RISC vs. CISC Machines

Feature	RISC	CISC
Registers	ඎ32	6, 8, 16
Register Classes	One	Some
Arithmetic Operands	Registers	Memory+Registers
Instructions	3-addr	2-addr
Addressing Modes	r M[r+c] (l,s)	several
Instruction Length	32 bits	Variable
Side-effects	None	Some
Instruction-Cost	“Uniform”	Varied

Main Types of Instructions

- Arithmetic
  - Integer
  - Floating Point
- Memory access instructions
  - Load & Store
- Control flow
  - Jump
  - Conditional Branch
  - Call & Return

MIPS arithmetic

- Most instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `A = B + C`

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

($s0, $s1 and $s2 are associated with variables by compiler)

MIPS arithmetic

C code:

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

MIPS code:

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

- Operands must be registers, only 32 registers provided
- Design Principle: *smaller is faster*. Why?

Instructions: load and store

Example:

C code:

```
A[8] = h + A[8];
```

MIPS code:

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 32($s3)
```

- Store word operation has no destination (reg) operand
- Remember arithmetic operands are registers, not memory!

Our First C code translated

- Can we figure out the code?

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

➡

```
swap:
   muli $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
```

Explanation:
  index k : $5
  base address of v: $4
  address of v[k] is $4 + 4.$5

Machine Language: R-type instr

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - Registers have numbers: `$t0=9, $s1=17, $s2=18`
- Instruction Format:

op	***rs***	***rt***	***rd***	***shamt***	***funct***
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Can you guess what the field names stand for?

Machine Language: I-type instr

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: *Good design demands a compromise*
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	9	32
op	rs	rt	16 bit number

Control

- 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
```

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

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

Control

- MIPS unconditional branch instructions:

```
    j   label
```

- Example:

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

- *Can you build a simple for loop?*

So far (including J-type instr):

- <u>Instruction</u>   <u>Meaning</u>

```
add $s1,$s2,$s3  $s1 = $s2 + $s3
sub $s1,$s2,$s3  $s1 = $s2 – $s3
lw $s1,100($s2)  $s1 = Memory[$s2+100]
sw $s1,100($s2)  Memory[$s2+100] = $s1
bne $s4,$s5,L    Next instr. is at Label if $s4 ° $s5
beq $s4,$s5,L    Next instr. is at Label if $s4 = $s5
j Label          Next instr. is at Label
```

- Formats:

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

Instructions: Jump

Instruction	Meaning
jal my_proc	jump and link start procedure my_proc $ra holds address of instruction following the jal
jr $ra	jump register return from procedure call puts $ra value back into the PC

Modern Architectures

- Parameters
  - first k parameters are passed in registers, others on the stack
- Return address
  - normally saved in a register on a call (jal)
  - a non-leaf procedure saves this value on the stack
- Function result
  - normally saved in a register on a return

- No stack support in the hardware

Stack Operations in RISC

- PUSH

```
sub   $sp, 4
sw    $ra, ($sp)
```

- POP

```
lw   $ra, ($sp)
add   $sp, 4
```

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
slt $t0, $s1, $s2
```

meaning:

