

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 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:*

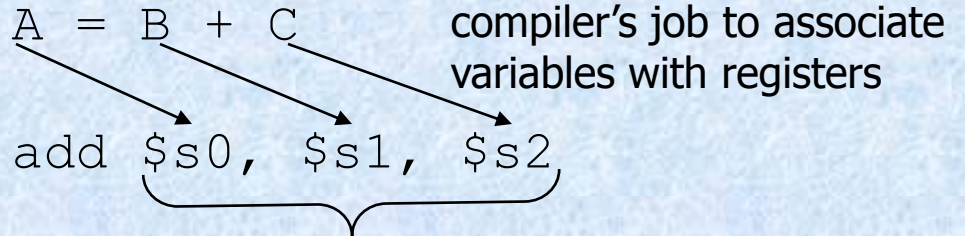
C code:

$A = B + C$

compiler's job to associate
variables with registers

MIPS code:

add $\underbrace{\$s0, \$s1, \$s2}$



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 `add $t0, $s1, $s2`
(arithmetic): `add $s0, $t0, $s3`
 `sub $s4, $s5, $s0`

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

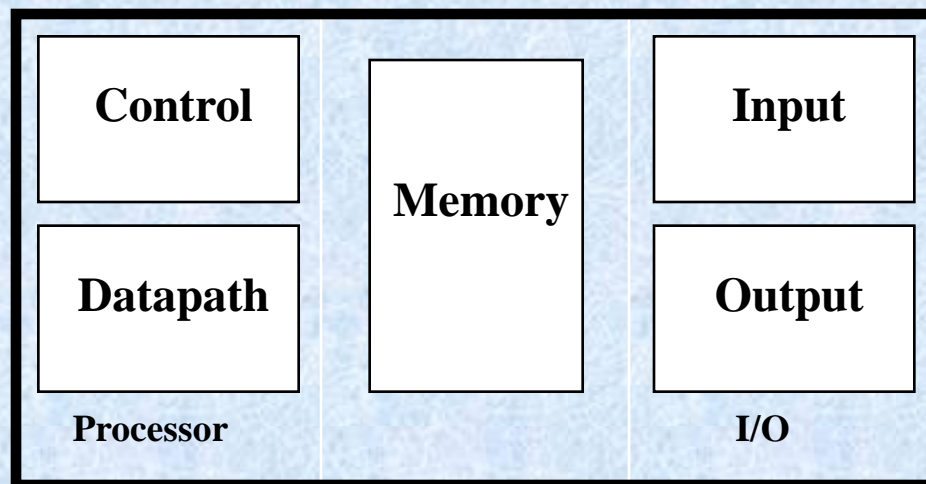
- Performance penalty: high-level code translates to denser machine code.

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

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

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): $lw \quad \$t0, 32(\$s3)$

 (arithmetic): $add \quad \$t0, \$s2, \$t0$

 (store): $sw \quad \$t0, 32(\$s3)$

Diagram illustrating the mapping of C code to MIPS instructions:

- value** points to $\$t0$ in the MIPS code.
- offset** points to 32 in the MIPS code.
- address** points to $(\$s3)$ in the MIPS code.

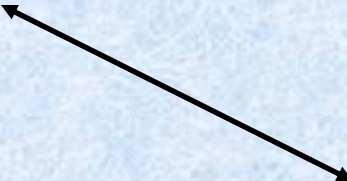
- Load word has destination first, store has destination last
- Remember MIPS arithmetic operands are registers, not memory locations
 - therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory

A MIPS Example

- *Can we figure out the assembly code?*

```
swap(int v[], int k);  
{ int temp;  
  temp = v[k];  
  v[k] = v[k+1];  
  v[k+1] = temp;  
}
```

\$5=k
\$4=Base address of v[]



```
swap:  
mul    $2, $5, 4  
add    $2, $4, $2  
lw     $15, 0($2)  
lw     $16, 4($2)  
sw     $16, 0($2)  
sw     $15, 4($2)  
jr     $31
```

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)

$\$s1 = \text{Memory}[\$s2+100]$

sw \$s1, 100(\$s2)

