Computer Organization II

Lecture 8: Pentium, ARM, RISC


Lecture 8: Pentium, ARM, RISC

CPU Examples & RISC

- x86/ARM

Ch 12.5-6 [Sta10]

Ch 13 [Sta10]

Instruction analysis

RISC vs. CISC

Register use

Comp. Org II, Spring 2012

1

X86 architecture

(e.g., Pentium)

X86 Processor Registers

(Sta10 Table 12.21)

(a) Integer Unit in 32-bit Mode

<table>
<thead>
<tr>
<th>Type</th>
<th>Number</th>
<th>Length (bits)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>8</td>
<td>32</td>
<td>General-purpose user registers</td>
</tr>
<tr>
<td>Segment</td>
<td>6</td>
<td>16</td>
<td>Contains register selectors</td>
</tr>
<tr>
<td>EIP</td>
<td>1</td>
<td>32</td>
<td>Stack and control bits</td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td>2</td>
<td>32</td>
<td>Instruction pointer</td>
</tr>
</tbody>
</table>

(b) Integer Unit in 64-bit Mode

<table>
<thead>
<tr>
<th>Type</th>
<th>Number</th>
<th>Length (bits)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>16</td>
<td>32</td>
<td>General-purpose user registers</td>
</tr>
<tr>
<td>Segment</td>
<td>6</td>
<td>16</td>
<td>Contains register selectors</td>
</tr>
<tr>
<td>EIP</td>
<td>1</td>
<td>64</td>
<td>Stack and control bits</td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td>1</td>
<td>64</td>
<td>Instruction pointer</td>
</tr>
</tbody>
</table>

Pentium: FP / MMX Registers

Floating-Point Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>M137</th>
<th>M136</th>
<th>M135</th>
<th>M134</th>
<th>M133</th>
<th>M132</th>
<th>M131</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Pentium: EFLAGS Register

Condition of the processor: carry, parity, auxiliary, zero, sign, and overflow

Programmed responsibility

Discussion


Discussion?
Pentium: Control Registers

- **Page Directory Base**
- **Page Fault Base**
- **Page Fault Lease Address**
- **Not used!**

**System control flags**

- **Code Disable**
- **Non-Write Through**
- **Physical Address Extension**
- **Page-Table Extension**
- **Task-Address Extension**
- **Task-Size Disable**
- **Task-Segmentation**
- **Segment-Size Disable**
- **Page-Level Cache Disable**
- **Page-Level Write Transparency**

**Pentium: Interrupts**

- **Calling interrupt handler; atomic hardware functionality!**
  - PUSH(SS) stack segment selector to stack
  - PUSH(ESP) stack pointer to stack
  - PUSH(EFLAGS) status register to stack
  - EFLAGS.IOPL 00 set privileged mode
  - EFLAGS.IF 0 disable interrupts (keskeytys)
  - EFLAGS.TF 0 disable exceptions (poikkeus)
  - PUSH(CS) code segment selector to stack
  - PUSH(EIP) instruction pointer to stack (Adakyoosäätö)
  - PUSH(error code) if needed

- **Return**
  - Privileged IRET-instruction
  - POP everything from stack to their places

**ARM (Ch 12.6 Sta10)**

- **Array of uniform registers (moderate number)**
- **Fixed length (32 bit) instruction (Thumb 16 bit)**
- **Load/Store architecture**
- **Small number of addressing modes (reg + instr. field)**
- **Autoincrement addressing mode (for program loops)**
- **Data processing instructions allow shift or rotate to preprocess one of source regs**
  - Separate ALU and shifter for this purpose (avoid structural dependency or hazard)
- **Conditional execution of instructions**
  - Fewer conditional branches, improves pipeline efficiency

**ARM Processor Organization**

- Varies substantially - different versions of ARM architecture
- Simplified, generic organization
- Register file: set of 32-bit registers, total 37 regs
  - 31 general-purpose regs
  - 6 status regs
  - Partially overlapping banks (=only 16 visible at a time)
ARM Processor execution modes

