Review: Organization

- All computers consist of five components
  - Processor: (1) datapath and (2) control
  - (3) Memory
  - I/O: (4) Input devices and (5) Output devices

- Not all “memory” is 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

- All have interfaces & organizations

Overview of Today’s Lecture

- Review from Last Lecture (1 min)
- Classes, Addressing, Format (20 min)
- Administrative Matters (3 min)
- Operations, Branching, Calling conventions (25 min)
- Break (5 min)
- MIPS Details, Performance (25 min)
Review: Instruction Set Design

Which is easier to change/design???

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

Most real machines are hybrids of these:

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 (can be memory/memory):

Load/Store:
3 address: add Ra 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>Store C, R3</td>
<td></td>
</tr>
</tbody>
</table>
General Purpose Registers Dominate

- 1975-1999 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)

MIPS I Registers

- Programmable storage
  - \(2^{32} \times 8\) 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: Endianess 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.
### 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 ← R4+R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 ← R4+Mem[100+R1]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 ← R4+Mem[R1]</td>
</tr>
<tr>
<td>Indexed / Base</td>
<td>Add R3,(R1+R2)</td>
<td>R3 ← R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>R1 ← R1+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,(R3)</td>
<td>R1 ← R1+Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Post-increment</td>
<td>Add R1,(R2)+</td>
<td>R1 ← R1+Mem[R2]; R2 ← R2+d</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>Add R1,–(R2)</td>
<td>R2 ← R2–d; R1 ← R1+Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 ← R1+Mem[100+R2+R3’d]</td>
</tr>
</tbody>
</table>

**Why Post-increment/Pre-decrement? Scaled?**

### 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
85% displacement, immediate & register indirect

### Displacement Address Size?

- Avg. of 5 SPECInt92 programs v. avg. 5 SPECfp92 programs
- 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:**
  - ... [Blank]
  - [Blank] [Blank] [Blank] [Blank] ... [Blank]

- **Fixed:**
  - [Red]

- **Hybrid:**
  - [Pink]

---

**Instruction Formats**

- If code size is most important, use variable length instructions
- If performance 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
- Some architectures actually exploring on-the-fly decompression for more density.

---

**Instruction Format**

- If have many memory operands per instruction and/or many addressing modes:
  - => Need one address specifier per operand
- If have load-store machine with 1 address per instr. and one or two addressing modes:
  - => Can encode addressing mode in the opcode
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

<table>
<thead>
<tr>
<th>Mode</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register (direct)</td>
<td>op    rs  rt  rd</td>
<td>register</td>
</tr>
<tr>
<td>Immediate</td>
<td>op  rs  rt  imm</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register (base-index)</td>
<td>op  rs  rt  imm</td>
<td>Memory</td>
</tr>
<tr>
<td>PC-relative</td>
<td>op  rs  rt  imm</td>
<td>Memory</td>
</tr>
</tbody>
</table>

- Register Indirect?

Administrative Matters

- CS152 news group: ucb.class.cs152 (email cs152@cory with specific questions)
- Slides and handouts available via WWW: http://www-inst.eecs.berkeley.edu/~cs152
- Sections are on Tuesdays:
  - 10:00-12:00 310 Soda
  - 12:00-2:00 45 Evans
- First few labs without partners
- Get Cory key card/card access to Cory 119
  - Homework #1 due on Wed 9/1 at beginning of lecture
  - Prerequisite quiz will also be on Wed: CS 61C, CS150
  - Lab 1 due Friday 9/3 by 5pm in box in 283 Soda Hall
  - Don’t forget to petition if you are on the waiting list!
  - Turn in survey forms with photo!

