Instruction Set Architectures

- Last Time
  - Computer elements
    - Transistors, wires, pins
- Today
  - Finish computer elements
  - ISA overview
  - MIPS ISA
  - ISA extensions
  - "Alternate" ISAs

Instruction Set Architecture

- Contract between programmer and the hardware
  - Defines visible state of the system
  - Defines how state changes in response to instructions

- Programmer: ISA is model of how a program will execute
- Hardware Designer: ISA is formal definition of the correct way to execute a program

- ISA specification
  - The binary encodings of the instruction set
ISA Basics

Architecture vs. Implementation

- **Architecture**: defines what a computer system does in response to a program and a set of data
  - Programmer visible elements of computer system

- **Implementation**: defines how a computer does it
  - Sequence of steps to complete operations
  - Time to execute each operation
  - Hidden "bookkeeping" functions
Classifying Instruction Set Architectures

Based on how the instructions receive/produce operands

- **Stack**
  - Burroughs B5000 (1963), Java Virtual Machine (late 90s)
- **Accumulator**
  - Univac-I (1951), EDSAC (1949)
- **Register-memory**
- **Load/Store**
  - IBM 801 (1974-prototype), Stanford MIPS/Berkeley RISC (mid 1980s)
  - IBM Power, MIPS, DEC Alpha, DSP, VLIW, etc.
- **Dataflow**
  - Numerous research machines over the years
- **Special case: Vector ISAs**

ISA Basics

Instruction formats
Instruction types
Addressing modes

Machine state
Memory organization
Register organization

Data types
Operations
Interrupts/Events

Mem
Regs
Before State

Mem
Regs
After State

Op Mode Ra Rb
Memory Addressing

- Different size accesses (byte, half-word, word, etc.)
- Endian-ness
  - Byte ordering within a larger object
- Alignment
  - Is an access allowed to span any arbitrary boundaries?
- Addressing modes
  - How is an address computed?
  - Absolute, relative to the program counter, computed

Addressing Mode Summary

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#n</td>
<td>Immediate</td>
</tr>
<tr>
<td>(0x1000)</td>
<td>Absolute</td>
</tr>
<tr>
<td>Rn</td>
<td>Register</td>
</tr>
<tr>
<td>(Rn)</td>
<td>Register indirect</td>
</tr>
<tr>
<td>− (Rn)</td>
<td>Predecrement</td>
</tr>
<tr>
<td>(Rn) +</td>
<td>Postincrement</td>
</tr>
<tr>
<td>* (Rn)</td>
<td>Memory indirect</td>
</tr>
<tr>
<td>* (Rn) +</td>
<td>Postincrement</td>
</tr>
<tr>
<td>d (Rn)</td>
<td>Displacement (b,w,l)</td>
</tr>
<tr>
<td>d (Rn) [Rx]</td>
<td>Scaled</td>
</tr>
</tbody>
</table>

VAX 11 had 27 addressing modes (why?)
Native Data Types

- **General purpose processors**
  - Various size integers (now represented using 2’s complement)
    - Previous included binary coded decimal, signed-magnitude, etc.
  - Floating-point (single/double precision at 32/64 bits)
  - “Addresses” - usually just integers, but not always
  - Conditions (results of comparison operations)

- **But - these are not written in stone**
  - Vectors (short as well as long)
  - Fixed point (often used for signal processing)
  - XYZW vertices (128 bits) for graphics
Operations (Instructions)

- Arithmetic/logical
- Data transfer (load/store, move)
- Control (branch, jump, call)
  - Branch delay slots
- System (operating system call, access to special state)
- Miscellaneous
  - Subword parallel, saturating arithmetic
  - Floating-point
  - Decimal (such as for BCD)
  - String operations (some found in VAX, etc.)
  - Graphics - pixel/vertex operations
- Basically - whatever you can justify and build in HW
  - But how do you decide what to put in?
    - Shouting contest
    - Democracy
    - Seniority
    - Analysis of tradeoffs

Instruction Encoding

- Variable length - VAX, x86
  - Instruction length depends on the number of operands, etc.
  - Intel 432 took this to an extreme
- Fixed length - MIPS, other "RISC" ISAs
  - All instructions are the same length
- Hybrid - ARM Thumb, MIPS 16
  - May support small number of fixed sizes (16/32 bits)
Control - Exceptions/Events

- Implied multi-way branch after every instruction
  - External events (interrupts)
    - completion of I/O operations
  - Internal events (faults or exceptions)
    - arithmetic overflow
    - page fault

- What happens????
  - EPC ← PC of instruction that caused fault
  - PC ← f(Fault type)
    - new PC from HW table lookup
  - Return: PC ← EPC + 4

- How would you use this to aid compatibility?

