Computer Architecture

The Language of the Machine
Instruction Sets

- Basic ISA
- Classes, Addressing, Format
- Administrative Matters
- Operations, Branching, Calling conventions
- Break
Organization
° All computers consist of five components
  • Processor: (1) datapath and (2) control
  • (3) Memory
  • (4) Input devices and (5) Output devices
° Not all “memory” are created equally
  • Cache: fast (expensive) memory are placed closer to the processor
  • Main memory: less expensive memory--we can have more
° Input and output (I/O) devices have the messiest organization
  • Wide range of speed: graphics vs. keyboard
  • Wide range of requirements: speed, standard, cost ...
  • Least amount of research (so far)
Summary: Computer System Components

- Proc
- Caches
- Memory
- Busses
- Controllers
- Disks
- Displays
- Keyboards
- Networks

° All have interfaces & organizations
Review: Instruction Set Design

Which is easier to change?
Instruction Set Architecture: What Must be Specified?

- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
  - fetch-decode-execute is implicit!
Basic ISA Classes

Accumulator (1 register):

1 address   add A   acc ← acc + mem[A]
1+x address addx A   acc ← acc + mem[A + x]

Stack:

0 address   add       tos ← tos + next

General Purpose Register:

2 address   add A B   EA(A) ← EA(A) + EA(B)
3 address   add A B C  EA(A) ← EA(B) + EA(C)

Load/Store:

3 address   addRa Rb Rc   Ra ← Rb + Rc
             load Ra Rb    Ra ← mem[Rb]
             store Ra Rb   mem[Rb] ← Ra

Comparison:

Bytes per instruction?  Number of Instructions?  Cycles per instruction?
Comparing Number of Instructions

- Code sequence for \( C = A + B \) for four classes of instruction sets:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store C,R3</td>
</tr>
</tbody>
</table>
General Purpose Registers Dominate

- 1975-1995 all machines use general purpose registers

- Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g., \((A*B) - (C*D) - (E*F)\) can do multiplies in any order vs. stack
  - registers can hold variables
    - memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)
Summary on Instruction Classes

° Expect new instruction set architecture to use
° general purpose register

Pipelining => Expect it to use load store variant of GPR ISA
RISC features

° Reduced Instruction Set
° General Purpose Register File (large number: 32 or more)
° Load/Store Architecture
° Few Addressing modes
° Fixed Instruction Format
MIPS I Registers

° Programmable storage
  • $2^{32}$ x bytes of memory
  • 31 x 32-bit GPRs (R0 = 0)
  • 32 x 32-bit FP regs (paired DP)
  • HI, LO, PC
Memory Addressing

° Since 1980 almost every machine uses addresses to level of 8-bits (byte)

° 2 questions for design of ISA:
  • Since could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address, how do byte addresses map onto words?
  • Can a word be placed on any byte boundary?
Addressing Objects: Endianness and Alignment

- **Big Endian**: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian**: address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

**Alignment**: require that objects fall on address that is multiple of their size.

```
binary          big endian byte 0
| 3 | 2 | 1 | 0 |
msb            Aligned

little endian byte 0
| 3 | 2 | 1 | 0 |
lwb

| 0 | 1 | 2 | 3 |
Aligned
```
### Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>(R4 \leftarrow R4 + R3)</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>(R4 \leftarrow R4 + 3)</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>(R4 \leftarrow R4 + \text{Mem}[100+R1])</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>(R4 \leftarrow R4 + \text{Mem}[R1])</td>
</tr>
<tr>
<td>Indexed / Base</td>
<td>Add R3,(R1+R2)</td>
<td>(R3 \leftarrow R3 + \text{Mem}[R1+R2])</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>(R1 \leftarrow R1 + \text{Mem}[1001])</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>(R1 \leftarrow R1 + \text{Mem}[\text{Mem}[R3]])</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>Add R1,(R2)+</td>
<td>(R1 \leftarrow R1 + \text{Mem}[R2]; R2 \leftarrow R2 + d)</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1,–(R2)</td>
<td>(R2 \leftarrow R2 - d; R1 \leftarrow R1 + \text{Mem}[R2])</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>(R1 \leftarrow R1 + \text{Mem}[100+R2+R3*d])</td>
</tr>
</tbody>
</table>

*Why Auto-increment/decrement? Scaled?*
MIPS Addressing Formats (Summary)