Typical Operations (little change since 1960)

- Data Movement
  - Load (from memory)
  - Store (to memory)
  - memory-to-memory move
  - register-to-register move
  - input (from I/O device)
  - output (to I/O device)
  - push, pop (to/from stack)

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

- 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
- UNICODE 16 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

### MIPS arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td><code>add $1,$2,$3</code></td>
<td>$1 = $2 + $3`</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td><strong>subtract</strong></td>
<td><code>sub $1,$2,$3</code></td>
<td>$1 = $2 – $3`</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td><strong>add immediate</strong></td>
<td><code>addi $1,$2,100</code></td>
<td>$1 = $2 + 100 + constant;</td>
<td>exception possible</td>
</tr>
<tr>
<td><strong>add unsigned</strong></td>
<td><code>addu $1,$2,$3</code></td>
<td>$1 = $2 + $3`</td>
<td>no exceptions</td>
</tr>
<tr>
<td><strong>subtract unsigned</strong></td>
<td><code>subu $1,$2,$3</code></td>
<td>$1 = $2 – $3`</td>
<td>no exceptions</td>
</tr>
<tr>
<td><strong>add imm. unsigned</strong></td>
<td><code>addiu $1,$2,100</code></td>
<td>$1 = $2 + 100 + constant;</td>
<td>no exceptions</td>
</tr>
<tr>
<td><strong>multiply</strong></td>
<td><code>mult $2,$3</code></td>
<td>Hi, Lo = $2 × $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td><strong>multiply unsigned</strong></td>
<td><code>mulu $2,$3</code></td>
<td>Hi, Lo = $2 × $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td><strong>divide</strong></td>
<td><code>div $2,$3</code></td>
<td>Lo = $2 ÷ $3, Hi = remainder</td>
<td>Hi = $2 mod $3</td>
</tr>
<tr>
<td><strong>divide unsigned</strong></td>
<td><code>divu $2,$3</code></td>
<td>Lo = $2 ÷ $3, Hi = remainder</td>
<td>Hi = $2 mod $3</td>
</tr>
<tr>
<td><strong>Move from Hi</strong></td>
<td><code>mfhi $1</code></td>
<td>$1 = Hi`</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td><strong>Move from Lo</strong></td>
<td><code>mflo $1</code></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>and $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>or $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xor $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 left arithm.</td>
<td>slt $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by variable</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>
</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>

### 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

### When does MIPS sign extend?

- **When value is sign extended, copy upper bit to full value:**
  
  Examples of sign extending 8 bits to 16 bits:
  
  00001010 ⇒ 00000000 00001010
  10001100 ⇒ 11111111 10001100

- **When is an immediate value sign extended?**
  - Arithmetic instructions (add, sub, etc.) sign extend immediates even for the unsigned versions of the instructions!
  - Logical instructions do not sign extend

- **Load/Store half or byte do sign extend, but unsigned versions do not.**
Conditional Branch Distance

- 25% of integer branches are 2 to 4 instructions

Conditional Branch Addressing

- PC-relative since most branches are relatively close to the current PC
- At least 8 bits suggested (±128 instructions)
- Compare Equal/Not Equal most important for integer programs (86%)

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 ops 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 if ($1 == $2) go to PC+4+100</td>
<td></td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100 if ($1 != $2) go to PC+4+100</td>
<td></td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3 if ($2 &lt; $3) $1=1; else $1=0</td>
<td></td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100 if ($2 &lt; 100) $1=1; else $1=0</td>
<td></td>
</tr>
<tr>
<td>set less than uns.</td>
<td>sltiu $1,$2,$3 if ($2 &lt; 100) $1=1; else $1=0</td>
<td></td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>sltiu $1,$2,100 if ($2 &lt; 100) $1=1; else $1=0</td>
<td></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 $31 = PC + 4; go to 10000</td>
<td></td>
</tr>
</tbody>
</table>
Signed vs. Unsigned Comparison

R1= 0...00 0000 0000 0000 0001_
R2= 0...00 0000 0000 0000 0010_
R3= 1...11 1111 1111 1111 1111_

- After executing these instructions:
  - slt r4, r2, r1; if (r2 < r1) r4=1; else r4=0
  - slt r5, r3, r1; if (r3 < r1) r5=1; else r5=0
  - sltu r6, r2, r1; if (r2 < r1) r6=1; else r6=0
  - sltu r7, r3, r1; if (r3 < r1) r7=1; else r7=0