MIPS ISA

- 32 GP Integer registers (R0-31) – 32 bits each
  - R0=0, other registers governed by conventions (SP, FP, RA, etc.)
- 32 FP registers (F0-F31)
  - 16 double-precision (use adjacent 32-bit registers)
- 8, 16, and 32 bit integer data types
- Load/Store architecture (no memory operations in ALU ops)
- Simple addressing modes
  - Immediate \( R1 ← 0x23 \)
  - Displacement \( R2 ← d(Rx) \ldots O(R3), 0x1000(R0) \)
- Simple fixed instruction format (3 types), 90 instructions
- Fused compare and branch
- "ISA" has pseudo instruction that are synthesized into simple sequences (ie. rotate left rol = combination of shift and mask)
- Designed for fast hardware (pipelining) + optimizing compilers
MIPS ISA (a visual)

Fixed-Format

R: rd ← rs1 op rs2
6 5 5 5 11
Op RS1 RS2 RD func

I: ld/st, rd ← rs1 op imm, branch
6 5 5 16
Op RS1 RD Const

J: j, jal
6 26
Op Const

MIPS: Software conventions for Registers

| 0 | zero | constant 0 |
| 1 | at | reserved for assembler |
| 2 | v0 | expression evaluation & |
| 3 | v1 | function results |
| 4 | a0 | arguments |
| 5 | a1 | |
| 6 | a2 | |
| 7 | a3 | |
| 8 | t0 | temporary: caller saves |
| 9 | | (callee can clobber) |
| 10 | | |
| 11 | | |
| 12 | | |
| 13 | | |
| 14 | | |
| 15 | t7 | |
| 16 | s0 | callee saves |
| 17 | | (caller can clobber) |
| 18 | | |
| 19 | | |
| 20 | | |
| 21 | | |
| 22 | | |
| 23 | s7 | |
| 24 | t8 | temporary (cont’d) |
| 25 | t9 | |
| 26 | k0 | reserved for OS kernel |
| 27 | k1 | |
| 28 | gp | Pointer to global area |
| 29 | sp | Stack pointer |
| 30 | fp | frame pointer |
| 31 | ra | Return Address (HW) |

Plus a 3-deep stack of mode bits.
## 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; <strong>exception possible</strong></td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; <strong>exception possible</strong></td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; <strong>exception possible</strong></td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; <strong>no exceptions</strong></td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; <strong>no exceptions</strong></td>
</tr>
<tr>
<td>add imm. unsig.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100</td>
<td>+ constant; <strong>no exceptions</strong></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, Hi = $2 mod $3</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3</td>
<td>Unsigned quotient &amp; remainder</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>

### 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`
### 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 Å $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~(S2</td>
<td>S3)</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>slv $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>sr1v $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>
<tr>
<td>LUI R1, 40</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

### Why need LUI?

LUI R5

0000 ... 0000
MIPS Compare and Branch

- **Compare and Branch**
  - **BEQ rs, rt, offset** if \( R[rs] = R[rt] \) then PC-relative branch
  - **BNE rs, rt, offset** \( \neq \)

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

- 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Equal test; PC relative branch</em></td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ( $1 \neq $2 ) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Not equal test; PC relative</em></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 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare less than; 2’s comp.</em></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 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare &lt; constant; 2’s comp.</em></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 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Compare &lt; constant; natural numbers</em></td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Jump to target address</em></td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>For switch, procedure return</em></td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>For procedure call</em></td>
</tr>
</tbody>
</table>
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

Multimedia Instruction Extensions

- **Properties of multimedia applications**
  - Narrower data types
    - Bytes, halfwords, fixed-point, single-precision
  - Statically known loop bounds
  - Different common idioms
    - Multiply+accumulate, averaging, permutations
- ....results in different instructions
  - Short SIMD (single instruction/multiple data)
  - Segmented arithmetic
  - Loop counters (more often in multimedia processors)
  - New instructions implemented directly in hardware
Philips Trimedia ISA (TM1300)

