Announcements

Homework #1 handed out today
• Due on Monday
No lecture on Sept. 12th
**Endian Wars**

Order of bytes in words
- Big endian — MSB at address 0x......0
- Little endian — MSB at address 0x......3

Big endian — IBM, Motorola, SPARC
Little endian — DEC, Intel
- Windows NT requires this (pathetic!)

Mode selectable — PowerPC, MIPS
- becoming more common

---

**Operand Alignment**

What is alignment?
- address mode size = 0
- natural boundaries

e.g., aligned word (4 bytes) load from 0x....0
e.g., unaligned word (4 bytes) load from 0x....1

Placing no restrictions
- simpler software
- hardware must detect misalignment, make 2 memory accesses
- expensive logic, slows down all references (why?)
- sometimes required for backward compatibility!
Operand Alignment (Cont.)

Restricted alignment
- software must guarantee alignment
- hardware only detects misalignment
- trap handler aligns

Middle ground
- misaligned data ok but requires multiple instructions
- compiler must still know
- trap on misaligned access

Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>R_i</td>
</tr>
<tr>
<td>Immediate</td>
<td>#n</td>
</tr>
<tr>
<td>Displacement</td>
<td>M[R_i + #n]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>M[R_i]</td>
</tr>
<tr>
<td>Indexed</td>
<td>M[R_i + R_j]</td>
</tr>
<tr>
<td>Absolute</td>
<td>M[#n]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>M[M[R_i]]</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>M[R_i]; R_i += d</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>M[R_i]; R_i -= d</td>
</tr>
<tr>
<td>Scaled</td>
<td>M[R_i + #n + R_j * d]</td>
</tr>
<tr>
<td>Update</td>
<td>M[R_i = #n + R_j]</td>
</tr>
</tbody>
</table>

Modes 1-4 account for 93% of all VAX operands [Clark & Emer]
Operations

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>and, add</td>
</tr>
<tr>
<td>Data transfer</td>
<td>move, load</td>
</tr>
<tr>
<td>Control</td>
<td>branch, jump, call, return</td>
</tr>
<tr>
<td>System</td>
<td>trap, rett</td>
</tr>
<tr>
<td>Floating point</td>
<td>add, mul, div, sqrt</td>
</tr>
<tr>
<td>Decimal</td>
<td>addd, convert</td>
</tr>
<tr>
<td>String</td>
<td>move, compare</td>
</tr>
</tbody>
</table>

Endian Wars

Order of bytes in words
- Big endian — MSB at address 0x......0
- Little endian — MSB at address 0x.....3

Big endian — IBM, Motorola, SPARC

Little endian — DEC, Intel
- Windows NT requires this (pathetic!)

Mode selectable — PowerPC, MIPS
- becoming more common
Operand Alignment

What is alignment?
- address mode size = 0
- natural boundaries

e.g., aligned word (4 bytes) load from 0x....0
e.g., unaligned word (4 bytes) load from 0x....1

Placing no restrictions
- simpler software
- hardware must detect misalignment, make 2 memory accesses
- expensive logic, slows down all references (why?)
- sometimes required for backward compatibility!

Operand Alignment (Cont.)

Restricted alignment
- software must guarantee alignment
- hardware only detects misalignment
- trap handler aligns

Middle ground
- misaligned data ok but requires multiple instructions
- compiler must still know
- trap on misaligned access
**Addressing Modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>$R_i$</td>
</tr>
<tr>
<td>Immediate</td>
<td>$#n$</td>
</tr>
<tr>
<td>Displacement</td>
<td>$M[R_i + #n]$</td>
</tr>
<tr>
<td>Register indirect</td>
<td>$M[R_i]$</td>
</tr>
<tr>
<td>Indexed</td>
<td>$M[R_i + R_j]$</td>
</tr>
<tr>
<td>Absolute</td>
<td>$M[#n]$</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>$M[M[R_i]]$</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>$M[R_i]; R_i := d$</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>$M[R_i]; R_i := -d$</td>
</tr>
<tr>
<td>Scaled</td>
<td>$M[R_i + #n + R_j * d]$</td>
</tr>
<tr>
<td>Update</td>
<td>$M[R_i = #n + R_j]$</td>
</tr>
</tbody>
</table>

