RISC vs. CISC Machines

Feature	RISC	CISC
Registers	℘32	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

MIPS is a RISC

- RISC = Reduced (Regular/Restricted) Instruction Set Computer
- All arithmetic operations are of the form:

```
Rd <- Rs op Rt      # the Rs are registers
```

- Important restriction: MIPS is a load store architecture: the ALU can only operate on registers Why?
- Basic operations (really only a few kinds)
  1. Arithmetic (addition, substraction, etc)
  2. Logical (and, or, xor, etc)
  3. Comparison (less-than, greater-than, etc)
  4. Control (branches, jumps, etc)
  5. Memory access (load and store)
- All MIPS instructions are 32 bits long

MIPS is a Load-Store Architecture

- Every operand of a MIPS instruction must be in a register (with some exceptions)
- Variables must be loaded into registers
- Results have to be stored back into memory
- Example C fragment...

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

- ... would be "translated" into something like:

```
Load b into register Rx
Load c into register Ry
Rz <- Rx + Ry
Store Rz into a
Rz <- Rz + Rx
Store Rz into d
```

MIPS Registers

- Provides thirty-two, 32-bit registers, named $0, $1, $2 .. $31 used for:
  - integer arithmetic
  - address calculations
  - special-purpose functions defined by convention
  - temporaries
- A 32-bit program counter (PC)
- Two 32-bit registers HI and LO used specifically for multiplication and division
- Thirty-two 32-bit registers $f0, $f1, $f2 .. $f31 used for floating point arithmetic
- Other special-purpose registers (see later)

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./function return	
$4-7	$a0-a3	proc/funct call parameters	
$8-15	$t0-t7	volatile temporaries	not saved on call
$16-23	$s0-s7	temporaries (saved across calls)	saved on call
$24-25	$t8-t9	volatile temporaries	not saved on call
$26-27	$k0-k1	reserved kernel/OS	don't use them
$28	$gp	pointer to global data area	
$29	$sp	stack pointer	
$30	$fp	frame pointer	
$31	$ra	proc/funct return address	

MIPS Instruction Types

- As we said earlier, there are very few basic operations :
  1. Memory access (load and store)
  2. Arithmetic (addition, substraction, etc)
  3. Logical (and, or, xor, etc)
  4. Comparison (less-than, greater-than, etc)
  5. Control (branches, jumps, etc)
- We'll use the following notation when describing instructions:

rd: destination register (modified by instruction)
rs: source register (read by instruction)
rt: source/destination register (read or read+modified)
immed: a 16-bit value

Load and Store Instructions

- Data is explicitly moved between memory and registers through load and store instructions.
- Each load or store must specify the memory address of the memory data to be read or written.
- Think of a MIPS address as a 32-bit, unsigned integer.
- Because a MIPS instruction is always 32 bits long, the address must be specified in a more compact way.
- We always use a *base register* to address memory
- The base register points somewhere in memory, and the instruction specifies the register number, and a 16-bit, *signed offset*
- A single base register can be used to access any byte within ??? bytes from where it points in memory.

Load and Store Examples

- Load a word from memory:

```
lw  rt, offset(base)  # rt <- memory[base+offset]
```

- Store a word into memory:

```
sw rt, offset(base)  # memory[base+offset] <- rt
```

- For smaller units (bytes, half-words) only the lower bits of a register are accessible. Also, for loads, you need to specify whether to sign or zero extend the data.

```
lb  rt, offset(base) # rt <- sign-extended byte
lbu rt, offset(base) # rt <- zero-extended byte
sb  rt, offset(base) # store low order byte of rt
```

Arithmetic Instructions

```
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, $8, $10     # r8 <- r9 + r10
ADD     $t0, $t1, $t2   # t0 <- t1 + t2
SUB     $s0, $s0, $s1   # s0 <- s0 - s1
ADDI    $t3, $t4, 5     # t3 <- t4 + 5
```

Flow of Control: Conditional Branches

```
BEQ     rs, rt, target # branch if rs == rt
BNE     rs, rt, target # branch if rs != rt
```

Comparison Between Registers

- What if you want to branch if R6 is greater than R7?
- We can use the SLT instruction:

```
SLT     rd, rs, rt      # if rs<rt then rd <- 1
                        #   else rd <- 0
SLTU    rd, rs, rt      # same, but rs,rt unsigned
```

- Example: Branch to L1 if $5 > $6

```
SLT     $7, $6, $5   # $7 = 1, if $6 < $5
BNE     $7, $0, L1
```

Jump Instructions

- Jump instructions allow for unconditional transfer of control:

```
J       target      # go to specified target
JR      rs          # jump to addr stored in rs
```

- Jump and link is used for procedure calls:

```
JAL     target      # jump to target, $31 <- PC
JALR    rs, rd      # jump to addr in rs
                    # rd <- PC
```

- When calling a procedure, use JAL; to return, use JR $31

Logic Instructions

- Used to manipulate bits within words, set up masks, etc.

```
Opcode  Operands        Comments

AND     rd, rs, rt      # rd <- AND(rs, rt)
ANDI    rt, rs, immed   # rt <- AND(rs, immed)
OR      rd, rs, rt
ORI     rt, rs, immed
XOR     rd, rs, rt
XORI    rt, rs, immed
```

- The immediate constant is limited to 16 bits
- To load a constant in the 16 upper bits of a register we use LUI:

```
Opcode  Operands        Comments

LUI     rt, immed       # rt<31,16> <- immed
                        # rt<15,0>  <- 0
```

Pseudoinstructions

Data moves

Name	Assembly syntax	Expansion	Operation in C
move	`move $t, $s`	`addiu $t, $s, 0`	`t = s`
clear	`clear $t`	`addu $t, $zero, $zero`	`t = 0`
load 16-bit immediate	`li $t, C`	`addiu $t, $zero, C_lo`	`t = C`
load 32-bit immediate	`li $t, C`	`lui $t, C_hi` `ori $t, $t, C_lo`	`t = C`
load label address	`la $t, A`	`lui $t, A_hi` `ori $t, $t, A_lo`	`t = A`


```
        .data
A:      .word  5
B:      .word  10

        .globl outputAMsg
outputAMsg:
        .asciiz "\n A = "
        .globl outputBMsg
outputBMsg:
        .asciiz "\n B = "
        .globl blankMsg
blankMsg:
        .asciiz "   "

        .text
        .globl  foo
foo:
        lw      $t0, 0($a0)
        lw      $t1, 0($a1)
        sw      $t0, 0($a1)
        sw      $t1, 0($a0)
        jr      $ra
```

```
        .globl main
main:
        addu    $s7, $0, $ra

        la      $a0, A
        la      $a1, B
        jal     foo

        li      $v0, 4
        la      $a0, outputAMsg
        syscall
        li      $v0, 1
        lw      $a0, A
        syscall
        li      $v0, 4
        la      $a0, blankMsg

                #Output B
        li      $v0, 4
        la      $a0, outputBMsg
        syscall
        li      $v0, 1
        lw      $a0, B
        syscall

        addu    $ra, $0, $s7
        jr      $ra
        add     $0, $0, $0
```

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"
```