- **User mode**
  - No access to protected system resources, can cause exception
- **Supervisor mode**
  - For OS, starts with software interrupt instruction
  - For memory faults
  - Instruction not supported
  - Fast interrupt mode
  - Interrupt from designated fast interrupt source
  - Not interruptable, can interrupt normal interrupt
- **Interrupt mode**
  - Any other interrupt signal, can be interrupted by fast interrupt
- **System mode**
  - Only for certain privileged OS tasks

Exception modes

- **User mode**
  - No access to protected system resources, can cause exception
- **Supervisor mode**
  - For OS, starts with software interrupt instruction
- **Abort mode** – due to memory faults
- **Undefined mode** – instruction not supported
- **Fast interrupt mode**
  - Interrupt from designated fast interrupt source
  - Not interruptable, can interrupt normal interrupt
- **Interrupt mode**
  - Any other interrupt signal, can be interrupted by fast interrupt
- **System mode**
  - Only for certain privileged OS tasks

ARM Program status regs (CPSR & SPSR)

- **CPSR**
  - **N, Z, C, V** – condition code
  - **E** – endianness in load/store
  - **Q** – overflow or saturation in SIMD-orient. instr.
  - **J** – Jazelle instruction in use
  - **GE[3:0]** – for SIMD as greater than or equal flags for individual bytes or halfwords of the result
  - **M[4:0]** – processor mode

- **SPSR**
  - **S** – supervisor mode
  - **A** – abort mode
  - **U** – undefined mode
  - **I** – interrupt mode

ARM Interrupt vector

Table lists the exception types and the address in the interrupt vector for that type.

The vector contains the start addresses of the interrupt handlers.

Hardware milestones

- **IBM S/360, DEC PDP-8**
- **IBM S/360**
- **Atlas**
- **Univac**
- **IBM, Intel**
- **Intel, Sony-Toshiba-IBM**
- **Maurice Wilkes**
- **Tom Kilburn**
- **Gene Amdahl**
- **J.P. Eckert, John Mauchly**
- **Maurice Wilkes**
- **John Cocke, 1974, IBM 801**
- **J.L. Hennessy & D.A. Patterson**
- **John Cocke, 1965**
- **Intel**
- **IBM**
- **CDC, 1964**
- **Intel**
- **IBM**
- **Multicore CPU, 2005**
  - Several full processors on chip
- **Hyperthreading CPU, 2001**
  - Several register sets and virtual processors on chip
- **Superscalar CPU, 1989**
  - Multiple instruction per cycle
- **Cache, 1965**
  - Test_and_set instruction needed
- **Virtual memory, 1962**
  - Simpler memory management
- **Pipeline, 1962**
  - Architecture family concept, 1964
  - Set of computers using the same instruction set
  - Microprogrammed control, 1964
  - Easier control design and impl.
  - Multiple processors, 1964
  - test_and_set instruction needed
  - Cache, 1965
- **RISC-architecture, 1980**
  - Simple instruction set
  - Superscalar CPU, 1989
  - Multiple instruction per cycle
  - Hyperthreading CPU, 2001
  - Several register sets and virtual processors on chip
- **Multicore CPU, 2005**
  - Several full processors on chip
### CISC - Complex Instruction Set Computer

- **Goal**: Shrink the semantic gap (semantinen kuilu) between high-level language and machine instruction set.
- **Expressiveness of high-level languages had increased** (in 1970’s)
- **Wanted “simple” compilations**
- **Lots of different instructions for different purposes**
- **Lots of different data types** (int, float, char, boolean, …)
- **Lots of different addressing modes**
- **Complex tasks performed in hardware by control unit (single instruction)**, not in the machine code level (multiple instructions)
  - Less instructions in one program (shorter code)
  - Efficient (just a few instructions) execution of complex tasks

### Observations on Subroutine (procedure, function) calls?

- **Lots of subroutine calls**
- **Calls rarely have many parameters**
- **Nested calls** are rare