How memory can be addressed in MIPS

1. Immediate addressing
   - op
   - rs
   - rt
   - Immediate

2. Register addressing
   - op
   - rs
   - rt
   - rd
   - funct
   - Registers
   - Register

3. Base addressing
   - op
   - rs
   - rt
   - Address
   - Register
   - Memory
   - Byte
   - Halfword
   - Word

4. PC-relative addressing
   - op
   - rs
   - rt
   - Address
   - PC
   - Memory
   - Word

5. Pseudodirect addressing
   - op
   - Address
   - PC
   - Memory
   - Word
Addressing Mode Usage? (ignore register mode)
3 programs measured on machine with all address modes (VAX)

--- Displacement: 42% avg, 32% to 55%
--- Immediate: 33% avg, 17% to 43%
--- Register deferred (indirect): 13% avg, 3% to 24%
--- Scaled: 7% avg, 0% to 16%
--- Memory indirect: 3% avg, 1% to 6%
--- Misc: 2% avg, 0% to 3%

75% displacement & immediate
88% displacement, immediate & register indirect
Displacement Address Size?

- Avg. of 5 SPECint92 programs v. avg. 5 SPECfp92 programs
- X-axis is in powers of 2: 4 => addresses > $2^3$ (8) and $3 \times 2^4$ (16)
- 1% of addresses > 16-bits
- 12 - 16 bits of displacement needed
Immediate Size?

• 50% to 60% fit within 8 bits
• 75% to 80% fit within 16 bits
Addressing Summary

• Data Addressing modes that are important: Displacement, Immediate, Register Indirect

• Displacement size should be 12 to 16 bits

• Immediate size should be 8 to 16 bits
Generic Examples of Instruction Format Widths

Variable:  

Fixed:  

Hybrid:  

Summary of Instruction Formats

• If code size is most important, use variable length instructions

• If performance is over is most important, use fixed length instructions

• Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density
Instruction Format

• If have many memory operands per instructions and many addressing modes, => Address Specifier per operand

• If have load-store machine with 1 address per instr. and one or two addressing modes, => encode addressing mode in the opcode
# MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

**Register (direct)**  
\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{rd} \\
\end{array}
\]

- Immediate  
\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{immed} \\
\end{array}
\]

- Base+index  
\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{immed} \\
\end{array}
\]

- PC-relative  
\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{immed} \\
\end{array}
\]

- Register Indirect?
Typical Operations (little change since 1960)

<table>
<thead>
<tr>
<th>Data Movement</th>
<th>Load (from memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store (to memory)</td>
<td></td>
</tr>
<tr>
<td>memory-to-memory move</td>
<td></td>
</tr>
<tr>
<td>register-to-register move</td>
<td></td>
</tr>
<tr>
<td>input (from I/O device)</td>
<td></td>
</tr>
<tr>
<td>output (to I/O device)</td>
<td></td>
</tr>
<tr>
<td>push, pop (to/from stack)</td>
<td></td>
</tr>
</tbody>
</table>

| Arithmetic                     | integer (binary + decimal) or FP |
|                                | Add, Subtract, Multiply, Divide  |

| Shift                          | shift left/right, rotate left/right |

| Logical                        | not, and, or, set, clear            |

| Control (Jump/Branch)          | unconditional, conditional          |

| Subroutine Linkage             | call, return                        |

| Interrupt                     | trap, return                         |

| Synchronization               | test & set (atomic r-m-w)            |

| String                        | search, translate                    |

| Graphics (MMX)                | parallel subword ops (4 16bit add)   |
Top 10 80x86 Instructions

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer</th>
<th>Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td></td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td></td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td></td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td></td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td></td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td></td>
<td></td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td></td>
<td></td>
<td>1%</td>
</tr>
</tbody>
</table>

Total 96%

° Simple instructions dominate instruction frequency
Operation Summary

• Support these simple instructions, since they will dominate the number of instructions executed:

  load,
  store,
  add,
  subtract,
  move register-register,
  and,
  shift,
  compare equal, compare not equal,
  branch,
  jump,
  call,
  return;
Compilers and Instruction Set Architectures

• **Ease of compilation**
  ° orthogonality: no special registers, few special cases, all operand modes available with any data type or instruction type
  ° completeness: support for a wide range of operations and target applications
  ° regularity: no overloading for the meanings of instruction fields
  ° streamlined: resource needs easily determined

• **Register Assignment is critical too**
  ° Easier if lots of registers
Summary of Compiler Considerations

• Provide at least 16 general purpose registers plus separate floating-point registers,

