

Instructions: a simple example

- A C statement

`f = (g + h) - (i + j)`

- `f, g, h, i, j` are assigned to `$s0, $s1, $s2, $s3, $s4`

`add $t0, $s1, $s2`

`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

Load and store instructions

- Load and store instructions
- `lw $tn, c_off($S_base)`
- `sw $tn, c_off($S_base)`
- Example:

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

MIPS code: `lw $t0, 32($s3)`
`add $s1, $s2, $t0`

`$tn` : destination register
`$S_base` : register with base address
`c_off` : offset from base

`g` : `$s1`
`h` : `$s2`
base address of `A` : `$s3`

- Spilling registers
 - *doubly* slow

Load and store instructions

- Example:

C code: `A[12] = h + A[8];`

MIPS code: `lw $t0, 32($s3)`
 `add $t0, $s2, $t0`
 `sw $t0, 48($s3)`

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

Can't write: `add 48($s3), $s2, 32($s3)`

So far we've learned:

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

<u>Instruction</u>	<u>Meaning</u>
<code>add \$s1, \$s2, \$s3</code>	<code>\$s1 = \$s2 + \$s3</code>
<code>sub \$s1, \$s2, \$s3</code>	<code>\$s1 = \$s2 - \$s3</code>
<code>lw \$s1, 100(\$s2)</code>	<code>\$s1 = Memory[\$s2+100]</code>
<code>sw \$s1, 100(\$s2)</code>	<code>Memory[\$s2+100] = \$s1</code>

Constants

- To use a constant, have to use memory, just like for variables
- Example: add 4 to register `$s3`

```
lw $t0, AddrConstant4($s1)    # t0 is the constant 4
```

```
add $s3, $s3, $t0
```

- Quick add instruction: **addi**

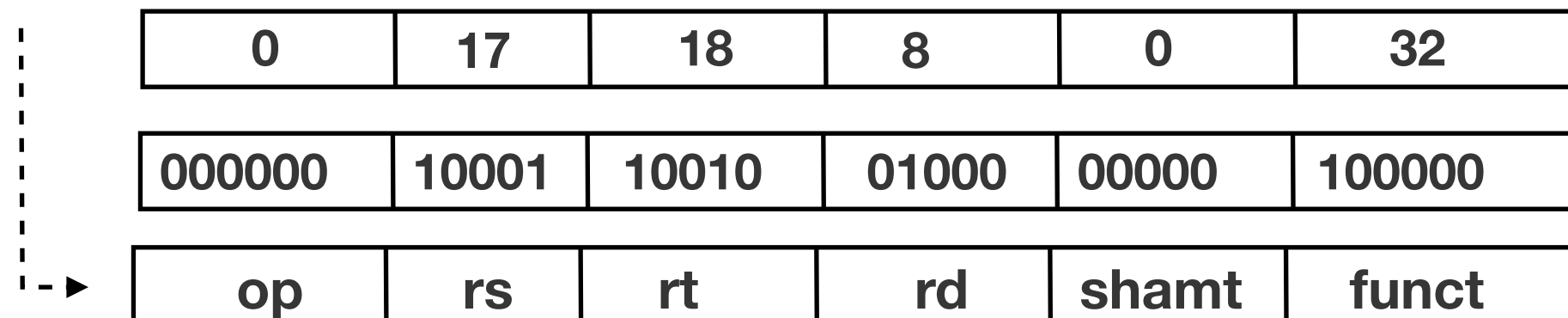
```
addi $s3, $s3, 4
```

- Design principle: make the common case fast

Representing Instructions in the Computer Machine Language

- Instructions, like registers and words (data), are also 32 bits long
 - Example: `add $t0, $s1, $s2`
 - registers **must** have numbers (why?) `$t0=8, $s1=17, $s2=18`