- What are values of registers r4 - r7? Why?
  - r4 =
  - r5 =
  - r6 =
  - r7 =

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:

SP | Last Full? | c | b | a
  | Next Empty? | inf. Big grows up | 0 Little grows down | Memory Addresses

How is empty stack represented?
- Little --> Big/Last Full
- POP: Read from Mem(SP)
- PUSH: Increment SP

Call-Return Linkage: Stack Frames

ARGS
Callee Save Registers
(old FP, RA)
Local Variables

FP
Grows and shrinks during expression evaluation

Hi Mem

Ref args and local variables at fixed (positive) offset from FP

Lo Mem

- Many variations on stacks possible (up/down, last pushed / next )
- Compilers normally keep scalar variables in registers, not memory!

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:

SP
0 Little inf. Big

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

Sometimes stacks are implemented via software convention (e.g., MIPS)
### MIPS: Software conventions for Registers

- **Register zero** always has the value 0 (even if you try to write it)
- Branch/jump and link put the return addr. PC+4 or 8 into the link register (R31) (depends on logical vs physical architecture)
- 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 / GCC Calling Conventions

- First four arguments passed in registers.

### Delayed Branches

- In the “Raw” MIPS, the instruction after the branch is executed even when the branch is taken:
  - This is hidden by the assembler for the MIPS “virtual machine”
  - allows the compiler to better utilize the instruction pipeline (???)
**Branch & Pipelines**

By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?

---

**Filling Delayed Branches**

```plaintext
Branch:    Inst Fetch    Iod & Op Fetch    Execute
execute successor even if branch taken!
Then branch target or continue
Single delay slot impacts the critical path

Compiler can fill a single delay slot with a useful instruction 50% of the time.
- try to move down from above jump
- move up from target, if safe
```

Is this violating the ISA abstraction?

---

**Miscellaneous MIPS I instructions**

- break: A breakpoint trap occurs, transfers control to exception handler
- syscall: A system trap occurs, transfers control to exception handler
- coprocessor instrs.: Support for floating point
- TLB instructions: Support for virtual memory: discussed later
- restore from exception: Restores previous interrupt mask & kernel/user mode bits into status register
- load word left/right: Supports misaligned word loads
- store word left/right: Supports misaligned word stores

---

**Performance**

- **Purchasing perspective**
  - given a collection of machines, which has the
    - best performance?
    - least cost?
    - best performance / cost?

- **Design perspective**
  - faced with design options, which has the
    - best performance improvement?
    - least cost?
    - best performance / cost?

- **Both require**
  - basis for comparison
  - metric for evaluation

- **Our goal is to understand cost & performance implications of architectural choices**
Two notions of “performance”

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (pmph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boeing 747</td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td>BAD/Sud Concorde</td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

Which has higher performance?

° Time to do the task (Execution Time)
   – execution time, response time, latency
° Tasks per day, hour, week, sec, ns.. (Performance)
   – throughput, bandwidth

Response time and throughput often are in opposition

Definitions

- Performance is in units of things-per-second
  - bigger is better
- If we are primarily concerned with response time
  - \[ \text{performance}(x) = \frac{1}{\text{execution time}(x)} \]

“X is n times faster than Y” means

\[ \text{Performance}(X) \]
\[ \frac{n}{\text{Performance}(Y)} \]

Example

- Time of Concorde vs. Boeing 747?
  - Concord is 1350 mph / 610 mph = 2.2 times faster
    = 6.5 hours / 3 hours

- Throughput of Concorde vs. Boeing 747?
  - Concord is 178,200 pmph / 286,700 pmph = 0.62 “times faster”
  - Boeing is 286,700 pmph / 178,200 pmph = 1.6 “times faster”

- Boeing is 1.6 times (“60%”) faster in terms of throughput
- Concord is 2.2 times (“120%”) faster in terms of flying time

We will focus primarily on execution time for a single job

Basis of Evaluation

Pros

- representative
- portable
- widely used
- improvements useful in reality
- easy to run, early in design cycle
- identify peak capability and potential bottlenecks