• Be sure all addressing modes apply to all data transfer instructions,

• Aim for a minimalist instruction set.
MIPS I Operation Overview

° Arithmetic logical
  ° Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
  ° AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI
  ° SLL, SRL, SRA, SLLV, SRLV, SRAV
° Memory Access
  ° LB, LBU, LH, LHU, LW, LWL, LWR
  ° SB, SH, SW, SWL, SWR
Multiply / Divide

° Start multiply, divide
  • MULT rs, rt
  • MULTU rs, rt
  • DIV rs, rt
  • DIVU rs, rt

° Move result from multiply, divide
  • MFHI rd
  • MFLO rd

° Move to HI or LO
  • MTHI rd
  • MTLO rd

° Why not Third field for destination?
  (Hint: how many clock cycles for multiply or divide vs. add?)
Data Types

**Bit:** 0, 1

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

**Character:**
- ASCII 7 bit code

**Decimal:**
- digits 0-9 encoded as 0000b thru 1001b
- two decimal digits packed per 8 bit byte

**Integers:**
- 2's Complement

**Floating Point:**
- Single Precision
- Double Precision
- Extended Precision

\[ M \times R^E \]
- mantissa
- exponent
- base

How many +/- #'s?
Where is decimal pt?
How are +/- exponents represented?
Operand Size Usage

- Doubleword: 0% (Int Avg.), 69% (FP Avg.)
- Word: 19% (Int Avg.), 31% (FP Avg.), 74% (FP Avg.)
- Halfword: 0% (Int Avg.), 7% (FP Avg.)
- Byte: 0% (Int Avg.), 0% (FP Avg.)

Frequency of reference by size

- Support these data sizes and types:
  - 8-bit, 16-bit, 32-bit integers
  - 32-bit and 64-bit IEEE 754 floating point numbers
## MIPS arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>

Which add for address arithmetic? Which add for integers?
# 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>3 reg. operands; 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>3 reg. operands; 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 reg, 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 &amp;~10</td>
<td>Logical XOR reg, constant</td>
</tr>
<tr>
<td>shift left logical</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 logical</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 arithm.</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 logical</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 logical</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 arithm.</td>
<td>srav $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>
# MIPS data transfer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW 500(R4), R3</td>
<td>Store word</td>
</tr>
<tr>
<td>SH 502(R2), R3</td>
<td>Store half</td>
</tr>
<tr>
<td>SB 41(R3), R2</td>
<td>Store byte</td>
</tr>
<tr>
<td>LW R1, 30(R2)</td>
<td>Load word</td>
</tr>
<tr>
<td>LH R1, 40(R3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>LHU R1, 40(R3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>LB R1, 40(R3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>LBU R1, 40(R3)</td>
<td>Load byte unsigned</td>
</tr>
</tbody>
</table>

**Why need LUI?**

![LUI R5 diagram]

<table>
<thead>
<tr>
<th>R5</th>
<th>0000 ... 0000</th>
</tr>
</thead>
</table>
Methods of Testing Condition

° Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

ex: add r1, r2, r3
    bz label

° Condition Register

Ex: cmp r1, r2, r3
    bgt r1, label

° Compare and Branch

Ex: bgt r1, r2, label
Condition Codes

Setting CC as side effect can reduce the # of instructions

\[
\begin{align*}
X: & \quad . \\
& \quad . \\
& \quad \text{SUB } r0, #1, r0 \\
& \quad \text{BRP } X
\end{align*}
\]

vs.

\[
\begin{align*}
X: & \quad . \\
& \quad . \\
& \quad \text{SUB } r0, #1, r0 \\
& \quad \text{CMP } r0, #0 \\
& \quad \text{BRP } X
\end{align*}
\]

But also has disadvantages:

--- not all instructions set the condition codes; which do and which do not often confusing!
\text{e.g., shift instruction sets the carry bit}

--- dependency between the instruction that sets the CC and the one that tests it: to overlap their execution, may need to separate them with an instruction that does not change the CC

\[
\begin{array}{c|c|c|c}
\text{ifetch} & \text{read} & \text{compute} & \text{write} \\
\end{array}
\]

Old CC read

New CC computed

\[
\begin{array}{c|c|c|c}
\text{ifetch} & \text{read} & \text{compute} & \text{write} \\
\end{array}
\]
Conditional Branch Distance

- Distance from branch in instructions $2^i \Rightarrow \hat{S} \pm 2^{i-1} \& > 2^{i-2}$
- 25% of integer branches are $> 2$ to $\hat{S}$ 4 or -2 to -4 instructions
Conditional Branch Addressing

• PC-relative since most branches are relatively close to the current PC address

• At least 8 bits suggested (± 128 instructions)