$\text{Memory}[\$s2+100] = \$s1$

Machine Language

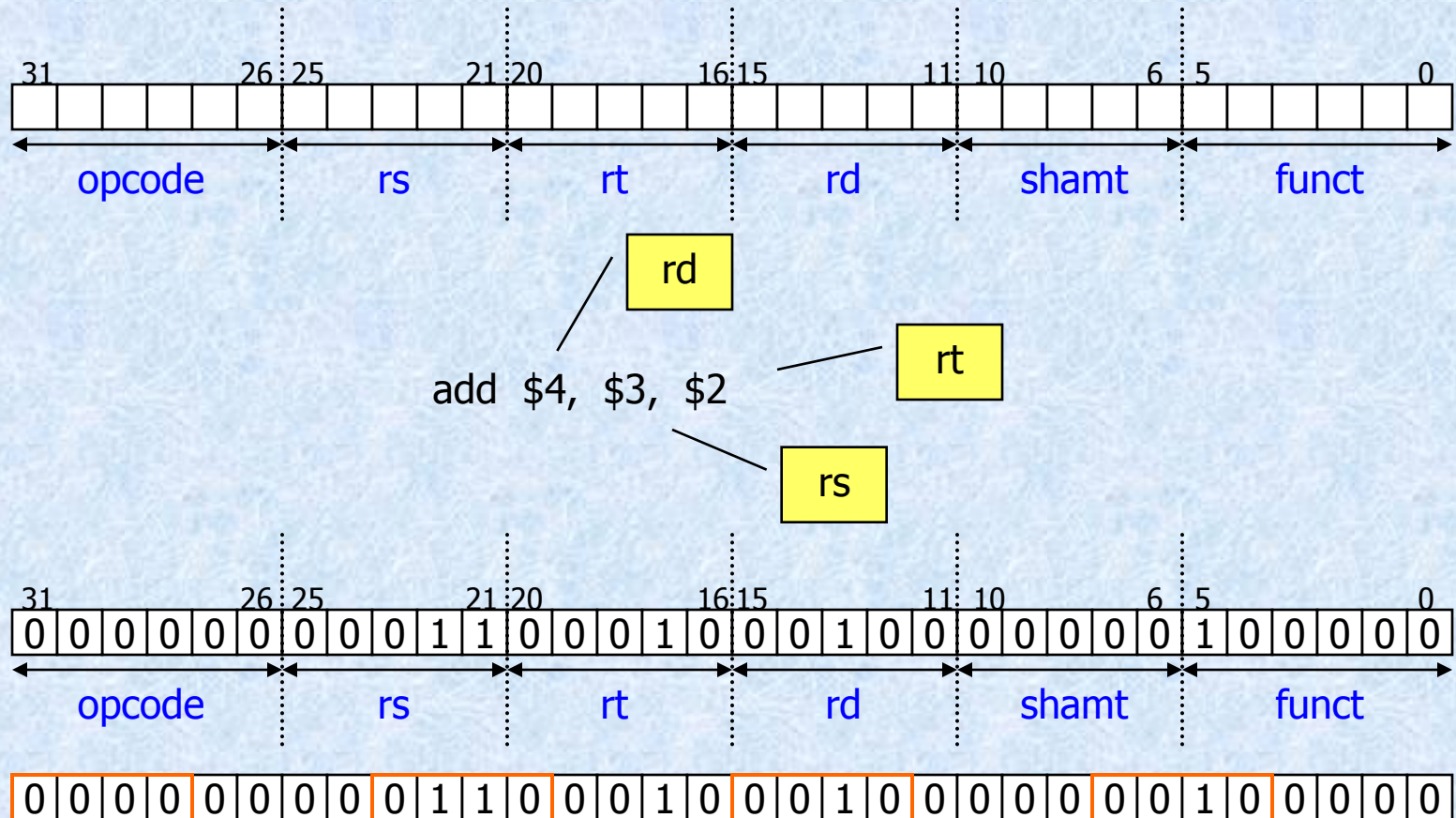
- Instructions, like registers and words of data, are also 32 bits long
 - *Example:* `add $t0, $s1, $s2`
 - registers are numbered, e.g., `$t0` is 8, `$s1` is 17, `$s2` is 18
- Instruction Format **R-type** ("R" for aRithmetic):

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

op	rs	rt	rd	shamt	funct
opcode -	first	second	register	shift	function field -
operation	register	register	destin-	amount	selects variant
	source	source	ation		of operation
	operand	operand	operand		

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
--------	--------	--------	--------	--------	--------

MIPS Encoding: R-Type



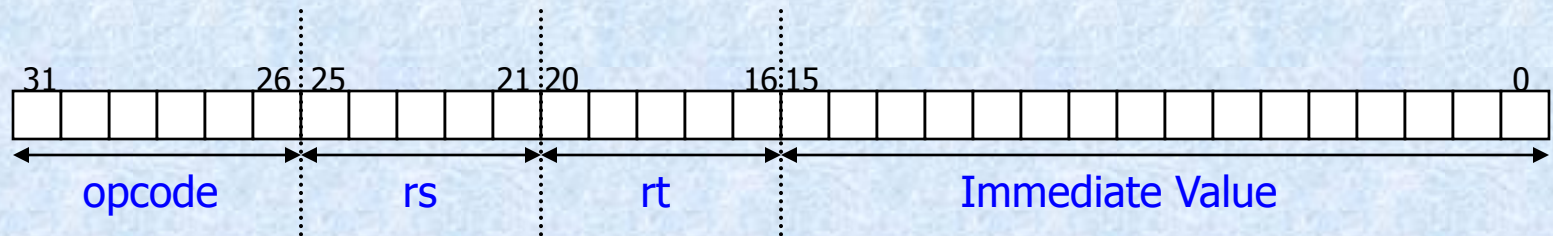
Encoding = 0x00622020

Machine Language

- Consider the load-word and store-word instructions,
 - what would the regularity principle have us do?
 - we would have only 5 or 6 bits to determine the offset from a base register - too little...
- Design Principle 3: *Good design demands a compromise*
- Introduce a new type of instruction format
 - **I-type** ("I" for Immediate) for data transfer instructions
 - *Example*: `lw $t0, 1002($s2)`

100011	10010	01000	0000001111101010
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit offset