Actual Target Workload

Cons

- very specific
- non-portable
- difficult to run, or measure
- hard to identify cause
- less representative
- easy to “fool”

Full Application Benchmarks

Small “Kernel” Benchmarks

Microbenchmarks

“peak” may be a long way from application performance
**SPEC95**

- Eighteen application benchmarks (with inputs) reflecting a technical computing workload
- Eight integer
  - go, m88ksim, gcc, compress, li, ijpeg, perl, vortex
- Ten floating-point intensive
  - tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fppp, wave5
- Must run with standard compiler flags
  - eliminate special undocumented incantations that may not even generate working code for real programs

---

**Metrics of performance**

Each metric has a place and a purpose, and each can be misused

---

**Aspects of CPU Performance**

<table>
<thead>
<tr>
<th>CPU time =</th>
<th>Seconds</th>
<th>Instructions x Cycles x Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Program</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cycle</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instr count</th>
<th>CPI</th>
<th>clock rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. Set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Aspects of CPU Performance**

<table>
<thead>
<tr>
<th>CPU time =</th>
<th>Seconds</th>
<th>Instructions x Cycles x Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Program</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cycle</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instr count</th>
<th>CPI</th>
<th>clock rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Instr. Set</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
CPI

“Average cycles per instruction”

\[
\text{CPI} = \frac{\text{CPU Time} \times \text{Clock Rate}}{\text{Instruction Count}}
\]

\[
= \frac{\text{Clock Cycles}}{\text{Instruction Count}}
\]

\[
\text{CPU time} = \text{ClockCycleTime} \times \sum_{i=1}^{n} \text{CPI}_i \times l_i
\]

\[
\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i \times F_i \quad \text{where} \quad F_i = \frac{l_i}{\text{Instruction Count}}
\]

“Instruction frequency”

Invest Resources where time is Spent!

Example (RISC processor)

Base Machine (Reg / Reg)

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
<td>23%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
<td>45%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
<td>14%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
<td>18%</td>
</tr>
</tbody>
</table>

Typical Mix: 2.2

How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

How does this compare with using branch prediction to shave a cycle off the branch time?

What if two ALU instructions could be executed at once?

Amdahl's Law

\[
\text{Speedup due to enhancement E:}
\]

\[
\frac{\text{ExTime without E}}{\text{ExTime with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}
\]

\[
\text{Suppose that enhancement E accelerates a fraction F of the task}
\]

by a factor S and the remainder of the task is unaffected then,

\[
\text{ExTime(with E)} = ((1-F) + F/S) \times \text{ExTime(without E)}
\]

\[
\text{Speedup(with E)} = \frac{1}{(1-F) + F/S}
\]

Summary: Salient features of MIPS I

- 32-bit fixed format inst (3 formats)
- 32 32-bit GPR (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- 3-address, reg-reg arithmetic instr.
- Single address mode for load/store: base+displacement
  - no indirection, scaled
- 16-bit immediate plus LUI
- Simple branch conditions
  - compare against zero or two registers for =,≠
  - no integer condition codes
- Delayed branch
  - execute instruction after a branch (or jump) even if the branch is taken
  (Compiler can fill a delayed branch with useful work about 50% of the time)
Summary: Instruction set design (MIPS)

- Use general purpose registers with a load-store architecture: **YES**
- Provide at least 16 general purpose registers plus separate floating-point registers: **31 GPR & 32 FPR**
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred: **YES: 16 bits for immediate, displacement (disp=0 => register deferred)**
- All addressing modes apply to all data transfer instructions: **YES**
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size: **Fixed**
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: **YES**
- 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 (with a PC-relative address at least 8-bits long), jump, call, and return: **YES, 16b**
- Aim for a minimalist instruction set: **YES**

Summary: Evaluating Instruction Sets?

Design-time metrics:
- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

Static Metrics:
- How many bytes does the program occupy in memory?

Dynamic Metrics:
- How many instructions are executed?
- How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

**Best Metric:** Time to execute the program!

NOTE: this depends on instructions set, processor organization, and compilation techniques.