CSE 502: Computer Architecture

Instruction Decode
RISC ISA Format

• This should be review...
  – Fixed-length
    • MIPS all insts are 32-bits/4 bytes
  – Few formats
    • MIPS has 3 formats: 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[5,3]</th>
<th>func</th>
<th>rt</th>
<th>j</th>
<th>jal</th>
<th>beq</th>
<th>bne</th>
<th>blez</th>
<th>bgtz</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td>001xxx = Br/Jump (except for 000000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
<td>addi</td>
<td>addiu</td>
<td>slti</td>
<td>sltiu</td>
<td>andi</td>
<td>ori</td>
<td>xori</td>
</tr>
<tr>
<td>010</td>
<td></td>
<td>001xxx = Immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td>rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>lb</td>
<td>lh</td>
<td>lw1</td>
<td>lw</td>
<td>lbu</td>
<td>lhu</td>
<td>lwr</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>sb</td>
<td>sh</td>
<td>swl</td>
<td>sw</td>
<td></td>
<td></td>
<td>swr</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>lwc0</td>
<td>lwc1</td>
<td>lwc2</td>
<td>lwc3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>swc0</td>
<td>swc1</td>
<td>swc2</td>
<td>swc3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PLA Decoders (1/2)

- PLA = Programmable Logic Array
- Simple logic to transform opcode to control signals
  - $is\_jump = \neg op5 \& \neg op4 \& \neg op3 \& (op2 \lor op1 \lor op0)$
  - $use\_funct = \neg op5 \& \neg op4 \& \neg op3 \& \neg op2 \& \neg op1 \& \neg op0$
  - $use\_imm = op5 \lor \neg op5 \& \neg op4 \& op3$
  - $is\_load = op5 \& \neg op3$
  - $is\_store = op5 \& op3$
PLA Decoders (2/2)

op_5
op_4
op_3
op_2
op_1
op_0

4-input AND gate

2-input OR gate

is_store
is_load
is_mem
use_imm
use_funct
is_jump

AND Array

OR Array
Superscalar Decode for RISC ISAs

- Decode X insns. per cycle (e.g., 4-wide)
  - Just duplicate the hardware
  - Instructions aligned at 32-bit boundaries
CISC ISA

• RISC focus on fast access to information
  – Easy decode, I$, large RF’s, D$

• CISC focus on max expressiveness per min space
  – Designed in era with fewer transistors, chips
  – 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 = 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 = 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/Absolute</td>
<td>ADD R1, (1234)</td>
<td>R1 = R1 + Mem[1234]</td>
</tr>
<tr>
<td>Memory Indirect</td>
<td>ADD R1, @(R3)</td>
<td>R1 = R1 + Mem[Mem[R3]]</td>
</tr>
<tr>
<td>Auto-Increment</td>
<td>ADD R1,(R2)+</td>
<td>R1 = R1 + Mem[R2]; R2++</td>
</tr>
<tr>
<td>Auto-Decrement</td>
<td>ADD R1, -(R2)</td>
<td>R2--; R1 = 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
  - Type Conversion
  - Conditional Moves

Many ways to do the same/similar operation
x86 Encoding

• Basic x86 Instruction:

  - Prefixes: 0-4 bytes
  - Opcode: 1-2 bytes
  - Mod R/M: 0-1 bytes
  - SIB: 0-1 bytes
  - Displacement: 0/1/2/4 bytes
  - Immediate: 0/1/2/4 bytes

  Shortest Inst: 1 byte

  Longest Inst: 15 bytes

•Opcode has flag indicating Mod R/M is present
  - Most 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

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 $\begin{array}{c}00\ 010\ 101\ 0cff1234\end{array}$  \[ EDX = EDX + \text{Mem}[0cff1234] \]

• Mod=00, 01 or 10, R/M = 4 $\rightarrow$ use the “SIB” byte
  - SIB = Scale/Index/Base
    - $\text{SIB} = \text{Scale/Index/Base}$
    - $\text{Mod} R/M$:
      - $00\ 010\ 100$
      - $\text{SIB}: \begin{array}{c} ss\ iii\ bbb \end{array}$
    - $bbb \neq 5$: use $\text{reg}_{bbb}$
    - $bbb = 5$: use 32-bit imm (Mod = 00 only)
    - $iii \neq 4$: use $\text{si}$
    - $iii = 4$: use 0
    - $\text{si} = \text{reg}_{iii} \ll \text{ss}$
x86 Opcode Confusion

- There are different opcodes for A ← B and B → A

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

```
10001011  11000011  MOV EAX, EBX
10001001  11000011  MOV EBX, EAX
10001001  11011000  MOV EAX, EBX
11011000  11R/M   FP opcode
```
**x86 Decode Example**

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) → Scale by 8
  - use EAX, EBX

**Opcode:** 11000111

**Mod R/M:** 10000100 11000011

**SIB:**

**Disp:**

**Imm:**

\[ *( (EAX \ll 3) + EBX + Disp ) = Imm \]

**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 [EBX+EAX*8+Disp], Imm

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 64-bit mode
- m=1 32-bit mode

Register specifiers are now 4 bits each: can choose 1 of 16 registers
IA32+64-bit exts