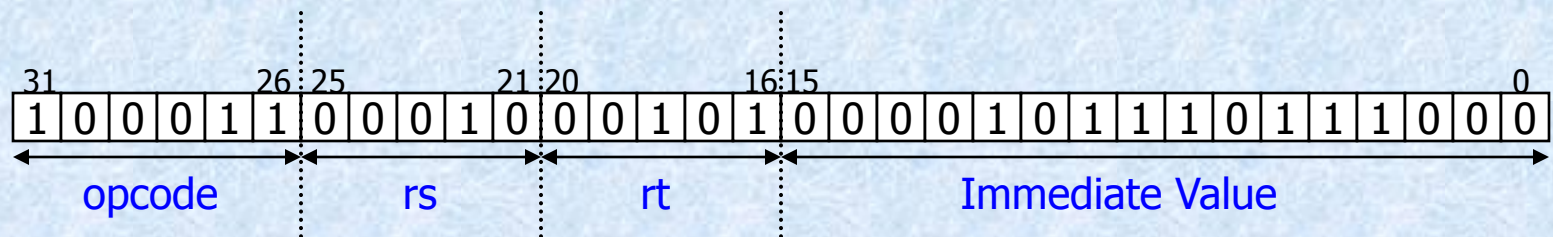
MIPS Encoding: I-Type



lw \$5, 3000(\$2)

Diagram showing the mapping of the instruction fields to the assembly code:

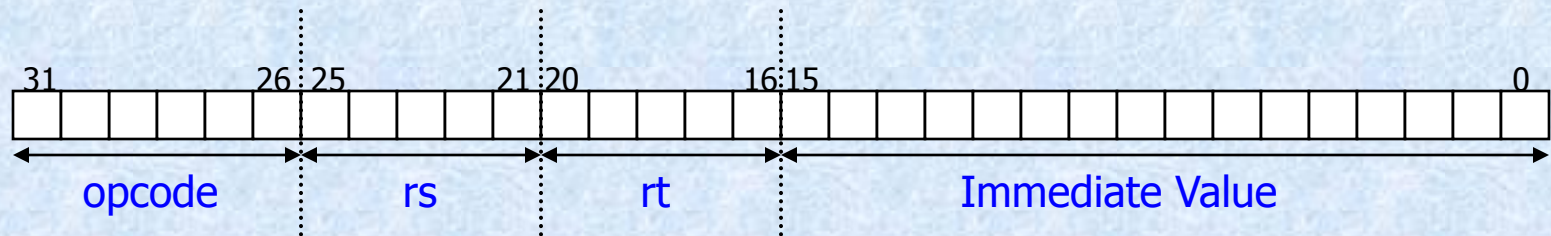
- rt (Register 5)
- Immediate (3000)
- rs (Register 2)



1 0 0 0 1 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0

Encoding = 0x8C450BB8

MIPS Encoding: I-Type

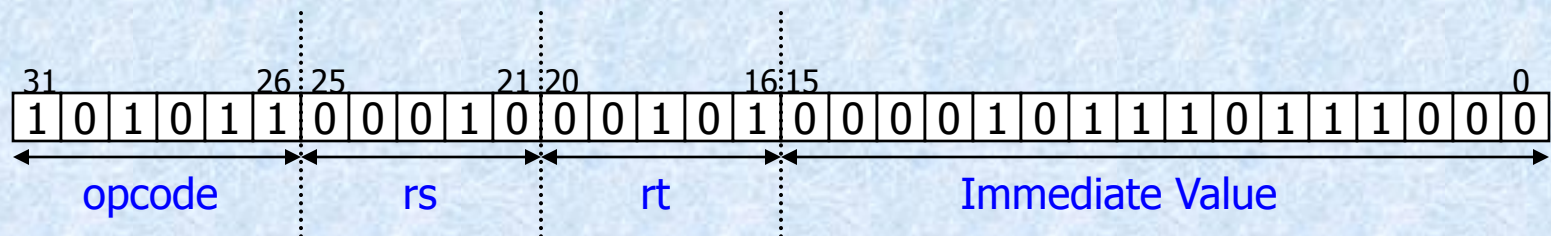


sw \$5, 3000(\$2)

rt

Immediate

rs



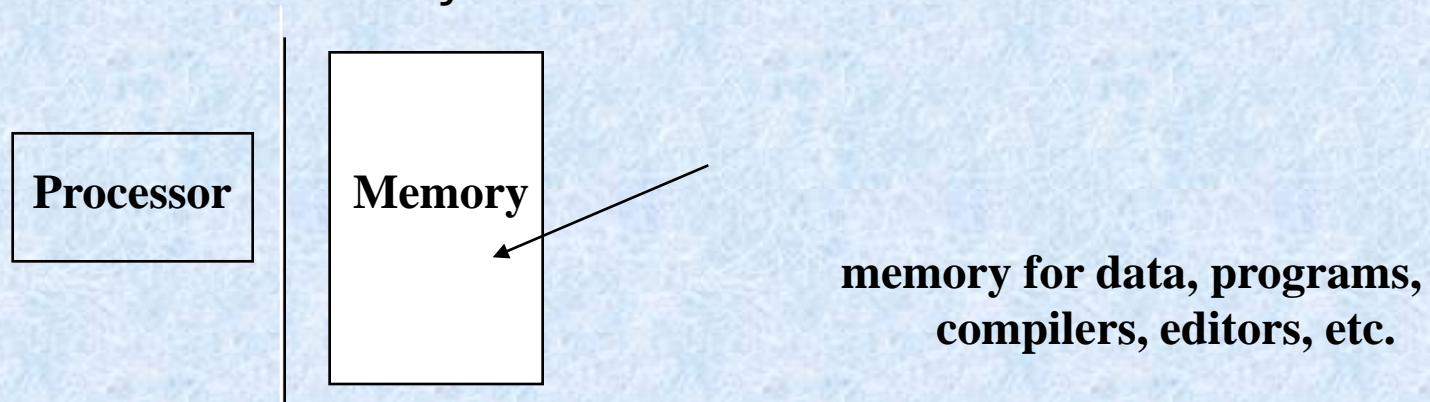
1 0 1 0 1 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0