• Compare Equal/Not Equal most important for integer programs (86%)

<table>
<thead>
<tr>
<th>Comparison Type</th>
<th>Int Avg.</th>
<th>FP Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LT/GE</td>
<td>7%</td>
<td>40%</td>
</tr>
<tr>
<td>GT/LE</td>
<td>7%</td>
<td>23%</td>
</tr>
<tr>
<td>EQ/NE</td>
<td>86%</td>
<td>37%</td>
</tr>
</tbody>
</table>

Frequency of comparison types in branches
MIPS Compare and Branch

° Compare and Branch
  • BEQ rs, rt, offset if R[rs] == R[rt] then PC-relative branch
  • BNE rs, rt, offset <>

° Compare to zero and Branch
  • BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
  • BGTZ rs, offset >
  • BLT <
  • BGEZ >=
  • BLTZAL rs, offset if R[rs] < 0 then branch and link (into R 31)
  • BGEZAL >=

° Remaining set of compare and branch take two instructions

° Almost all comparisons are against zero!
# MIPS jump, branch, compare instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,100</td>
<td>if ($1 == $2) go to PC+4+100 Equal test; PC relative branch</td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ($1!= $2) go to PC+4+100 Not equal test; PC relative</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0 Compare less than; 2’s comp.</td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0 Compare &lt; constant; 2’s comp.</td>
</tr>
<tr>
<td>set less than uns.</td>
<td>sltu $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0 Compare less than; natural numbers</td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>sltiu $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0 Compare &lt; constant; natural numbers</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000 Jump to target address</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31 For switch, procedure return</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000 For procedure call</td>
</tr>
</tbody>
</table>
Signed vs. Unsigned Comparison

<table>
<thead>
<tr>
<th>Value?</th>
<th>2’s comp</th>
<th>Unsigned?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1= 0...00 0000 0000 0000 0001</td>
<td>two</td>
<td></td>
</tr>
<tr>
<td>R2= 0...00 0000 0000 0000 0010</td>
<td>two</td>
<td></td>
</tr>
<tr>
<td>R3= 1...11 1111 1111 1111 1111</td>
<td>two</td>
<td></td>
</tr>
</tbody>
</table>

After executing these instructions:

- \texttt{slt r4,r2,r1}; if (r2 < r1) r4=1; else r4=0
- \texttt{slt r5,r3,r1}; if (r3 < r1) r5=1; else r5=0
- \texttt{sltu r6,r2,r1}; if (r2 < r1) r6=1; else r6=0
- \texttt{sltu r7,r3,r1}; if (r3 < r1) r7=1; else r7=0

What are values of registers r4 - r7? Why?

\[ r4 = \quad ; \quad r5 = \quad ; \quad r6 = \quad ; \quad r7 = \quad ; \]
Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:

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

Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)
Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:

Next Empty?

<table>
<thead>
<tr>
<th>Last Full?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

inf. Big  
0 Little

grows up  
grows down

Memory Addresses

How is empty stack represented?

Little --> Big/Last Full

POP: Read from Mem(SP)
Decrement SP

PUSH: Increment SP
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Increment SP
Call-Return Linkage: Stack Frames

- Reference args and local variables at fixed (positive) offset from FP
- Grows and shrinks during expression evaluation

° Many variations on stacks possible (up/down, last pushed / next)
° Block structured languages contain link to lexically enclosing frame
° Compilers normally keep scalar variables in registers, not memory!
### MIPS: Software conventions for Registers

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

Plus a 3-deep stack of mode bits.
MIPS / GCC Calling Conventions

fact:

addiu $sp, $sp, -32
sw $ra, 20($sp)
sw $fp, 16($sp)
addiu $fp, $sp, 32

... sw $a0, 0($fp)
...

lw $31, 20($sp)
lw $fp, 16($sp)
addiu $sp, $sp, 32
jr $31

low address

First four arguments passed in registers.
Details of the MIPS instruction set

° Register zero **always** has the value **zero** (even if you try to write it)
° Branch/jump **and link** put the return addr. PC+4 into the link register (R31)
° All instructions change **all 32 bits** of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
° Immediate arithmetic and logical instructions are extended as follows:
  • logical immediates ops are zero extended to 32 bits
  • arithmetic immediates ops are sign extended to 32 bits (including addu)
° The data loaded by the instructions lb and lh are extended as follows:
  • lbu, lhu are zero extended
  • lb, lh are sign extended
° Overflow can occur in these arithmetic and logical instructions:
  • add, sub, addi
  • it **cannot** occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu
### MIPS Instructions (Quick Summary)