<table>
<thead>
<tr>
<th>Percentage of Executed Procedures Calls with Various Types of Arguments</th>
<th>Compiler, Interpreters, and Typewriters</th>
<th>Small Numeric Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;3 arguments</td>
<td>0.7%</td>
<td>0.5%</td>
</tr>
<tr>
<td>&gt;8 arguments</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>&gt;12 local variables</td>
<td>1.3%</td>
<td>0.6%</td>
</tr>
<tr>
<td>&gt;12 total arguments and local variables</td>
<td>1.3%</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

98% less than 6 parameters
92% less than 6 local variables

**How to use the information?**

### Observations from Real Programs

- **Most operands are simple**
- **Many jumps and branches**
- **Compilers do not always use the complex instructions**
  - They use only a subset of the instruction set
  - Easier to do? Faster?

**Conclusion?**

**Occam’s razor**

Entia non sunt multiplicanda praeter necessitatem

**William Of Occam (1300-1349)**

**English monk, philosopher**

It is vain to do with more that which can be done with less

### Optimize for Execution Speed

- **Optimize the parts that consume most of the time**
  - Procedure calls, loops, memory references, addressing, ...
- **Avoid optimizing rare events**
  - Rarely used (10%) floating point instructions improved to run 2x:
    
    \[
    \text{Speedup} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{0.95} = 1.053 \ll 2
    \]

**Amdahl’s law**

Speedup due to an enhancement is proportional to the fraction of the time (in the original system) that the enhancement can be used.

**Gene Amdahl**

### Which Operations and Operands Are Used?

- **Year 1982**, computers VAX, PDP-11, Motorola 68000
- **Observe dynamic execution time behaviour**

<table>
<thead>
<tr>
<th>Dynamic Percentage of Operations Used</th>
<th>80% of references to local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>2%</td>
</tr>
<tr>
<td>LOCAL</td>
<td>5%</td>
</tr>
<tr>
<td>CALL</td>
<td>1%</td>
</tr>
<tr>
<td>RET</td>
<td>2%</td>
</tr>
<tr>
<td>OTHER</td>
<td>9%</td>
</tr>
</tbody>
</table>

### RISC Approach

- **Optimize design for execution speed, instead of ease of compilation**
  - Compilers are good, machines are efficient
  - Do most important, common things in hardware and fast
    - E.g. 1-dim array reference
      - One machine instruction
    - And the rest in software (and slow)
      - E.g. multidimensional arrays, string processing, ...
      - Library routines for these

**RISC architecture** (Reduced Instruction Set Computer)
**RISC architecture**
- Plenty of registers (minimum 32)
- Compilers optimize register usage
- LOAD/STORE architecture
- Only LOAD and STORE do memory referencing
- Small set of simple instructions
- Simple, fixed-length instruction format (32b)
- Instruction fetch and decoding simple and efficient
- Small selection of simple addresses references
- Fast address translation
- Limited set of different operands
  - 32b integers, floating point
- One or more instructions are completed on each cycle

**RISC vs. CISC**

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>CISC (dark)</th>
<th>RISC (light)</th>
<th>Supercomputer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year developed</td>
<td>2010</td>
<td>2000</td>
<td>1980</td>
</tr>
<tr>
<td>Number of instructions</td>
<td>100,000</td>
<td>20,000</td>
<td>5,000</td>
</tr>
<tr>
<td>Addressing modes</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Number of general purpose registers</td>
<td>16</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Current memory size (GB)</td>
<td>32</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Card size (KB)</td>
<td>64</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Characteristics of Some CISCs, RISCs, and Supercomputers:

**Register Window to Register File**
- More physical registers than addressable in the instruction
- E.g., SPARC has just 5 bits for register number \( 0..31 \),
  but the processor has 40 to 540 registers
- Small subset of registers available for each instruction in register window
- In the window references to register \( r0..r31 \)
- CPU maps them to actual (true) registers \( r0..r539 \)

**Register Files**

```
Register Window to Register File
```

```
Computer Organization II 7.2.2012
Comp. Org II, Spring 2012 5
```
Overlapped Register Windows

