**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: \( M \times 2^E \)
- single precision
- double precision
**MIPS Storage Model**

$2^{32}$ 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 diagram]

FP registers are paired for double-precision.
Specify the even register, which holds the less-significant word.

CDA 4150 – MIPS ISA

---

**MIPS Computational Instructions**

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

CDA 4150 – MIPS ISA
**MIPS Integer Arithmetic**

- **add**: $1 = $2 + $3  
  3 operands; exceptions
- **subtract**: $1 = $2 − $3  
  3 operands; exceptions
- **add immediate**: $1 = $2 + 100  
  + constant; exceptions
- **add unsigned**: $1 = $2 + $3  
  3 operands; no exceptions
- **subtract unsigned**: $1 = $2 − $3  
  3 operands; no exceptions
- **add imm unsigned**: $1 = $2 + 100  
  + constant; no exceptions
- **set less than signed**: $1 = ($2 < $3)  
  compare signed <
- **set less than unsigned**: $1 = ($2 < 100)  
  compare unsigned < constant
- **set less than uns signed**: $1 = ($2 < $3)  
  compare unsigned <
- **set l. t. imm. uns signed**: $1 = ($2 < 100)  
  compare unsigned < const

*Note: Immediates are sign-extended to form constant for arithmetic operations*

**MIPS Multiply/Divide**

- **multiply**: Hi, $1 = $2 x $3  
  64-bit signed product
- **multiply unsigned**: Hi, $1 = $2 x $3  
  64-bit unsigned prod.
- **divide**: Hi = $2 mod $3, Lo = $2 + $3, \( \text{Hi} = \text{remainder} \)
- **divide unsigned**: Hi = $2 mod $3, Lo = $2 + $3, \( \text{Hi} = \text{remainder} \)

*Move from Hi:* $1 = Hi  
Get copy of Hi

*Move from Lo:* $1 = Lo  
Get copy of Lo

**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>$1 = 2 \mid 3$</td>
<td>Logical OR</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1$, $2$, $3$</td>
<td>$1 = 2 \bxor 3$</td>
<td>Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1$, $2$, $3$</td>
<td>$1 = \neg(2 \mid 3)$</td>
<td>Logical NOR</td>
</tr>
<tr>
<td>and immediate</td>
<td>and $1$, $2$, $10$</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>$1 = 2 \mid 10$</td>
<td>Logical OR w. constant</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xor $1$, $2$, $10$</td>
<td>$1 = 2 \bxor 10$</td>
<td>Logical XOR w. constant</td>
</tr>
<tr>
<td>shift left log</td>
<td>sll $1$, $2$, $10$</td>
<td>$1 = 2 \ll 10$</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right log</td>
<td>srl $1$, $2$, $10$</td>
<td>$1 = 2 \gg 10$</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arith</td>
<td>sra $1$, $2$, $10$</td>
<td>$1 = 2 \gg 10$</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left log var</td>
<td>slv $1$, $2$, $3$</td>
<td>$1 = 2 \ll 3$</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right log var</td>
<td>srvl $1$, $2$, $3$</td>
<td>$1 = 2 \gg 3$</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arith</td>
<td>srav $1$, $2$, $3$</td>
<td>$1 = 2 \gg 3$</td>
<td>Shift right arith. by var</td>
</tr>
<tr>
<td>and upper imm</td>
<td>lui $1$, $40$</td>
<td>$1 = 40 \ll 16$</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)

+ register = memory address

In addition:
- Displacement = 0 uses register contents as address
- Register = 0 uses 16-bit displacement as address
### MIPS Load/Store Instructions

<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 bits</td>
<td>sh $3, 6($2)</td>
<td>Mem[$2+6]=$3</td>
<td>Stores only lower 16</td>
</tr>
<tr>
<td>store 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>lub $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</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

Instruction | Example | Meaning |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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>
</tbody>
</table>

branch l.t. 0  bltz $1,100  if ($1 < 0) PC = PC+4+400
branch g.t./eq 0 bgez $1,100  if ($1 >= 0) PC = PC+4+400
**Programming Example: Searching**

C source code:
```
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
- 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:
```

- Straightforward, but not best (smallest or fastest) code!

---

**Stacks**

Stacking of Subroutine Calls & Returns and Environments

- A: CALL B
  - B: CALL C
    - C: RET
    - RET

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
### Stack Frames

**How is empty stack represented?**

<table>
<thead>
<tr>
<th>Next Empty?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Stacks can grow up or down:**

<table>
<thead>
<tr>
<th>Memory Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf.</td>
</tr>
<tr>
<td>grows up</td>
</tr>
<tr>
<td>grows down</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**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.

---

### Stack Frame Layout

**Higher Addresses**

<table>
<thead>
<tr>
<th>Arguments in reverse order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved registers including old FP, SP</td>
</tr>
<tr>
<td>Local variables and temporaries</td>
</tr>
</tbody>
</table>

**Lower Addresses**

<table>
<thead>
<tr>
<th>Arguments for next procedure</th>
</tr>
</thead>
</table>

Access arguments at fixed offset from FP. Allow variable number of arguments.

Access local variables and saved registers at fixed (positive) offset from SP.

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

CDA 4150 – MIPS ISA
**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

```
| global scalars | x |
```

Offset (determined at link time)

Dynamic allocated in heap, reserved memory below stack

**MIPS Address Map**

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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>$at</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0</td>
<td>expression evaluation &amp; function results</td>
</tr>
<tr>
<td>$a0</td>
<td>arguments (caller saves)</td>
</tr>
<tr>
<td>$a1</td>
<td></td>
</tr>
<tr>
<td>$a2</td>
<td></td>
</tr>
<tr>
<td>$a3</td>
<td></td>
</tr>
<tr>
<td>$t0</td>
<td>temporary: caller saves</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>$t7</td>
<td></td>
</tr>
<tr>
<td>$s0</td>
<td>callee saves</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>$s7</td>
<td></td>
</tr>
<tr>
<td>$t8</td>
<td>temporary (cont’d)</td>
</tr>
<tr>
<td>$t9</td>
<td></td>
</tr>
<tr>
<td>$k0</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$k1</td>
<td></td>
</tr>
<tr>
<td>$gp</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>Return Address (HW)</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 Register Saving Convention

- **Preserved on Call**
  - Saved Registers: ($s0 .. $s7)
  - Stack Pointer: ($sp)
  - Frame Pointer: ($fp)
  - Return Address: ($ra)
  - Global Pointer: ($gp)

- **Preserved on Call**
  - Argument Registers: ($a0 .. $a3)
  - Return Value Registers: ($v0 .. $v1)
  - Temporaries: ($t0 .. $t9)

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

---

**MIPS Calling Convention Steps**

**Before call:**
- First four arguments passed in registers

**Callee set-up:**
- Adjust SP
  - step 1
  - Save registers as needed
  - step 2
  - Adjust FP
  - step 3

CDA 4150 – MIPS ISA
MIPS Calling Convention Example

```assembly
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
```

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

<table>
<thead>
<tr>
<th>#bits</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode=0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>func</td>
<td></td>
</tr>
</tbody>
</table>

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

**MIPS Instruction Encoding (III)**

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

<table>
<thead>
<tr>
<th>#bits:</th>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>jump target</td>
<td></td>
</tr>
</tbody>
</table>

"pseudo-direct" jump target address

CDA 4150 – MIPS ISA
**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, lh, zero-extended
  - lb, lh, sign-extended

**MIPS ISA Details (II)**

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