Encoding = 0xAC450BB8

The immediate value is signed

Stored Program Concept

- *Instructions are bit sequences, just like data*
- 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* (= *execution*)
 - fetch the next instruction and *repeat*

SPIM – the MIPS simulator

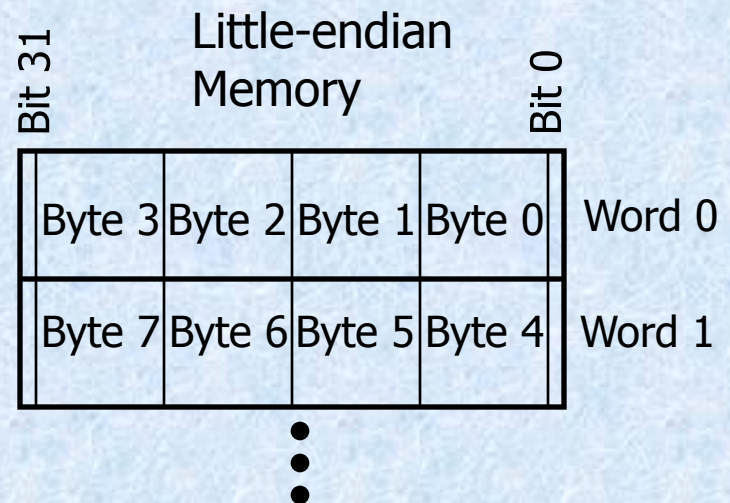
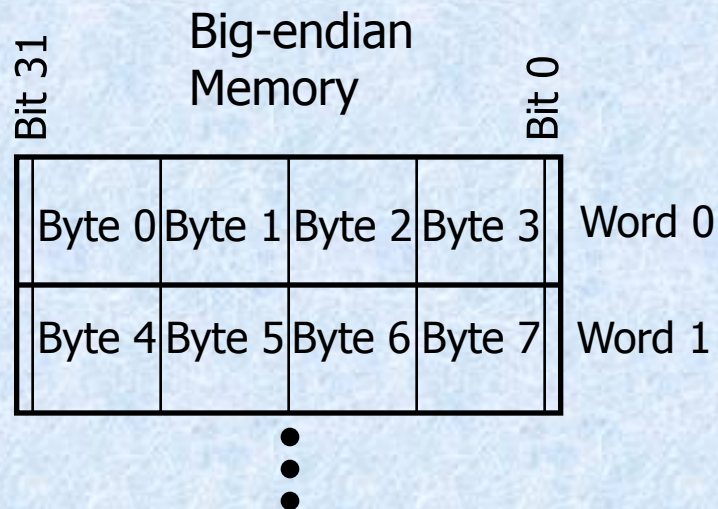
- SPIM (MIPS spelt backwards!) is a MIPS simulator that
 - *reads* MIPS assembly language files and *translates* to machine language
 - *executes* the machine language instructions
 - shows contents of *registers* and *memory*
 - works as a *debugger* (supports *break-points* and *single-stepping*)
 - provides basic *OS-like services*, like simple I/O
- SPIM is freely available on-line

Memory Organization: Big/Little Endian Byte Order

- Bytes in a word can be numbered in two ways:
 - byte 0 at the leftmost (most significant) to byte 3 at the rightmost (least significant), called *big-endian*

0	1	2	3
---	---	---	---
 - byte 3 at the leftmost (most significant) to byte 0 at the rightmost (least significant), called *little-endian*

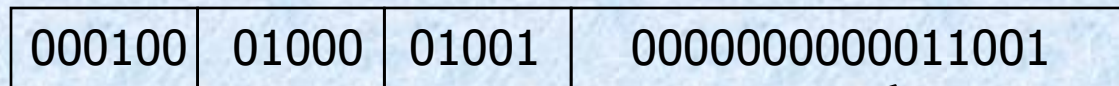
3	2	1	0
---	---	---	---



Control: Conditional Branch

- 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` } I-type instructions



`beq $t0, $t1, Label`
(= addr.100)

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

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

`Label: `

word-relative addressing:
25 words = 100 bytes;
also *PC-relative* (more...)