```
if  $s1 < $s2 then
   $t0 = 1
else
   $t0 = 0
```

- Can use this instruction to build "`blt $s1, $s2, Label`"
  — can now build general control structures

- Note that the assembler needs a register to do this,
  — use conventions for registers

Used MIPS compiler conventions

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-$s7`	16-23	saved (by callee)
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Small Constants: immediates

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

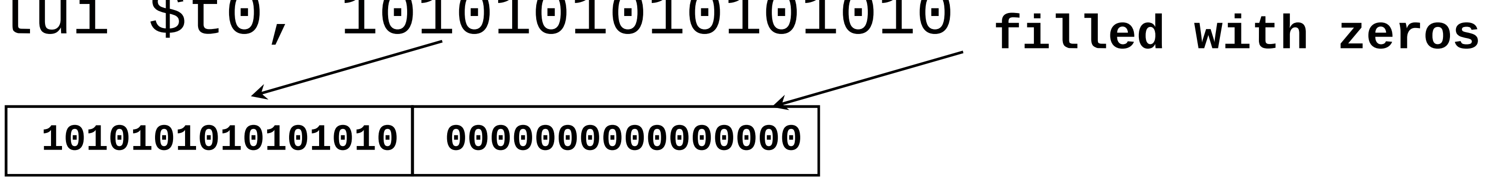
- MIPS Instructions:

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

How about larger constants?

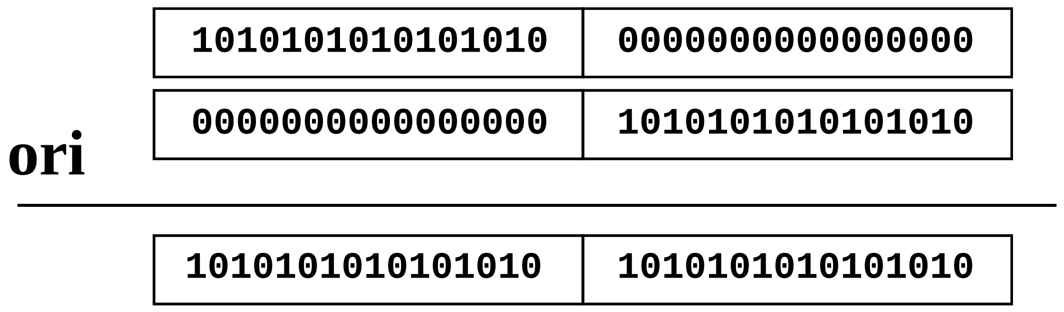
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions; new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```



- Then must get the lower order bits right, i.e.,

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



Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide '*pseudoinstructions*'
  - e.g., "`move $t0, $t1`" exists only in Assembly
  - would be implemented using "`add $t0,$t1,$zero`"
- When considering performance you should count real instructions

Instructions: Load and Store

- load from memory to register
- store from register to memory
- load immediate

Instruction	Meaning
li $v0, 4	$v0 ← 4
la $a0, msg	$a0 ← address of msg
lw $t0, x	$t0 ← x
sw $t0, y	y ← $t0

Remarks

- la vs. li
  - Since a label represents a fixed memory address after assembly, la is actually a special case of load immediate.
- lw vs. la
  - x is at address 10 and contains 2

```
la   $a0, x          $a0 ← 10
lw  $a0, x           $a0 ← 2
```

- lw $t0 8($sp)

Addresses in Branches and Jumps

- Instructions:

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4` $\neq$ `$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4 = $t5`
`j Label`	Next instruction is at Label

- Formats:

I	**op**	**rs**	**rt**	**16 bit address**
J	**op**	**26 bit address**		

- Addresses are not 32 bits
  — How do we handle this with load and store instructions?

Addresses in Branches

- 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

- Formats: use I-type

I	**op**	**rs**	**rt**	**16 bit address**

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
-

  Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

To summarize:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	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.

To summarize:

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)`	`$s1 =` Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	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
Uncondi-tional 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

System Calls

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

Hello World

```
        .text           # text segment
        .global __start
__start:                # execution starts here
        la $a0,str      # put string address into a0
        li $v0,4        #
        syscall         # print
        li v0, 10       #
        syscall         # au revoir...
        .data           # data segment
str:    .asciiz  "hello world\n"
```

Riddle

```
      .data
endl:  .asciiz  “\n”

      .text
print_int:
       li  $v0, 1
       syscall
       jr  $ra
endline:
       la  $a0, endl
       li  $v0, 4
       syscall
       jr  $ra
__start:
        li  $a0, 42
        jal print_int
        jal endline
        li  $a0, 2006
        jal print_int
        jal endline
```