Other ISAs - ARM, x86

Hwansoo Han
ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>15 GPR × 32-bits</td>
<td>31 GPR × 32-bits</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
Compare and Branch (ARM)

- ARM uses **condition codes** for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
  - vs. MIPS uses the contents of registers

- Each instruction can be conditional
  - Predicated execution / conditional execution
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions (if-conversion)
Conditional Execution (ARM)

```c
while (a!=b) {
    if (a>b) a -= b; else b -= a;
}
```

Assume a in r1 and b in r2

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>CMP r1, r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEQ complete</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BLT lessthan</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB r1, r1, r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B gcd</td>
<td></td>
</tr>
<tr>
<td>lessthan</td>
<td>SUB r2, r1, r2</td>
<td></td>
</tr>
<tr>
<td>complete</td>
<td>B gcd</td>
<td></td>
</tr>
</tbody>
</table>

ARM code with conditional execution

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>CMP r1, r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUBGT r1, r1, r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUBLT r2, r2, r1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BNE gcd</td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Encoding (ARM vs. MIPS)

<table>
<thead>
<tr>
<th>Category</th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register-register</strong></td>
<td><img src="#" alt="Register-register ARM" /></td>
<td><img src="#" alt="Register-register MIPS" /></td>
</tr>
<tr>
<td><strong>Data transfer</strong></td>
<td><img src="#" alt="Data transfer ARM" /></td>
<td><img src="#" alt="Data transfer MIPS" /></td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td><img src="#" alt="Branch ARM" /></td>
<td><img src="#" alt="Branch MIPS" /></td>
</tr>
<tr>
<td><strong>Jump/Call</strong></td>
<td><img src="#" alt="Jump/Call ARM" /></td>
<td><img src="#" alt="Jump/Call MIPS" /></td>
</tr>
</tbody>
</table>

![Legend](#)
The Intel x86 ISA

- Evolution with backward compatibility
  - **8080 (1974):** 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - **8086 (1978):** 16-bit extension to 8080
    - Complex instruction set (CISC)
  - **8087 (1980):** floating-point coprocessor
    - Adds FP instructions and register stack
  - **80286 (1982):** 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - **80386 (1985):** 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments
The Intel x86 ISA (cont’d)

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
    - New microarchitecture (see Colwell, The Pentium Chronicles)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions
The Intel x86 ISA (cont’d)

- And further…
  - **AMD64 (2003): extended architecture to 64 bits**
  - **EM64T – Extended Memory 64 Technology (2004)**
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - **Intel Core (2006)**
    - Added SSE4 instructions, virtual machine support
  - **AMD64 (announced 2007): SSE5 instructions**
    - Intel declined to follow, instead…
  - **Advanced Vector Extension (announced 2008)**
    - Longer SSE registers, more instructions
  - If Intel didn’t extend with compatibility, its competitors would!
  - Technical elegance ≠ market success
# Basic x86 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR 7</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>

---

9
Basic x86 Addressing Modes

- Two operands per instruction

<table>
<thead>
<tr>
<th>Source/dest operand</th>
<th>Second source operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

- Memory addressing modes
  - Address in register (register indirect)
  - Address = $R_{base} + \text{displacement}$
    - (8- or 32-bit displacement)
  - Address = $R_{base} + 2^{\text{scale}} \times R_{\text{index}}$
    - (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

[restrictions on register use]
x86 Instruction Encoding

- Variable length encoding
  - 1 byte ~ 15 bytes
  - Opcode byte contains
    - Operand length (8 or 32 bits)
  - Some instructions, opcode byte include
    - Addressing mode
    - Register operand
  - Others use extra byte to specify addressing mode

Typical x86 instruction formats
Implementing IA-32

- Complex ISA makes implementation difficult
  - Hardware translates instructions to simpler micro-operations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this added complexity economically viable

- Comparable performance to RISC
  - Compilers avoid complex instructions that are hard to implement fast
CISC Instruction Sets

- Complex Instruction Set Computer
  - Dominant style through mid-80’s
- Arithmetic instructions can access memory
  - `addl %eax, 12(%ebx,%ecx,4)`
    - Requires memory read and write
    - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions
- Philosophy
  - Add instructions to perform “typical” programming tasks
RISC Instruction Sets

- Reduced Instruction Set Computer
  - Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
- Fewer, simpler instructions
  - Might take more instructions to get given task done
  - Can execute them with small and fast hardware
- Register-oriented instruction set
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- Only load and store instructions can access memory
- No Condition codes
  - Test instructions return 0/1 in register
CISC vs. RISC

Original Debate
- Strong opinions!
- CISC proponents - easy for compiler, fewer code bytes
- RISC proponents - better for optimizing compilers, can make them run fast with simple chip design

Current Status
- For desktop processors, choice of ISA is not a technical issue
  - With enough hardware, can make anything run fast
  - Code compatibility is more important
- For embedded processors, RISC makes sense
  - Smaller, cheaper, less power
Fallacies

- Powerful instruction $\Rightarrow$ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code $\Rightarrow$ more errors and less productivity
Fallacies

- Backward compatibility $\Rightarrow$ instruction set doesn’t change
  - But they do add more instructions ($\approx 1$ instr/month)
  - Competitors had difficulties to make compatible processors

![Graph showing the growth of the x86 instruction set over time]
Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!

- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped
Conclusions

- **Design principles**
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

- **Layers of software/hardware**
  - Compiler, assembler, hardware
  - ISA is a very important abstraction between SW and HW

- **MIPS: typical of RISC ISAs**
  - c.f. x86 – typical of CISC ISAs
Conclusions (cont’d)

- Measure MIPS instruction executions in benchmarks
  - Consider making the common case fast
  - Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lhu, sb, lui</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, ori, sll, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltiu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>