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
- Datapath and Control typically on on chip
- Not all “memory” is created equally
  - Cache: fast (expensive) memory close to processor
  - Main memory: less expensive memory
- 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)

Review: It’s all about communication

- All have interfaces & organizations
- Um…. It’s the network stupid??!

Review: Instruction Set Design

- software
- hardware

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!

** ISA Choices**

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

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):

3 address add A B C EA[A] ← EA[B] + EA[C]

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></th>
<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>
<td></td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store C, R1</td>
<td>Add R3,R1,R2</td>
<td>Store C,R3</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Purpose Registers Dominate

- 1975-2000 all machines use general purpose registers
- Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g., \((A \cdot B) - (C \cdot D) - (E \cdot F)\) 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)

Example: MIPS I Registers

- Programmable storage
  - \(2^{32} \times \text{bytes of memory}\)
  - 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)
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:**
  
  ![Variable Width Instruction Format](image1)

- **Fixed:**

  ![Fixed Width Instruction Format](image2)

- **Hybrid:**

  ![Hybrid Width Instruction Format](image3)

**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:
  
  \[\Rightarrow\text{Need one address specifier per operand}\]

- If have load-store machine with 1 address per instr. and one or two addressing modes:
  
  \[\Rightarrow\text{Can encode addressing mode in the opcode}\]
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

Register (direct)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Immediate

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Base+index

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PC-relative

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory

+ Memory

- Register Indirect?

Administrative Matters

- CS152 news group: ucb.class.cs152 (email cs152@cory with specific questions)
- Slides and handouts available via web: http://inst.eecs.berkeley.edu/~cs152
- Sign up to the cs152-announce mailing list:
  - Go to the “Information” page, look under “Course Operation”
  - Sections are on Thursday: Starting this Thursday (1/30)!
    - 10:00 – 12:00 310 Hearst Mining
    - 4:00 – 6:00 3 Evans
- Get Cory key card/card access to Cory 119
  - Your NT account names are derived from your UNIX “named” accounts: ‘cs152-yourUNIXname’
  - Homework #1 due Monday at beginning of lecture
    - It is up there now – really!
  - Prerequisite quiz will also be on Monday 1/30: CS 61C, CS150 Review Chapters 1-4, Ap, B of COD, Second Edition
  - Don’t forget to petition if you are on the waiting list!
    - Available on third-floor, near front office
    - Must turn in soon
- Turn in survey forms with photo before Prereq quiz!

MIPS I Operation Overview

- Arithmetic Logical:
  - Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
  - Addl, AddIU, SLTI, SLTIU, Andl, Ori, Xorl, 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>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>add $1,$2,100</td>
<td>$1 = $2 + 100 + constant</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. unsigned</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100 + constant</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, Lo = quotient, Hi = remainder</td>
<td>Hi = $2 mod $3</td>
</tr>
</tbody>
</table>
| divide unsigned   | divu $2,$3  | Lo = $2 ÷ $3, Hi = $2 mod $3       | Unsigned quotient & remainder |}

**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>$1$ &amp; $2$ &amp; $3$</td>
<td>$1 = 2 &amp; 3$</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>$1$ &amp; $2$</td>
<td>$1 = 2</td>
<td>3$</td>
</tr>
<tr>
<td>xor</td>
<td>$1$ &amp; $2$ &amp; $3$</td>
<td>$1 = 2 ^ 3$</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>$1$ &amp; $2$ &amp; $3$</td>
<td>$1 = \neg (2</td>
<td>3)$</td>
</tr>
<tr>
<td>and immediate</td>
<td>$1$ &amp; $2$, $10$</td>
<td>$1 = 2 &amp; 10$</td>
<td>Logical AND reg. constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>$1$ &amp; $2$, $10$</td>
<td>$1 = 2</td>
<td>10$</td>
</tr>
<tr>
<td>xor immediate</td>
<td>$1$ &amp; $2$, $10$</td>
<td>$1 = \neg 2 ^ 10$</td>
<td>Logical XOR reg. constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>$1$, $2$, $10$</td>
<td>$1 = 2 &lt;&lt; 10$</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>$1$, $2$, $10$</td>
<td>$1 = 2 &gt;&gt; 10$</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>$1$, $2$, $10$</td>
<td>$1 = 2 &lt;&lt; 10$</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>$1$, $2$, $10$</td>
<td>$1 = 2 &gt;&gt; 10$</td>
<td>Shift right 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</td>
<td>Store word</td>
</tr>
<tr>
<td>SH</td>
<td>Store half</td>
</tr>
<tr>
<td>SB</td>
<td>Store byte</td>
</tr>
<tr>
<td>LW</td>
<td>Load word</td>
</tr>
<tr>
<td>LH</td>
<td>Load halfword</td>
</tr>
<tr>
<td>LHU</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>LB</td>
<td>Load byte</td>
</tr>
<tr>
<td>LBU</td>
<td>Load byte unsigned</td>
</tr>
</tbody>
</table>

LUI R1, 40 | Load Upper Immediate (16 bits shifted left by 16) |

Why need LUI?

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:

- 00000000 00001010
- 00000000 10001100
- 11111111 10001100

- When is an immediate operand sign extended?
  - Arithmetic instructions (add, sub, etc.) always sign extend immediates even for the unsigned versions of the instructions!
  - Logical instructions do not sign extend immediates (They are zero extended)
  - Load/Store address computations always sign extend immediates

- Multiply/Divide have no immediate operands however:
  - “unsigned” => treat operands as unsigned

- The data loaded by the instructions lb and lh are extended as follows (“unsigned” => don’t extend):
  - lb, lh are zero extended
  - lb, lh are sign extended

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
### 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</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>sltiu $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0 Compare &lt; constant; natural numbers</td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>slli $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

R1 = 0...00 0000 0000 0000 0001_1
R2 = 0...00 0000 0000 0000 0010_1
R3 = 1...11 1111 1111 1111 1111_1

- 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 = ;

Branch & Pipelines

- After executing these instructions:
  - `li r3, #7`
  - `sub r4, r4, 1`
  - `bz r4, LL`
  - `addi r5, r3, 1`

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.

Delayed Branches

- `li r3, #7`
- `sub r4, r4, 1`
- `bz r4, LL`
- `addi r5, r3, 1`

- 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 (???)

Jump and link (jal inst):
- Put the return addr. Into link register (R31):
  - PC+4 (logical architecture)
  - PC+8 physical (“Raw”) architecture => delay slot executed
- Then jump to destination address

Filling Delayed Branches

- 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 & mode bits into status register
- **load word left/right**: Supports misaligned word loads
- **store word left/right**: Supports misaligned word stores

---

### 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)