Addresses in Branch

- Instructions:

bne \$t4, \$t5, Label

Next instruction is at Label if \$t4 != \$t5

beq \$t4, \$t5, Label

Next instruction is at Label if \$t4 = \$t5

- Format:

I	op	rs	rt	16 bit offset
---	----	----	----	---------------

- 16 bits is too small a reach in a 2^{32} address space

- Solution: specify a register (as for `lw` and `sw`) and add it to offset

- use PC (= program counter), called *PC-relative* addressing, based on
- *principle of locality*: most branches are to instructions near current instruction (e.g., loops and *if* statements)

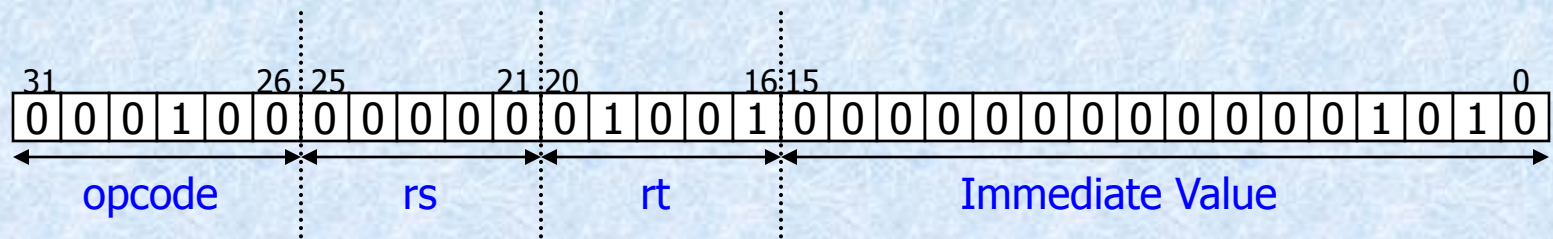
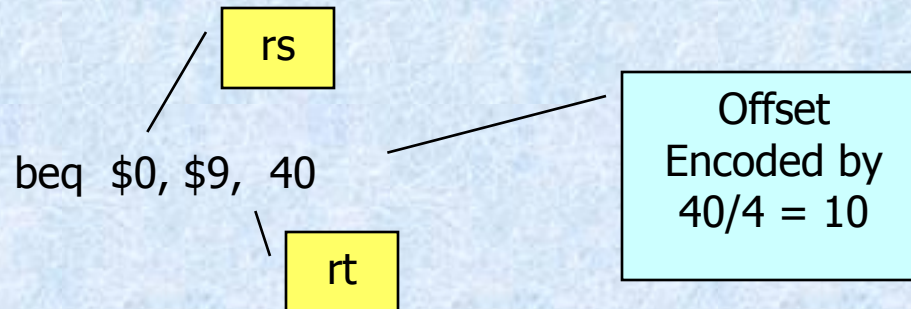
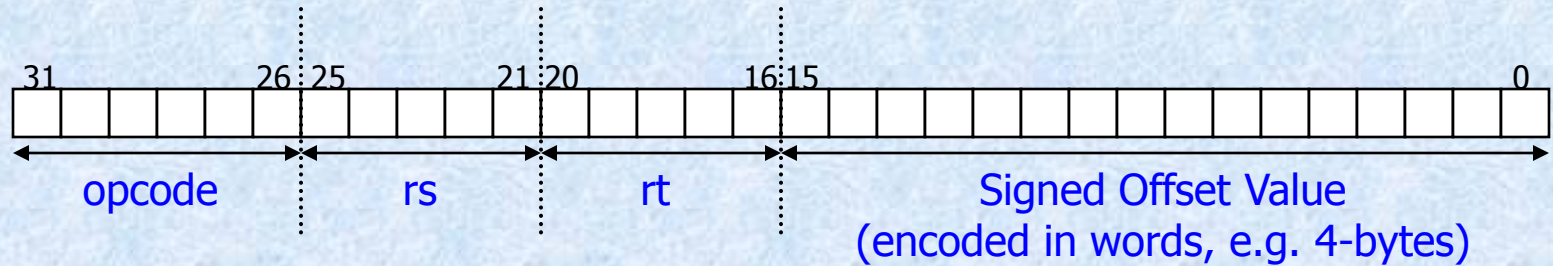
Addresses in Branch

- Further extend reach of branch by observing all MIPS instructions are a word (= 4 bytes), therefore *word-relative* addressing:
- MIPS branch destination address = $\underbrace{(\text{PC} + 4)} + (4 * \text{offset})$

Because hardware typically increments PC early in execute cycle to point to next instruction

- so offset = $(\text{branch destination address} - \text{PC} - 4)/4$

BEQ/BNE uses I-Type



Encoding = 0x1009000A

Control: Unconditional Branch (Jump)

- MIPS unconditional branch instructions:

`j Label`

- *Example:*

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

- **J-type** ("J" for Jump) instruction format

- *Example:* `j Label # addr. Label = 100`

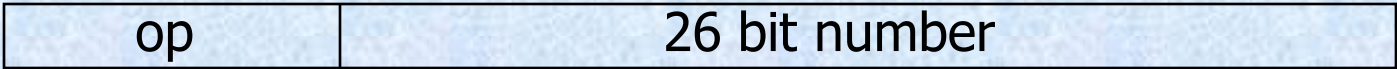
*word-relative
addressing:*

25 words = 100 bytes



6 bits

26 bits



Addresses in Jump

- Word-relative addressing also for jump instructions



- MIPS jump j instruction replaces *lower* 28 bits of the PC with $A00$ where A is the 26 bit address; it *never changes* upper 4 bits
 - *Example:* if $PC = 1011X$ (where $X = 28$ bits), it is replaced with $1011A00$
 - there are $16(=2^4)$ partitions of the 2^{32} size address space, each partition of size 256 MB ($=2^{28}$), *such that*, in each partition the upper 4 bits of the address is same.
 - if a program crosses an address partition, then a j that reaches a different partition has to be replaced by j_r with a full 32-bit address first loaded into the jump register
 - therefore, OS should always try to load a program inside a single partition

Constants

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

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

- Solutions? Will these work?
 - create hard-wired registers (like \$zero) for constants like 1
 - put program constants in memory and load them as required

- MIPS Instructions:

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

- *How to make this work?*

Immediate Operands

- Make operand part of instruction itself!
- Design Principle 4: *Make the common case fast*
- *Example*: `addi $sp, $sp, 4 # $sp = $sp + 4`

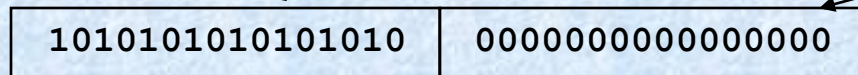
001000	11101	11101	00000000000000100
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit number

How about larger constants?

- First we need to load a 32 bit constant into a register
- Must use two instructions for this: first new *load upper immediate* instruction for upper 16 bits

```
lui $t0, 1010101010101010
```

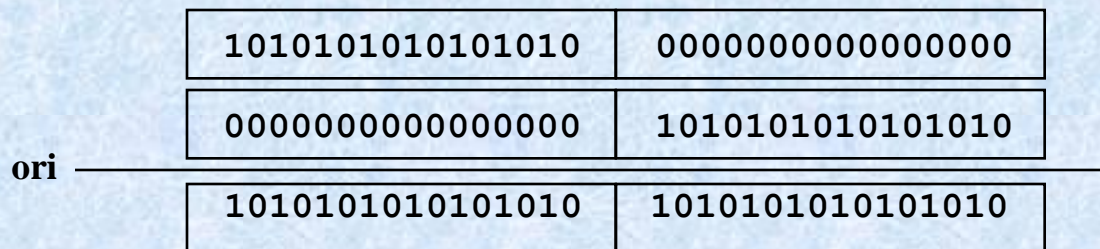
filled with zeros



1010101010101010	0000000000000000
------------------	------------------

- Then get lower 16 bits in place:

```
ori $t0, $t0, 1010101010101010
```



1010101010101010	0000000000000000
------------------	------------------

0000000000000000	1010101010101010
------------------	------------------

ori

1010101010101010	1010101010101010
------------------	------------------

- Now the constant is in place, use register-register arithmetic

So far

<u>Instruction</u>	<u>Format</u>	<u>Meaning</u>
add \$s1,\$s2,\$s3	R	\$s1 = \$s2 + \$s3
sub \$s1,\$s2,\$s3	R	\$s1 = \$s2 - \$s3
lw \$s1,100(\$s2)	I	\$s1 = Memory[\$s2+100]
sw \$s1,100(\$s2)	I	Memory[\$s2+100] = \$s1
bne \$s4,\$s5,Lab1	I	Next instr. is at Lab1 if \$s4 != \$s5
beq \$s4,\$s5,Lab2	I	Next instr. is at Lab2 if \$s4 = \$s5
j Lab3	J	Next instr. is at Lab3

■ Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Logical Operations

- ➔ Shift Logical Left (SLL \$S1,\$S2,10)
- ➔ Shift Logical Right (SRL \$S1,\$S2,10)
- ➔ AND (AND \$S1,\$S2,\$S3)
- ➔ OR (OR \$S1,\$S2,\$S3)
- ➔ NOR (NOR \$S1,\$S2,\$S3)
- ➔ ANDI (ANDI \$S1,\$S2,100)
- ➔ ORI (ORI \$S1,\$S2,100)

Shift Operations

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - sll by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - srl by i bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0

and \$t0, \$t1, \$t2

\$t2	0000	0000	0000	0000	0000	1101	1100	0000
\$t1	0000	0000	0000	0000	0011	1100	0000	0000
\$t0	0000	0000	0000	0000	0000	1100	0000	0000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged
- or \$t0, \$t1, \$t2

\$t2	0000	0000	0000	0000	0000	1101	1100	0000
\$t1	0000	0000	0000	0000	0011	1100	0000	0000
\$t0	0000	0000	0000	0000	0011	1101	1100	0000

NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero`

Register 0:
always read as
zero

`$t1` 0000 0000 0000 0000 0011 1100 0000 0000

`$t0` 1111 1111 1111 1111 1100 0011 1111 1111

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
 - *how?* We generate more than one instruction – *pseudo-instruction*
 - can now build general control structures
 - Slti is also available
- The assembler needs a register to manufacture instructions from pseudo-instructions
- There is a *convention* (not mandatory) for use of registers

Branch Instruction Design

- Why not b1t, bge, etc?
- Hardware for $<$, \geq , ... slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `slt $t0, $s0, $s1 # signed`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sltu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

Policy-of-Use Convention for Registers

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

Register 1, called \$at, is reserved for the assembler; registers 26-27, called \$k0 and \$k1 are reserved for the operating system.