Modes 1-4 account for 93% of all VAX operands [Clark & Emer]

**Operations**

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>and, add</td>
</tr>
<tr>
<td>Data transfer</td>
<td>move, load</td>
</tr>
<tr>
<td>Control</td>
<td>branch, jump, call, return</td>
</tr>
<tr>
<td>System</td>
<td>trap, rett</td>
</tr>
<tr>
<td>Floating point</td>
<td>add, mul, div, sqrt</td>
</tr>
<tr>
<td>Decimal</td>
<td>addd, convert</td>
</tr>
<tr>
<td>String</td>
<td>move, compare</td>
</tr>
</tbody>
</table>
Control Instructions

Aspects:
1. Taken or not taken?
2. Where is the target?
3. Links return address?
4. Saves/restores state?

Instructions that change the PC

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Conditional) branches</td>
<td>1,2</td>
</tr>
<tr>
<td>(Unconditional) jumps</td>
<td>2</td>
</tr>
<tr>
<td>Function calls</td>
<td>2,3,4</td>
</tr>
<tr>
<td>Function returns</td>
<td>2,4</td>
</tr>
<tr>
<td>System calls</td>
<td>2,3,4</td>
</tr>
<tr>
<td>System returns</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Taken or Not Taken

Compare and branch
✓ no extra compare
✓ no state passed between instructions
✗ requires ALU ops
✗ restricts code scheduling opportunities

Implicitly set condition codes
✓ can be set "for free"
✗ restricts code scheduling
✗ extra state to save/restore
Taken or Not Taken (Cont.)

Explicitly set condition codes, conditional registers
✓ can be set “for free”
✓ decouples branch/fetch from pipeline
✗ extra state to save/restore

Condition in general-purpose registers
✓ no special state
✗ uses up a register
✗ branch condition separate from branch logic in pipeline

Some data from MIPS
☐ > 80% branches use immediate data
☐ > 80% of these are zero immediates
☐ 50% branches compare equal, less than, or greater than zero

Compromise in MIPS
☐ branch instructions with equal, less than or greater than zero
☐ compare instructions for all other compares
Where is the Target?

Arbitrary specifier
✓ orthogonal
✗ more bits to specify => more time to decode
✗ branch execution and target separated in pipeline

PC-relative with immediate
✓ position independent, target computable in branch unit
✓ short immediate sufficient — #bits < 4 (47%), < 8 (94%)
✗ target must be known statically => can’t go far
✗ other techniques needed for returns, distant jumps

Where is the Target? (Cont.)

Register
✓ short specifier
✓ can jump anywhere
✓ can have dynamic target
✗ branch and target are separated in the pipeline

Vectored traps (for system calls)
✓ protection
✗ implementation headaches
### Where is the Target? (Cont.)

Common compromises

<table>
<thead>
<tr>
<th>Type</th>
<th>Where is the target?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Conditional) branches</td>
<td>PC-relative</td>
</tr>
<tr>
<td>(Unconditional) jumps</td>
<td>PC-relative and register</td>
</tr>
<tr>
<td>Function calls</td>
<td>PC-relative and register</td>
</tr>
<tr>
<td>Function returns</td>
<td>register</td>
</tr>
<tr>
<td>System calls</td>
<td>trap</td>
</tr>
<tr>
<td>System returns</td>
<td>register</td>
</tr>
</tbody>
</table>

### Saves/Restores State?

What state?
- function calls => registers
- system calls => registers, flags, PC, PSW, etc.

Hardware need not save registers
- caller can save registers in use
- callee can save registers it will use

Hardware register save
- IBM STM, VAX CALLS
- faster?