- **VLIW instruction set**
  - One instruction holds 5 independent operations
  - 27 different types of functional units
  - NOP placeholders required if operation slot is unused
  - 3 branch delay slots exposed to compiler
- **Implications**
  - Hazards must be detected/prevented in SW
- **Guarded/Predicated execution**
  - Execution/nullification of instruction depends on value of a general purpose register
- **State**
  - 128 32-bit registers
  - Instructions kept compressed until delivered to processor
- **Operations**
  - Also uses sub-word parallelism (2 16-bit operations)
  - Saturating arithmetic

---

Trimedia 1300 Block Diagram

![Trimedia 1300 Block Diagram](image)
TM1300 Issue Restrictions

<table>
<thead>
<tr>
<th>Issue slot 1</th>
<th>Issue slot 2</th>
<th>Issue slot 3</th>
<th>Issue slot 4</th>
<th>Issue slot 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST</td>
<td>CONST</td>
<td>CONST</td>
<td>CONST</td>
<td>CONST</td>
</tr>
<tr>
<td>ALU</td>
<td>ALU</td>
<td>ALU</td>
<td>ALU</td>
<td>ALU</td>
</tr>
<tr>
<td>SHIFTER</td>
<td>SHIFTER</td>
<td>FCOMP</td>
<td>DMEM</td>
<td>DMEM</td>
</tr>
<tr>
<td>FALU</td>
<td>DSPMUL</td>
<td>DSPMUL</td>
<td>FALU</td>
<td>DMEMSPEC</td>
</tr>
<tr>
<td>BRANCH</td>
<td>BRANCH</td>
<td>BRANCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFMUL</td>
<td>IFMUL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSPALU</td>
<td>FTough</td>
<td>DSPALU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GPU Instruction Sets
Example: DPH – homogeneous dot product

- The DPH instruction assigns the four-component dot product of the two source vectors where the W component of the first source vector is assumed to be 1.0 into the destination register.

- Semantics:
  \[ t.x = \text{source0.c}^{**}; \]
  \[ t.y = \text{source0.}^{*}c^{*}; \]
  \[ t.z = \text{source0.}^{*}c^{*}; \]
  \[ \text{if (negate0) } \{ t.x = -t.x; t.y = -t.y; t.z = -t.z; \} \]
  \[ u.x = \text{source1.}^{**}c; \]
  \[ u.y = \text{source1.}^{*}c^{*}; \]
  \[ u.z = \text{source1.}^{*}c^{*}; \]
  \[ u.w = \text{source1.}^{**}c; \]
  \[ \text{if (negate1) } \{ u.x = -u.x; u.y = -u.y; u.z = -u.z; u.w = -u.w; \} \]
  \[ v.x = t.x * u.x + t.y * u.y + t.z * u.z + u.w; \]
  \[ \text{if (xmask) } \text{destination.x} = v.x; \]
  \[ \text{if (ymask) } \text{destination.y} = v.x; \]
  \[ \text{if (zmask) } \text{destination.z} = v.x; \]
  \[ \text{if (wmask) } \text{destination.w} = v.x; \]
Principles of Instruction Set Design

- Keep it simple (KISS)
  - complexity
    - increases logic area
    - increases pipe stages
    - increases development time
  - evolution tends to make kludges
- Orthogonality (modularity)
  - simple rules, few exceptions
  - all ops on all registers

- Frequency
  - make the common case fast
    - some instructions (cases) are more important than others

Principles of Instruction Set Design (part 2)

- Generality
  - not all problems need the same features or instructions
  - principle of least surprise
  - performance should be easy to predict

- Locality and concurrency
  - design ISA to permit efficient implementation
  - today
  - 10 years from now
Review of ISA Principles

- **Good ISA design**
  - KISS! - only implement necessities (encodings, address modes, etc.)
  - FOG: Frequency, Orthogonality, Generality
- **Instruction Types**
  - ALU ops, Data movement, Control
- **Addressing modes**
  - Matched to program usage (local vars, globals, arrays)
- **Program Control**
  - Conditional/unconditional branches and jumps
  - Where to store conditions
  - PC relative and absolute

Next Time

- **Microarchitecture**
  - Components + structure
- **Some comments on pipelining**
- **Reading assignment**
  - Review due:
    - Colwell, Instruction sets and beyond
    - Burger et al., Scaling to end of silicon...