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

CPU architect

IA32

Ugly? Scary? But it works...
x86 Decode Hardware

Instruction bytes

- Prefix Decoder
- Left Shift
- Num Prefixes
- opcode decoder
- Mod R/M decoder
- SIB decoder
- Left Shift
- Left Shift
- +
- +
Decoded x86 Format

• RISC: easy to expand $\rightarrow$ 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
  – Segment information
  – “rep” specifiers
    • Would lead to 100’s of bits
    • Common case only needs a fraction $\rightarrow$ a lot of waste

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

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

```
ADD EAX, EBX → ADD EAX, EBX  1 uop
ADD EAX, [EBX] → Load tmp = [EBX]
                  ADD EAX, tmp  2 uops
ADD [EAX], EBX → Load tmp = [EAX]
                  ADD tmp, EBX
                  STA EAX
                  STD tmp  4 uops
```
uop Limits

• How many uops can a decoder generate?
  – For complex x86 insts, many are needed (10’s, 100’s?)
  – Makes decoder horribly complex
  – Typically there’s a limit to keep complexity under control
    • One x86 instruction $\rightarrow$ 1-4 uops
    • Most instructions translate to 1.5-2.0 uops

• What if a complex insn. needs more than 4 uops?
UROM/MS for Complex x86 Insts

- UROM (microcode-ROM) stores the uop equivalents
  - Used for nasty x86 instructions
    - Complex like > 4 uops (PUSHA or STRREP.MOV)
    - Obsolete (like AAA)

- Microsequencer (MS) handles the UROM interaction
UROM/MS Example (3 uop-wide)

**Cycle 1**
- ADD
- SUB
- REP.MOV
- SUB
- ADD [ ]
- ADD
- STA
- STD

**Cycle 2**
- REP.MOV
- ADD [ ]
- XOR
- LOAD
- STORE

**Cycle 3**
- ADD [ ]
- INC
- STORE
- LOAD
- mJCC

**Cycle 4**
- ADD [ ]
- XOR
- mJCC
- INC
- mJCC

**Cycle 5**
- ADD [ ]
- XOR
- mJCC
- LOAD
- ADD

**Cycle n**
- ADD [ ]
- XOR
- mJCC
- LOAD
- ADD

**Cycle n+1**
- ADD [ ]
- XOR
- mJCC
- LOAD
- ADD

Complex instructions, get uops from mcode sequencer

Fetch - x86 insts
Decode - uops
UROM - uops
Superscalar CISC Decode

- Instruction Length Decode (ILD)
  - Where are the instructions?
    - Limited decode – just enough to parse prefixes, modes

- Shift/Alignment
  - Get the right bytes to the decoders

- Decode
  - Crack into uops

Do this for N instructions per cycle!
ILD Recurrence/Loop

\[ PC_i = X \]
\[ PC_{i+1} = PC_i + \text{sizeof}( \text{Mem}[PC_i] ) \]
\[ PC_{i+2} = PC_{i+1} + \text{sizeof}( \text{Mem}[PC_{i+1}] ) \]
\[ = PC_i + \text{sizeof}( \text{Mem}[PC_i] ) + \text{sizeof}( \text{Mem}[PC_{i+1}] ) \]

- Can’t find start of next insn. before decoding the first
- Must do ILD serially
  - ILD of 4 insns/cycle implies cycle time will be 4x

Critical component not pipeline-able
Bad x86 Decode Implementation

ILD dominates cycle time; not scalable
Hardware-Intensive Decode

Decode from every possible instruction starting point!

Giant MUXes to select instruction bytes
ILD in Hardware-Intensive Approach

Previous: $3 \times \text{ILD} + 2 \times \text{add}$
Now: $1 \times \text{ILD} + 2 \times (\text{mux} + \text{add})$
Predecoding

• ILD loop is hardware intensive
  – Impacts latency
  – Consumes substantial power

• If instructions A, B and C are decoded
  – ... lengths for A, B, and C will still be the same next time
  – No need to repeat ILD

Possible to cache the ILD work
Decoder Example: AMD K5 (1/2)

From Memory
8 bytes → Predecode Logic
13 bytes → I$

8 bits
$b_0 \ | \ b_1 \ | \ b_2 \ | \ ... \ | \ b_7$

+5 bits

8 × (8-bit inst + 5-bit predecode)

Decode

16 × (8-bit inst + 5-bit predecode)

Up to 4 ROPs

Compute ILD on fetch, store ILD in the I$
Decoder Example: AMD K5 (2/2)

• Predecode makes decode easier by providing:
  – Instruction start/end location (ILD)
  – Number of ROPs needed per inst
  – Opcode and prefix locations

• Power/performance tradeoffs
  – Larger I$ (increase data by 62.5%)
    • Longer I$ latency, more I$ power consumption
  – Remove logic from decode
    • Shorter pipeline, simpler logic
    • Cache and reused decode work → less decode power
  – Longer effective I-L1 miss latency (ILD on fill)
Decoder Example: Intel P-Pro

- Only Decoder 0 interfaces with the uROM and MS
- If insn. in Decoder 1 or Decoder 2 requires > 1 uop
  1) do not generate output
  2) shift to Decoder 0 on the next cycle