Assembly Language vs. Machine Language

- Assembly provides convenient *symbolic representation*
 - much easier than writing down numbers
 - regular rules: e.g., destination first
- Machine language is the *underlying reality*
 - e.g., destination is no longer first
- Assembly can provide *pseudo-instructions*
 - e.g., `move $t0, $t1` exists only in assembly
 - would be implemented using `add $t0, $t1, $zero`
- When considering performance you should count actual number of machine instructions that will execute

Procedure Calling

- Steps required
 1. Place parameters in registers
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for caller
 6. Return to place of call

Procedure Call Instructions

- Procedure call: jump and link

`jal ProcedureLabel`

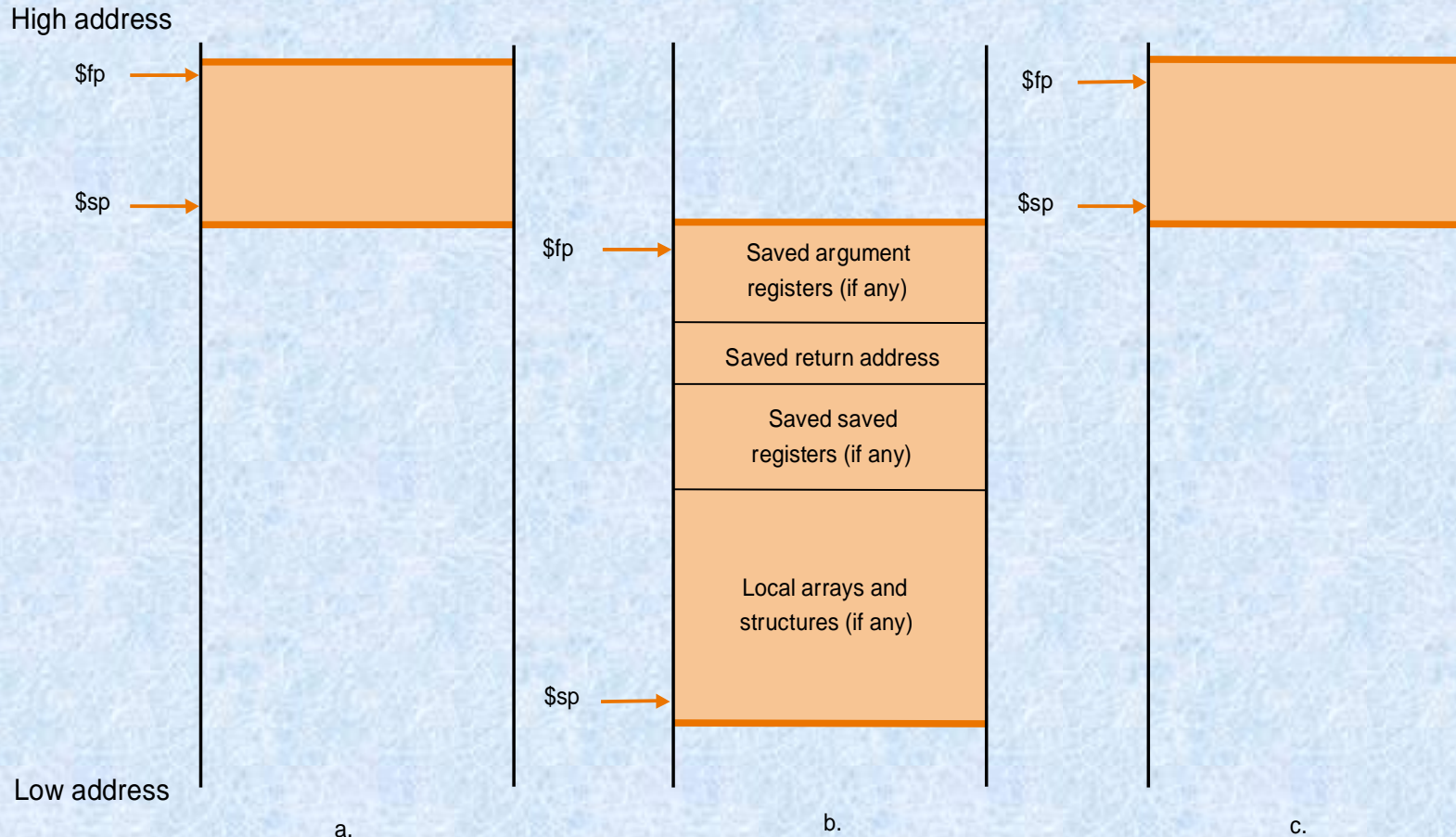
- Address of following instruction put in `$ra`
- Jumps to target address

- Procedure return: jump register

`jr $ra`

- Copies `$ra` to program counter
- Can also be used for computed jumps
 - e.g., for case/switch statements

Using a Frame Pointer



Variables that are local to a procedure but do not fit into registers (e.g., local arrays, structures, etc.) are also stored in the stack. This area of the stack is the *frame*. The *frame pointer* \$fp points to the top of the frame and the stack pointer to the bottom. The frame pointer does not change during procedure execution, unlike the stack pointer, so it is a stable base register from which to compute offsets to local variables.