- **Instruction Format:**



op 6-bits **o**pcode that specifies the operation
rs 5-bits **r**egister file address of the first **s**ource operand
rt 5-bits **r**egister file address of the second source operand
rd 5-bits **r**egister file address of the result's **d**estination
shamt 5-bits **s**hift **a**mount (for shift instructions)
funct 6-bits **f**unction code augmenting the opcode

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
\$zero	0	constant 0 (hardware)	n.a.
\$at	1	reserved for assembler	n.a.
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	yes
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	yes
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return addr (hardware)	yes

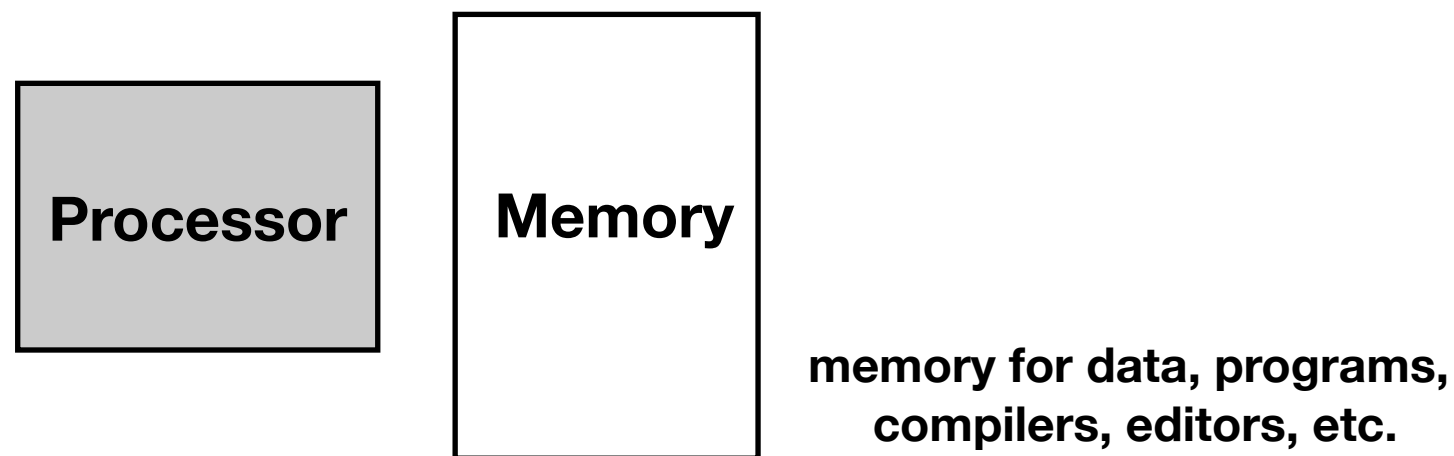
Machine Language

- What if an instruction needs longer fields
 - e.g.: in **lw**, address of constant may be more than 32 (2^5)
 - conflict: keep instruction length same vs. have a single instruction format
 - New principle: *Good design demands a compromise*
 - Here: different formats for different instructions (keep length same)
- Introduce a new type of instruction format
 - I-format for data transfer instructions and immediate instructions
 - other format was R-format for register
- Example: **lw \$t0, 44(\$s3)**

35	19	9	44
op	rs	rt	16 bit number

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

Basic MIPS instructions encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32	n.a.
sub	R	0	reg	reg	reg	0	34	n.a.
addi	I	8	reg	reg	n.a.	n.a.	n.a.	constant
lw	I	35	reg	reg	n.a.	n.a.	n.a.	address
sw	I	43	reg	reg	n.a.	n.a.	n.a.	address

Finally: Programmer to Computer

$$A[300] = h + A[300]$$

\$t1 holds base of array A
\$s2 holds h

```
lw    $t0, 1200($t1)
add   $t0, $s2, $t0
sw    $t0, 1200($t1)
```

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	1001	1000	0000 0100 1011 0000		
0	10010	0 100	0 100	0	100000
101011	1001	0 100	0000 0100 1011 0000		