MIPS Architectural Approach

Load/store or register-register instruction set
- Only operate on data in registers
  - Register operations affect the entire contents of register
    - No partial register writes except for single-precision FP
- Only load/store instructions access memory
- True in all RISC instruction sets
- True in all instruction sets designed since 1980

Emphasis on efficient implementation
- Make the common case fast
  - A system can be so simple that it obviously has no bugs, or so complex that it has no obvious bugs. (adapted C. A. R. Hoare)

Simplicity: provide primitives rather than solutions
- Simplicity favors regularity
MIPS Data Types

Bit String: sequence of bits of a particular length
  • 8 bits is a byte
  • 16 bits is a half-word
  • 32 bits is a word
  • 64 bits is a double-word

Character
  • supported as a byte (signed or unsigned)

Integers
  • 2's Complement

Floating Point: M x 2^E
  • single precision
  • double precision
2^{32} bytes of memory: accessible by loads/stores
31 x 32-bit GPRs (R0 = 0) or integer multiply/divide
• why only 32 registers? Smaller is faster
PC: incremented by 4 for each instruction
• except for branch, j, jal

FP registers are paired for double-precision. Specify the even register, which holds the less-significant word.
Arithmetic/Logical instructions
• Three operand format: result + two sources
• Operands: registers, 16-bit immediates
• Signed & unsigned arithmetic operations:
  – Sign-extension for immediates
  – Trapping of overflow for signed values
• Compare instructions
  – Signed vs. unsigned: comparison is different

Integer multiply/divide
• Use HI/LO registers

Floating Point instructions
• Operate on floating point registers
• Double and single precision
• Typical: add, multiply, divide, subtract
### MIPS Integer Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td><code>$1,$2,$3</code></td>
<td><code>$1 = $2 + $3</code></td>
</tr>
<tr>
<td><code>subtract</code></td>
<td><code>$1,$2,$3</code></td>
<td><code>$1 = $2 - $3</code></td>
</tr>
<tr>
<td><code>add immediate</code></td>
<td><code>$1,$2,100</code></td>
<td><code>$1 = $2 + 100</code></td>
</tr>
<tr>
<td><code>add unsigned</code></td>
<td><code>$1,$2,$3</code></td>
<td><code>$1 = $2 + $3</code></td>
</tr>
<tr>
<td><code>subtract unsigned</code></td>
<td><code>$1,$2,$3</code></td>
<td><code>$1 = $2 - $3</code></td>
</tr>
<tr>
<td><code>add imm unsigned</code></td>
<td><code>$1,$2,100</code></td>
<td><code>$1 = $2 + 100</code></td>
</tr>
<tr>
<td><code>set less than</code></td>
<td><code>$1,$2,$3</code></td>
<td><code>$1 = ($2 &lt; $3)</code></td>
</tr>
<tr>
<td><code>set less than imm</code></td>
<td><code>$1,$2,100</code></td>
<td><code>$1 = ($2 &lt; 100)</code></td>
</tr>
<tr>
<td><code>set less than uns</code></td>
<td><code>$1,$2,$3</code></td>
<td><code>$1 = ($2 &lt; $3)</code></td>
</tr>
<tr>
<td><code>set lt. imm. uns.</code></td>
<td><code>$1,$2,100</code></td>
<td><code>$1 = ($2 &lt; 100)</code></td>
</tr>
</tbody>
</table>

Note: Immediates are sign-extended to form constant for arithmetic operations.
## MIPS Multiply/Divide

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 \times $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>Multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 \times $3</td>
<td>64-bit unsigned prod.</td>
</tr>
<tr>
<td>Divide</td>
<td>div $2,$3</td>
<td>Lo = $2 \div $3, Hi = $2 \mod $3</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td>Divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 \div $3, Hi = $2 \mod $3</td>
<td>Unsigned quotient, Unsigned remainder</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Get copy of Lo</td>
</tr>
</tbody>
</table>

### Rationale

- Deal with 64-bit result
- Simplify handling of instruction
# MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ^$3</td>
<td>Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~($2</td>
<td>$3)</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND w. constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1, $2,10</td>
<td>$1 = $2 ^10</td>
<td>Logical XOR w. constant</td>
</tr>
<tr>
<td>shift left log</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right log</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arith</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left log var</td>
<td>sllv $1,$2,$3</td>
<td>$1 = $2 &lt;&lt; $3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right log var</td>
<td>srlv $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arith</td>
<td>srav $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right arith. by var</td>
</tr>
<tr>
<td>load upper imm</td>
<td>lui $1,40</td>
<td>$1 = 40 &lt;&lt; 16</td>
<td>Put imm in upper 16 bits</td>
</tr>
</tbody>
</table>
MIPS Memory Access

