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)

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 address: `adds 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></td>
<td>Store C,R3</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
  - \(2^{32} \times\) 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 Mode Usage? (ignore register mode)

3 programs measured on machine with all address modes (VAX)

--- Displacement: 42% avg, 32% to 55% 73%
--- Immediate: 33% avg, 17% to 43% 85%
--- 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

MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

Register (direct) op rs rt rd
Immediate op rs rt imm
Base+index op rs rt imm
PC-relative op rs rt imm

- Register Indirect?

Generic Examples of Instruction Format Widths

Variable: ...
Fixed: ... ...
Hybrid: ...

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;
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>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</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>add $1, $2, $3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>sub $1, $2, $3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsigned</td>
<td>add $1, $2, 100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mul $2, $3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>mult $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, Hi = quotient, Lo = remainder</td>
<td></td>
</tr>
<tr>
<td>divide unsigned</td>
<td>div $2, $3</td>
<td>Lo = $2 ÷ $3, Hi = remainder</td>
<td></td>
</tr>
<tr>
<td>divisor</td>
<td>divisor</td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>Move from Hi</td>
<td>move $1</td>
<td>$1 = $1</td>
<td></td>
</tr>
<tr>
<td>Move from Lo</td>
<td>move $1</td>
<td>$1 = $1</td>
<td></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>not</td>
<td>not $1,$2,$3</td>
<td>$1 = ~($2 &amp; $3)</td>
<td>3 reg. operands; Logical NOR</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>or $1,$2,10</td>
<td>$1 = $2</td>
<td>Logical OR reg, constant</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xor $1,$2,10</td>
<td>$1 = ~($2 &amp; $3)</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 arithmetic</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,$2,10</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithmetic</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right (sign extend)</td>
</tr>
</tbody>
</table>

### MIPS data transfer instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>500(R4), R3</td>
<td>Store word</td>
</tr>
<tr>
<td>SH</td>
<td>902(R2), R3</td>
<td>Store half</td>
</tr>
<tr>
<td>SB</td>
<td>40(R3), R2</td>
<td>Store byte</td>
</tr>
<tr>
<td>LW</td>
<td>R1, 30(R2)</td>
<td>Load word</td>
</tr>
<tr>
<td>LH</td>
<td>R1, 40(R3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>LHU</td>
<td>R1, 40(R3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>LB</td>
<td>R1, 40(R3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>LBU</td>
<td>R1, 40(R3)</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>LUI</td>
<td>R1, 40</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

**Why need LUI?**

**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` <
  - `BGZE` >=
  - `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!
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 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

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 (???)
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?

### 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: **Intel**
- Support these data sizes and types: 8bit, 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**
- Aim for a minimalist instruction set: **YES**