CSE 502: Computer Architecture

Basic Instruction Decode
RISC ISA Format

• Fixed-length
  – MIPS all insts are 32-bits/4 bytes

• Few formats
  – MIPS has 3: R (reg, reg, reg), I (reg, reg, imm), J (addr)
  – Alpha has 5: Operate, Op w/ Imm, Mem, Branch, FP

• Regularity across formats (when possible/practical)
  – MIPS & Alpha opcode in same bit-position for all formats
  – MIPS rs & rt fields in same bit-position for R and I formats
  – Alpha ra/fa field in same bit-position for all 5 formats
# RISC Decode (MIPS)

<table>
<thead>
<tr>
<th>opcode[2,0]</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000xxxx</td>
<td>func</td>
<td>rt</td>
<td>j</td>
<td>jal</td>
<td>beq</td>
<td>bne</td>
</tr>
<tr>
<td>001xxxx</td>
<td>addi</td>
<td>addiu</td>
<td>slti</td>
<td>sltiu</td>
<td>andi</td>
<td>ori</td>
</tr>
<tr>
<td>010xxxx</td>
<td></td>
<td></td>
<td>rs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011xxxx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100xxxx</td>
<td>lb</td>
<td>lh</td>
<td>lw</td>
<td>lw</td>
<td>lbu</td>
<td>lhu</td>
</tr>
<tr>
<td>101xxxx</td>
<td>sb</td>
<td>sh</td>
<td>swl</td>
<td>sw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110xxxx</td>
<td>lwc0</td>
<td>lwc1</td>
<td>lwc2</td>
<td>lwc3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111xxxx</td>
<td>swc0</td>
<td>swc1</td>
<td>swc2</td>
<td>swc3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **opcode[5,3]**: Memory
  - 1x0: LD
  - 1x1: ST

**000xxx** = Br/Jump (except for 000000)

**001xxx** = Immediate
PLA Decoders (1/2)

• PLA = Programmable Logic Array
• Simple logic to transform opcode to control signals
  – is_jump = !op5 & !op4 & !op3 & (op2 | op1 | op0)
  – use_funct = !op5 & !op4 & !op3 & !op2 & !op1 & !op0
  – use_imm = op5 | !op5 & !op4 & op3
  – is_load = op5 & !op3
  – is_store = op5 & op3
PLA Decoders (2/2)

op_5
op_4
op_3
op_2
op_1
op_0

4-input AND gate

2-input OR gate

Array

AND

Array

is_store
is_load
is_mem
use_imm
use_funct
is_jump
CISC ISA

• RISC focus on fast access to information
  – Easy decode, many registers, fast caches

• CISC focus on max expressiveness per min space
  – Designed in era with fewer transistors
  – Each memory access very expensive
    • Pack as much work into as few bytes as possible
    • More “expressive” instructions
      – Better potential code generation in theory
      – More complex code generation in practice
## ADD in RISC ISA

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>ADD R4, R3, R2</td>
<td>R4 &lt;= R3 + R2</td>
</tr>
</tbody>
</table>
### ADD in CISC ISA

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>ADD R4, R3</td>
<td>R4 &lt;= R4 + R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>ADD R4, #3</td>
<td>R4 &lt;= R4 + 3</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>ADD R4, (R1)</td>
<td>R4 &lt;= R4 + Mem[R1]</td>
</tr>
<tr>
<td>Displacement</td>
<td>ADD R4, 100(R1)</td>
<td>R4 &lt;= R4 + Mem[100+R1]</td>
</tr>
<tr>
<td>Indexed/Base</td>
<td>ADD R3, (R1+R2)</td>
<td>R3 &lt;= R3 + Mem[R1+R2]</td>
</tr>
<tr>
<td>Direct/Absolute</td>
<td>ADD R1, (1234)</td>
<td>R1 &lt;= R1 + Mem[1234]</td>
</tr>
<tr>
<td>Memory Indirect</td>
<td>ADD R1, @(R3)</td>
<td>R1 &lt;= R1 + Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Auto-Increment</td>
<td>ADD R1,(R2)+</td>
<td>R1 &lt;= R1 + Mem[R2]; R2++</td>
</tr>
<tr>
<td>Auto-Decrement</td>
<td>ADD R1,-(R2)</td>
<td>R2--; R1 &lt;= R1 + Mem[R2]</td>
</tr>
</tbody>
</table>
x86

- CISC, stemming from the original 4004 (~1971)
- Example: “Move” instructions
  - General Purpose data movement
    - \( R \rightarrow R, M \rightarrow R, R \rightarrow M, I \rightarrow R, I \rightarrow M \)
  - Exchanges
    - \( EAX \leftrightarrow ECX \), byte order within a register
  - Stack Manipulation
    - push / pop \( R \leftrightarrow Stack \), pusha/popa (removed in 64-bit, thankfully)
  - Type Conversion
  - Conditional Moves

Many ways to do the same/similar operation
x86 Encoding

• x86 Instruction Format:

<table>
<thead>
<tr>
<th>Prefixes</th>
<th>Opcode</th>
<th>Mod R/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4 bytes</td>
<td>1-2 bytes</td>
<td>0-1 bytes</td>
<td>0-1 bytes</td>
<td>0/1/2/4 bytes</td>
<td>0/1/2/4 bytes</td>
</tr>
</tbody>
</table>

- Shortest Inst: 1 byte
- Longest Inst: 15 bytes

• Opcode indicates if Mod R/M is present
  - Many (not all) instructions use the Mod R/M byte
  - Mod R/M specifies if optional SIB byte is used
  - Mod R/M and SIB may specify additional constants
    - Displacement, Immediate

Instruction length not known until after decode
x86 Mod R/M Byte

- Mode = 00: No-displacement, use Mem[Rmmm]
- Mode = 01: 8-bit displacement, Mem[Rmmm+disp)]
- Mode = 10: 32-bit displacement (similar to previous)
- Mode = 11: Register-to-Register, use Rmmm
x86 Mod R/M Exceptions

• Mod=00, R/M = 5 $\rightarrow$ get operand from 32-bit imm
  - Add $\begin{array}{llll}
  0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 2 & 3 & 4
  \end{array}$ EDX = EDX+Mem[0cff1234]

• Mod=00, 01, or 10, R/M = 4 $\rightarrow$ use the “SIB” byte
  - SIB = Scale/Index/Base

- $\begin{array}{llll}
  \text{Mod R/M} & \text{SIB} & \text{bbb} & \text{sss} & \text{iii}
  \end{array}$
  - $\text{bbb} \neq 5$: use reg$_{bbb}$
  - $\text{bbb} = 5$: use 32-bit imm
  - (Mod = 00 only)

- $\text{iii} \neq 4$: use si
  - $\text{iii} = 4$: use 0

- $\text{si} = \text{reg}_{\text{iii}} \ll \text{ss}$
x86 Opcode Confusion

• There are different opcodes for \( A \leftarrow B \) and \( B \rightarrow A \)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001001</td>
<td>MOV EAX, EBX</td>
</tr>
<tr>
<td>0001011</td>
<td>MOV EBX, EAX</td>
</tr>
<tr>
<td>0001001</td>
<td>MOV EAX, EBX</td>
</tr>
</tbody>
</table>

• If Opcode = 0F, then use next byte as opcode
• If Opcode = D8-DF, then FP instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101100</td>
<td>FP opcode</td>
</tr>
</tbody>
</table>
x86 Decode Example

```c
struct { long a; long b; long c; } myobj[5];
myobj[2]->b = 0xF001234;
```

```c
MOV 0xF001234,0x8(%ebx,%eax,8)
```

MOV reg\<imm\> (store 32-bit Imm in reg ptr, use Mod R/M)

- Mod=2(10) (use 32-bit Disp)
- R/M = 4(100) (use SIB)
- reg ignored
- ss=3(11) \rightarrow Scale by 8
  - use EAX, EBX

Total: 11 byte instruction

Note: Add 4 prefixes, and you reach the max size.
RISC (MIPS) vs CISC (x86)

lui R1, Disp[31:16]
ori R1, R1, Disp[15:0]
add R1, R1, R2
shli R3, R3, 3
add R3, R3, R1
lui R1, Imm[31:16]
ori R1, R1, Imm[15:0]
st [R3], R1

MOV Imm, Disp(%EBX,%EAX,8)

8 insns. at 32 bits each vs 1 insn. at 88 bits: 2.9x!
x86-64 / EM64T

- 8 → 16 general purpose registers
  - But we only used to have 3-bit register fields...
- Registers extended from 32 → 64 bits each
- Default: instructions still 32-bit
  - New “REX” prefix byte to specify additional information

```
0100 m R I B
```

- \( m=0 \) 32-bit mode
- \( m=1 \) 64-bit mode
IA32+64-bit exts

(Taken from Bob Colwell’s Eckert-Mauchly Award Talk, ISCA 2005)

Ugly? Scary? But it works…
Decoded x86 Format

• RISC: easy to expand → union of needed info
  – Generalized opcode (not too hard)
  – reg1, reg2, reg3, immediate (possibly extended)
  – Some fields ignored

• CISC: union of all possible info is huge
  – Generalized opcode (too many options)
  – Up to 3 regs, 2 immediates

Too expensive to decode x86 into control bits
x86 → RISC-like mops

- x86 decoded into “uops” (Intel) or “ROPs” (AMD) … (micro-ops or RISC-ops)
  - Each uop is RISC-like
  - uops have limitations to keep union of info practical

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>RISC-like MOPS</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD EAX, EBX</td>
<td>ADD EAX, EBX</td>
<td>1 uop</td>
</tr>
<tr>
<td>ADD EAX, [EBX]</td>
<td>LOAD tmp = [EBX]  ADD EAX, tmp</td>
<td>2 uops</td>
</tr>
<tr>
<td>ADD [EAX], EBX</td>
<td>LOAD tmp = [EAX]  ADD tmp, EBX  STA EAX  STD tmp</td>
<td>4 uops</td>
</tr>
</tbody>
</table>