All memory access through loads and stores

Aligned words, halfwords, and bytes
  • A halfword or byte loaded from memory can be sign- or zero-extended to form a word in the destination register

Floating-point loads/stores for FP registers

Single addressing mode (displacement or based)
  16-bit sign-extended displacement (immediate field)

In addition:
  • Displacement = 0 uses register contents as address
  • Register = 0 uses 16-bit displacement as address
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>store word</td>
<td>sw $3, 8($4)</td>
<td>Mem[$4+8]=$3</td>
<td>Store word</td>
</tr>
<tr>
<td>store halfword</td>
<td>sh $3, 6($2)</td>
<td>Mem[$2+6]=$3</td>
<td>Stores only lower 16 bits</td>
</tr>
<tr>
<td>bitstore byte</td>
<td>sb $2, 7($3)</td>
<td>Mem[$3+7]=$3</td>
<td>Stores only lowest byte</td>
</tr>
<tr>
<td>store float</td>
<td>sf $f2, 4($2)</td>
<td>Mem[$2+4]=$f2</td>
<td>Store FP word</td>
</tr>
<tr>
<td>load word</td>
<td>lw $1, 8($2)</td>
<td>$1=Mem[8+$2]</td>
<td>Load word</td>
</tr>
<tr>
<td>load halfword</td>
<td>lh $1, 6($3)</td>
<td>$1=Mem[6+$3]</td>
<td>Load half; sign extend</td>
</tr>
<tr>
<td>load half unsigned</td>
<td>lhu $1, 6($3)</td>
<td>$1=Mem[6+$3]</td>
<td>Load half; zero extend</td>
</tr>
<tr>
<td>load byte</td>
<td>lb $1, 5($3)</td>
<td>$1=Mem[5+$3]</td>
<td>Load byte; sign extend</td>
</tr>
<tr>
<td>load byte unsigned</td>
<td>lbu $1, 5($3)</td>
<td>$1=Mem[5+$3]</td>
<td>Load byte; zero extend</td>
</tr>
<tr>
<td>load float</td>
<td>lf $f1, 4($3)</td>
<td>$f1=Mem[4+$3]</td>
<td>Load FP register</td>
</tr>
</tbody>
</table>
Forming a Memory Address

Let’s say you want to load a value from a fixed location in memory, known at compile time.

Address: 0x123450

- `lui $1, 0x12`  # $1 = upper 16 bits of constant
- `addiu $1, $1, 0x3450`  # add in lower 16 bits
- `lw $2, 0($1)`  # perform the load

Not taking advantage of displacement capability

- `lui $1, 0x12`  # $1 = upper 16 bits of constant
- `lw $2, 0x3450($1)`  # perform the load
MIPS Branch/Jump Instructions

Two classes:
- Jumps
  - Unconditional, not PC-relative
  - For procedure call, unconditional control, switch statements, simulating long branches
- Branches
  - Conditional and PC relative
  - For conditional control and PC-relative unconditional

Jumps

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>PC = 40000</td>
<td>jump to address</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>PC = $31</td>
<td>jump to addr in register</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; PC = 40000</td>
<td>Save PC next instruction, jump to address</td>
</tr>
</tbody>
</table>
**MIPS Branches**

Conditional branch is compare-and-branch

- **Conditions:**
  - Comparison against 0: equality, sign-test
  - Comparison of two registers: equality only
  - Remaining set of compare-and-branch take two instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch equal</td>
<td>beq $1,$2,100</td>
<td>if ($1 == $2) PC=PC+4+400</td>
</tr>
<tr>
<td>branch not eq</td>
<td>bne $1,$2,100</td>
<td>if ($1 != $2) PC=PC+4+400</td>
</tr>
<tr>
<td>branch l.t. 0</td>
<td>bltz $1,100</td>
<td>if ($1 &lt; 0) PC = PC+4+400</td>
</tr>
<tr>
<td>branch g.t./eq 0</td>
<td>bgez $1,100</td>
<td>if ($1 &gt;= 0) PC = PC+4+400</td>
</tr>
</tbody>
</table>

CDA 4150 – MIPS ISA
Programming Example: Searching

C source code:
```c
count=0;
for (index=head; index<=n; index++)
    if (C[index] == target) count ++;
```