Many recent ISA's
- do no register saving
- or do implicit saving with register windows
Notation

Generic assembly code
- SUB R1, R2, R3
- Means R1 gets R2 - R3

Data sizes
- Byte = 8 bits
- Halfword = 16 bits
- Word = 32 bits
- Doubleword = 64 bits
- Quadword = 128 bits

VAX

DEC 1977 VAX 11/780
- upward compatible from PDP-11
- 32-bit words and addresses
- virtual memory
- 16 GPRs (R15 PC, R14 SP), CCs
- extremely orthogonal and memory-memory
- decode as byte stream, variable in length
  - opcode, operation, #operands, operand types
VAX Data Types

- 8, 16, 32, 64, 128
- char string, 8 bits/char
- decimal, 4 bits/char
- numeric string, 8 bits/digit

VAX Addressing Modes

- literal, 6 bits
- 8, 16, 32 bit immediates
- register, register deferred
- 8, 16, 32 bit displacements
- 8, 16, 32 bit displacements deferred
- indexed (scaled)
- autoincrement, autodecrement
- autoincrement deferred
VAX Operations

- data transfer including string move
- arithmetic and logical (2 and 3 operands)
- control (branch, jump, etc.)
  - e.g., AOBLEQ
- function calls save state
- bit manipulation
- floating point: add, sub, mul, div, polyf
- system: exception, VM
- other: CRC, INSQUE

VAX Example

addl3 R1, 737(R2), #456
- byte 1: addl3
- byte 2: mode, R1
- byte 3: mode, R2
- byte 4-5: 737
- byte 6: mode
- byte 7-10:456

VAX has too many modes and formats
However, few modes/formats => fast decoding in the pipeline
Argument for RISC?
8086

Intel in 1978
- chosen for IBM PC 1980
- remains most popular 16-bit architecture
- upward compatible with 8080
- complex: “difficult to explain and impossible to love”
- special purpose registers
  - 4 arithmetic, 4 address, 4 segment, 2 control
- addresses: 16-bit segments $\ll$ $+$ 16-bit offset
  - 64K 16 KB-aligned 64KB segments
- many formats, see H&P figure 4.13

DLX

H&P 1990
- RISC strongly related to MIPS
- 32-bit byte addresses aligned
- load/store: only displacement addressing
- standard data types
- 3 fixed length formats
  - 32 32-bit GPRs (R0 = 0)
  - 16 64-bit or 32 32-bit FPRs
  - FP status register
  - no CCs
DLX (Cont.)

Data transfer
- load/store word
- load/store byte/halfword signed?
- load/store FP single/double
- moves between GPRs and FPRs

ALU
- add/sub signed? Immediate?
- mul/div signed?
- and, or, xor, immediate?
- shifts: ll, rl, ra immediate?
- sets immediate?

DLX (Cont.)

Control
- branches == 0, <> 0
- conditional branch testing FP bit
- jump, jump register
- jump&link, jump&link register
- trap, return from trap

FP
- add/sub/mul/div single/double
- fp converts, fp set
DLX Instruction Formats

<table>
<thead>
<tr>
<th>I-type</th>
<th>R-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode rs1 rd</td>
<td>opcode rs1 rs2 rd</td>
<td>opcode offset added to PC</td>
</tr>
</tbody>
</table>

I: ALU immediate, load/store, branches, jump register
R: RRR ALU ops
J: unconditional jumps

Compiler 101

Compiler goals:
- all correct programs execute correctly
- most compiled programs execute fast
- compile fast
- provide support for debugging
Multiple phases to manage complexity
- Lexical analysis
- Parsing => Intermediate representation
- Optimization & code generation
  - Procedure inlining
  - Loop optimizations
  - Common sub-expression elimination
  - Jump optimization
  - Constant propagation
  - Register allocation
  - Strength reduction
  - Pipeline scheduling
- Generation assembly code

What compiler writers want?
- regularity
- orthogonality
- composability

Compilers perform a giant case analysis
- too many choices make it hard

Orthogonal instruction sets
- operation, addressing mode, data type
William Wulf’s Paper

One solution or all possible solutions
- 2 branch conditions, eq and lt
- or all six conditions, eq, ne, lt, gt, le, ge
- not 3 or 4

Primitives not solutions
“...by giving too much semantic content to the instruction, the machine designer made it possible to use the instruction only in limited contexts. In many cases the complex instructions are synthesized from more primitive operations, which if the compiler had access to, could be recomposed to more closely model the feature actually needed.”

Copyright © 2001 Falsafi, from Hill, Smith, Sohi, Vijaykumar, and Wood © Chapter 2: Instruction Set Architecture