Use of the frame pointer is *optional*. If there are no local variables to store in the stack it is not efficient to use a frame pointer.

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store

- String processing is a common case

`lb rt, offset(rs)` `lh rt, offset(rs)`

- Sign extend to 32 bits in `rt`

`lbu rt, offset(rs)` `lhu rt, offset(rs)`

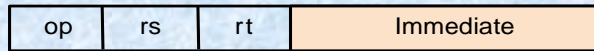
- Zero extend to 32 bits in `rt`

`sb rt, offset(rs)` `sh rt, offset(rs)`

- Store just rightmost byte/halfword

MIPS Addressing Modes

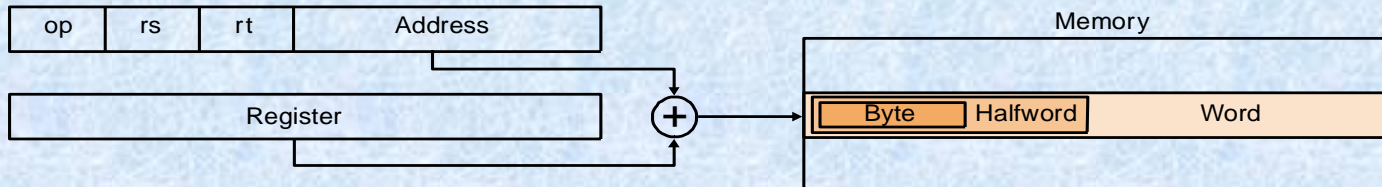
1. Immediate addressing



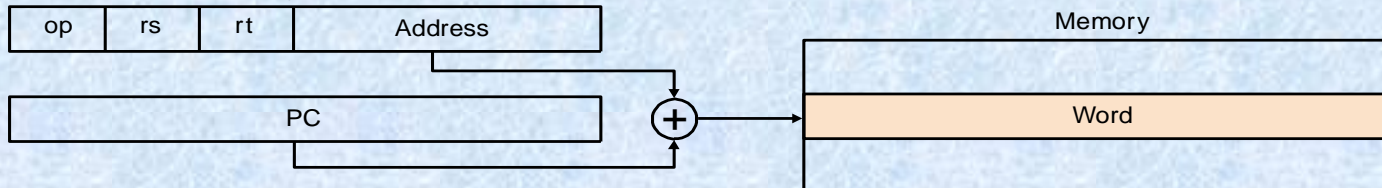
2. Register addressing



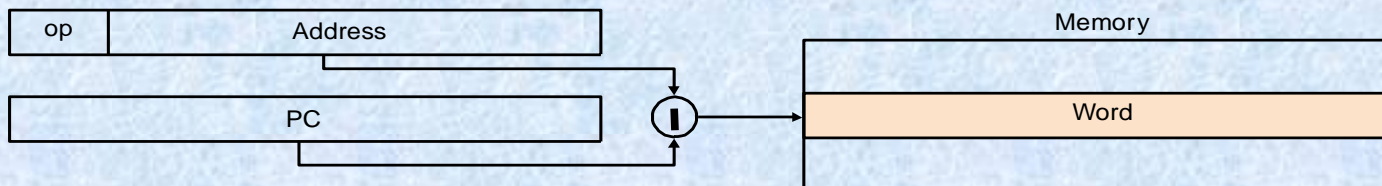
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Overview of MIPS

- Simple instructions – all 32 bits wide
- Very structured – no unnecessary baggage
- Only three instruction formats

R	op	rs	rt	rd	shamt	funct
---	----	----	----	----	-------	-------

I	op	rs	rt	16 bit address
---	----	----	----	----------------

J	op	26 bit address
---	----	----------------

Summarize MIPS:

MIPS operands

Name	Example	Comments
32 registers	<code>\$s0-\$s7, \$t0-\$t9, \$zero,</code> <code>\$a0-\$a3, \$v0-\$v1, \$gp,</code> <code>\$fp, \$sp, \$ra, \$at</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register <code>\$zero</code> always equals 0. Register <code>\$at</code> is reserved for the assembler to handle large constants.
2^{30} memory words	<code>Memory[0],</code> <code>Memory[4], ...,</code> <code>Memory[4294967292]</code>	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1, 100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$s1, \$s2, 25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1, \$s2, 25	if ($\$s1 != \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1, \$s2, \$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set less than immediate	slti \$s1, \$s2, 100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = PC + 4$; go to 10000	For procedure call

Alternative Architectures

- Design alternative:
 - provide more powerful operations
 - goal is to reduce number of instructions executed
 - danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as *R(educed)ISC* vs. *C(omplex)ISC*

A dominant architecture: 80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added.....

A dominant architecture: 80x86

- Complexity
 - instructions from 1 to 17 bytes long
 - one operand *must* act as both a source and destination
 - one operand *may* come from memory
 - several complex addressing modes
- Saving grace:
 - the most frequently used instructions are not too difficult to build
 - compilers avoid the portions of the architecture that are slow

Summary

- Instruction complexity is only one variable
 - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
 - simplicity favors regularity
 - smaller is faster
 - good design demands compromise
 - make the common case fast