MIPS assembly code, assuming:
- count in $5, index in $6, head in $1, addr of C in $2, target in $3; n in $4
- Straightforward, but not best (smallest or fastest) code!

```assembly
li $5, 0 # set count =0 (addiu $5,$0,0)
move $6,$1 # initial index (addu $6,$1,$0)
loop: slt $9,$4,$6 # $9=1 if n < index (index > n)
bne $9,$0,exit # if index>n goto exit label
sll $7,$6,2 # multiply index by 4
addu $7,$7,$2 # address of C [index]
lw $8,0($7) # C[index] = $8
bne $8,$3,next # test if equal
addiu $5,$5,1 # increment count
next: addiu $6,$6,1 # increment index
j loop # unconditional jump to loop
exit:
```

CDA 4150 – MIPS ISA
Stacks

Stacking of Subroutine Calls & Returns and Environments

Stacks are a natural structure for procedure calls / local variables

Implementing the stack

- Common rules are needed across procedures
- Recent machines use software convention
- Some earlier machines use hardware mechanisms and instructions

CDA 4150 – MIPS ISA
Stack Frames

How is empty stack represented?

Next Empty?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last Full?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SP

Stacks can grow up or down:

inf.  \[\text{grows up}\]  Inf.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Addresses

Down/Next Empty (MIPS)

POP: Increment (SP)
PUSH: Write to Mem(SP)
Read from Mem(SP)
Decrement SP

Single elements are not usually pushed/popped. Instead an entire stack frame may be pushed or popped in one increment/decrement.
Parameters are passed in registers; extras on stack

Compilers try to keep scalar variables in registers, not memory
- Stack locations for spilling/saving on procedure calls
MIPS Addressing Model

Local scalar variables of a procedure stored in registers
- Spilled to stack frame
- Space allocated when compiled
- Loaded/stored into registers as needed

Global static scalar variables (single variables, not arrays)
- Allocated in a 64KB static area at compile time
- Addressed with a register pointing into area + offset

Dynamic allocated in heap, reserved memory below stack

CDA 4150 – MIPS ISA
Stack and dynamic area grow towards one another to maximize storage use before collision.
### MIPS Software Register Convention

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>$1</td>
<td>$at reserved for assembler</td>
</tr>
<tr>
<td>$2</td>
<td>$v0 expression evaluation &amp;</td>
</tr>
<tr>
<td>$3</td>
<td>$v1 function results</td>
</tr>
<tr>
<td>$4</td>
<td>$a0 arguments (caller saves)</td>
</tr>
<tr>
<td>$5</td>
<td>$a1</td>
</tr>
<tr>
<td>$6</td>
<td>$a2</td>
</tr>
<tr>
<td>$7</td>
<td>$a3</td>
</tr>
<tr>
<td>$8</td>
<td>$t0 temporary: caller saves</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$15</td>
<td>$t7</td>
</tr>
<tr>
<td>$16</td>
<td>$s0 callee saves</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$23</td>
<td>$s7</td>
</tr>
<tr>
<td>$24</td>
<td>$t8 temporary (cont’d)</td>
</tr>
<tr>
<td>$25</td>
<td>$t9</td>
</tr>
<tr>
<td>$26</td>
<td>$k0 reserved for OS kernel</td>
</tr>
<tr>
<td>$27</td>
<td>$k1</td>
</tr>
<tr>
<td>$28</td>
<td>$gp global pointer</td>
</tr>
<tr>
<td>$29</td>
<td>$sp stack pointer</td>
</tr>
<tr>
<td>$30</td>
<td>$fp frame pointer</td>
</tr>
<tr>
<td>$31</td>
<td>$ra Return Address (HW)</td>
</tr>
</tbody>
</table>

CDA 4150 – MIPS ISA
## MIPS Register Saving Convention

<table>
<thead>
<tr>
<th>Preserved on Call</th>
<th>Not Preserved on Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved Registers ($s0..$s7)</td>
<td>Argument Registers ($a0..$a3)</td>
</tr>
<tr>
<td>Stack Pointer ($sp)</td>
<td>Return Value Regs ($v0..$v1)</td>
</tr>
<tr>
<td>Frame Pointer ($fp)</td>
<td>Temporaries ($t0..$t9)</td>
</tr>
<tr>
<td>Return Address ($ra)</td>
<td></td>
</tr>
<tr>
<td>Global Pointer ($gp)</td>
<td></td>
</tr>
</tbody>
</table>

Preserved registers must be saved and restored by called procedure if modified
Unpreserved registers must be saved by caller if needed after call completes
MIPS Calling Convention

**Caller**
- Save caller-saved registers: $a0–$a3, $v0–$v1 $t0–$t9 if used
- Load arguments: first four in $a0–$a3, rest on stack
- Execute jal instruction

**Callee**
- Allocate memory in frame: $sp = $sp – frame size
- Save callee-saved registers $s0–$s7,$fp,$ra if used
- Create frame: $fp = $sp + frame size - 4

**Return**
- Place return value in $v0
- Restore any callee-saved registers
- Pop stack: $sp = $sp+frame size
- Return by jr $ra

Only required to do what is needed!

