.

What happens to the 16-bit constant.

It is extended to a 32-bit number by replicating the sign bit (msb) in the 16 leftmost bits.

For a positive number will fill with 0's; for a negative number will fill with 1's.

R-type (register) format

Arithmetic-Logical and Comparison instructions require the encoding of 3 registers. The rest can be OPCODE.

To keep regularity of format OPCODE has a primary “opcode” and a “function” field.

Also needs for a constant in the case of SHIFT instructions (will be discussed soon).

The 16 bits used for immediate in the I-format are split into 5 bits for rd, 5 bits for sh_amt, and 6 bits for function.

Opcode	Operands	Comments
XXX	rd,rt,rs	

31 26		20 16		10 6	
OPC	rs	rt	rd	sht	fcn
	25 21		15 11		5 0

R-type Example

Opcode	Operands	Comments
SUB	\$7,\$8,\$9	#\$7 = \$8 - \$9

31 26		20 16		10 6	
0	8	9	7	0	34
	25 21		15 11		5 0