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

### MIPS assembly language

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

- Reduced Instruction Set
- General Purpose Register File (large number: 32 or more)
- Load/Store Architecture
- Few Addressing modes
- Fixed Instruction Format
MIPS Architecture

- 32 Registers
- Load/Store Architecture
- 5 Instruction Groups: Arithmetic, Logical, Data Transfer, Cond. Branch, Uncond. Jump
- Addressing modes: Register, Displacement, Immediate and PC-relative
- Fixed Instruction Format
Registers

- General Purpose Register Set
- Any register can be used with any instruction
- MIPS programmers have agreed upon a set of guidelines that specify how each of the registers should be used. Programmers (and compilers) know that as long as they follow these guidelines, their code will work properly with other MIPS code.
## Registers

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>at</td>
<td>1</td>
<td>Reserved for the Assembler</td>
</tr>
<tr>
<td>v0 – v1</td>
<td>2 - 3</td>
<td>Result Registers</td>
</tr>
<tr>
<td>a0 – a3</td>
<td>4 - 7</td>
<td>Argument Registers 1…4</td>
</tr>
<tr>
<td>t0 – t9</td>
<td>8 – 15, 24 - 25</td>
<td>Temporary Registers 0…9</td>
</tr>
<tr>
<td>s0 – s7</td>
<td>16 - 23</td>
<td>Saved Registers 0…7</td>
</tr>
<tr>
<td>k0 – k1</td>
<td>26 - 27</td>
<td>Kernel Registers 0…1</td>
</tr>
<tr>
<td>gp</td>
<td>28</td>
<td>Global Data Pointer</td>
</tr>
<tr>
<td>sp</td>
<td>29</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>fp</td>
<td>30</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>ra</td>
<td>31</td>
<td>Return Address</td>
</tr>
</tbody>
</table>
**Instruction Format**

- **Fixed Format**
- **3 Format Types**
  - Register: R-type
  - Immediate: I-type
  - PC-relative: J-type

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
</table>

**All MIPS Instructions Format**
R-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

Used by
- Arithmetic Instructions
- Logic Instructions
- Except when Immediate Addressing mode used
## I-Type

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

### Used by

- Instructions using Immediate addressing mode
- Instructions using Displacement addressing mode
- Branch instructions
J-Type

- **Used by**
  - Jump Instructions
Instructions

- 5 Groups
  - Arithmetic
  - Logic
  - Data Transfer
  - Conditional Branch
  - Unconditional Jump
Arithmetic

° add, addu: signed and unsigned addition on registers
° addi, addiu: signed and unsigned addition. One operand is immediate value
° sub, subu: signed and unsigned subtraction on registers
° subi, subiu: signed and unsigned subtraction. One operand is immediate value
° mult, multu: signed and unsigned multiplication on registers
° div, divu: signed and unsigned division on registers
° mfc0: move from coprocessor
° mfhi, mflo: move from Hi and Lo registers
Logical

- and, andi: logical ‘AND’ on registers and registers and an immediate value
- nor, nori: logical ‘NOT OR’ on registers and registers and an immediate value
- or, ori: logical ‘OR’ on registers and registers and an immediate value
- xor, xori: logical ‘Exclusive OR’ on registers and registers and an immediate value
- sll, sra, srl: shift left/right logical/arithmetic on registers. Size of shift can be immediate value.
- slt: comparison instruction: rd ← 1/0 depending on comparison outcome
Data Transfer

° lw, sw: load/store word
° lb, sb: load/store byte
° lbu: load byte unsigned
° lh, sh: load/store halfword
° lui: load upper half word immediate
Branch

° b: branch unconditional
° beq: branch if src1 == src2
° bne: branch if src1 /= src2
° bgez: branch if src1 >= 0
° bgtz: branch if src1 > 0
° blez: branch if src1 <= 0
° bltz: branch if src1 < 0
Jump

° j: jump
° jr: jump to src1 (address in reg src1)
° jal: jump and link; ra ← PC+4; jump to label
° jalr: jump and link; ra ← PC+4; jump to src1 (address in reg src1)
Addressing Modes

- **Register:** all operands are registers
- **Immediate:** one operand is an immediate value contained in the immediate field of I-type format
- **Displacement:** The address of the operand is src1 + displacement. Also contained in the immediate field of I-type format
- **PC-relative:** The +/- displacement is sign extended and added to the PC
- **Direct Address:** used by jump instructions. The full address is provided.