CDA 4150 – MIPS ISA
MIPS Calling Convention Steps

Before call:

- First four arguments passed in registers
  - FP, SP

Callee set-up:

- Adjust SP
  - Step 1:
    - FP, SP, ra

- Save registers as needed
  - Step 2:
    - FP, SP, ra
    - Old FP, $s0-$s7

- Adjust FP
  - Step 3:
    - FP, SP, ra
    - Old FP, $s0-$s7

CDA 4150 – MIPS ISA
MIPS Calling Convention Example

```c
int fact (int n)
{
    if (n <= 1)
        return (1);
    else
        return(n*fact(n-1));
}
```

```mips
fact:    slti $t0,$a0,2    ; test n < 2
        beq $t0,$0,skip
        li $v0,1    ; return value
        jr $ra      ; return
skip:    addiu $sp,$sp,-32 ; create frame
        sw $ra,20($sp) ; save $ra
        sw $fp,16($sp) ; save $fp
        addiu $fp,$sp,28 ; set $fp
        sw $a0,0($fp) ; save n
        addiu $a0,$a0,-1 ; n-1
        jal fact
        lw $a0,0($fp) ; restore n
        mult $v0,$v0,$a0 ; n*fact(n-1)
        lw $ra,20($sp) ; restore $ra
        lw $fp,16($sp) ; restore $fp
        addiu $sp,$sp,32 ; pop stack
        jr $ra        ; return
```

CDA 4150 – MIPS ISA
MIPS Instruction Encoding (I)

3 formats, all 32 bits in length

Fixed 6-bit opcode begins each instruction

ALU Format (also R format): one opcode
  • Register-register ALU instructions
    
    #bits: 6  5  5  5  5  6
    
    | opcode=0 | rs | rt | rd | sa | func |

Function code
  • Detailed opcode: add, sub, or, and, …
MIPS Instruction Encoding (II)

Immediate instruction format (I format)

- Loads/stores (incl. floating point) sign-extend imm
- Immediate instructions (e.g. addi, lui, etc.)
  - Sign-extend immediate for arithmetic ops (even addu)
  - Zero-extend for logical ops
- Branches sign-extend immediate and scale by 4
  - Add displacement to PC+4
- Different opcode for each instruction

<table>
<thead>
<tr>
<th>bits:</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

First source or base register
Second source or result register

CDA 4150 – MIPS ISA
Jump format (J format)

- Used for j, jal
- 26-bit offset is scaled by 4 to form 28 lsbs of new PC
  - 4 msbs of new PC copied from current PC

```
#bits: 6 26

<table>
<thead>
<tr>
<th>opcode</th>
<th>jump target</th>
</tr>
</thead>
</table>
```

“pseudo-direct” jump target address
MIPS ISA Details (I)

Register 0 is *always* 0 (even if you try to write it)

Jump and link (jal) puts the return address (PC+8) into the link register (R31)

All insts change *all 32 bits* of the dest register
  - Including lui, lb, lh

All read all 32 bits of sources (and, sub, and, or, …)

Data from sub-word loads extended as follows
  - lbu, lhu, zero-extended
  - lb, lh, sign-extended
The MIPS architecture defines a

- Branch delay slot
  - Instruction after branch is always executed
- Load delay slot
  - Value returned from load cannot be used the next cycle

The reason for restrictions will be clear next week

Makes perfect sense for simple in-order pipelined machines

Architecture definition

- Every implementation must obey these rules
- Branch delay slot is a burden for the R10000+
- Load delay slot is unnecessary
MIPS Summary

Reduced Instruction Set Computing (RISC) vs. Complex Instruction Set Computing (CISC)

• Terms coined by Patterson and Ditzel (1980)
• Widely-used terms, poorly defined
  – “A RISC processor is any with an instruction set defined after 1980”

Common attributes

• Fixed-length instructions
  – Some embedded processors use variable length instructions to reduce cost
• Load/store architecture (for memory accesses)
• “Large” general-purpose register file (>=32)
• “Simple” operations that can be directly controlled
• One register is hardwired to 0