- Procedure parameters passed in registers (not in stack)
- Fixed number of registers for parameters, local variables, and return value passed via overlapped register window
- Overlapping area to allow parameter passing to the next procedure and back to caller

Circular Buffer for Overlapped Register Window

- Too many nested calls?
- Most recent calls in registers
- Old activations saved to memory
- Restore when nesting depth decreases
- Overlap only when needed
- Global variables?
- In memory or own register window

SPARC
- r0-r7 global var.
- r8-r15 parameters (in caller)
- r16-r23 local variables
- r24-r31 parameters (to called)

Register File vs. Cache

<table>
<thead>
<tr>
<th>Large Register File</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>All local variables</td>
<td>Indirect memory access</td>
</tr>
<tr>
<td>Private variables</td>
<td>Direct memory access</td>
</tr>
<tr>
<td>Compiler-assigned global variables</td>
<td>recently used local variables</td>
</tr>
<tr>
<td>Slow-Sequential address</td>
<td>recently-used global variables</td>
</tr>
<tr>
<td>Register addressing</td>
<td>Number of bits</td>
</tr>
<tr>
<td></td>
<td>Memory addressing</td>
</tr>
</tbody>
</table>

- The register file acts like a small, fast buffer (as cache)?
- Register is faster, needs less bits in addressing, but
- It is difficult for compiler to determine in advance, which of the global variables to place in registers
- Cache decides this issue dynamically
- Most used and referenced data stay in cache

Allocation of registers (compiler-based register optimization)

- Node (solmu) = symbolic register
- Edge (särö) = symbolic registers used at the same time
- n colors = n registers

Compiler-based register optimization (allocation of registers)

- Problem: Graph coloring
- Minimize the number of different colors, while adjacent nodes have different color
- = Difficult problem (NP-complete)

- Form a network of symbolic registers based on the program code
- Symbolic register - any program quantity that could be in register
- The edges of the graph join together program quantities that are used in the same code fragment
- Allocate real registers based on the graph
- Two symbolic registers that are not used at the same time (no edge between them) can be allocated to the same real register (use the same color)
- If there are no more free registers, use memory addresses

RISC-pipeline

- Instruction fetch
- Execute E1: reg read, E2: Alu + reg write
- Data memory op

Details:
- Two-step program memory
- Two port MEM, or faster mem (split cache enough?)
RISC-pipeline, Delayed Branch

- Traditional pipeline
- RISC with inserted NOOP
- Two port MEM
- No need to clear pipeline (NOOP)
- RISC with reversed instructions (use of delay slot)
- What if conditional branch?
  - JZERO 105, rA
  - (need ADD 1, rA result before comparison, cannot use delay slot)

Extra gain: Dependency problem also solved!

RISC & CISC United?

- Pentium, CISC
  - Each 1–11 byte-length CISC-instruction is translated by hardware to 1–4 118-bit micro-operations (stored in L1 instruction cache)
- Lower levels (including control unit) as RISC
- Lots of work registers, visible only to hardware
- Crusoe (Transmeta)
  - Emulate Intel architecture with simpler HW architecture
  - Outside looks like Intel CISC-architecture
  - Group of instructions ‘translated’ by software, just before execution, to fixed-length micro-operations, these can be optimized before execution
  - VLIW (very long instruction word, 128 bits)
  - 4 P ops/VLIW-instruction
- Lower levels as RISC

Just in time (JIT) compilation

CISC to RISC "compilation" at every execution

"compilation" just once per group


Summary

- X86 and ARM processor implementation examples
- Registers, addressing modes, instruction sets
- What is CISC? What is "wrong" with CISC
- What is RISC? What is "good" with RISC?
- Lots of registers, load-store arch
- Small set of simple instructions with just a few operand types
- Simple instruction formats and addressing formats
- How to get more from HW registers?
- Register windows to register file
- Overlapping register windows
- Register file vs. cache?
- Register allocation problem and its solution
- Combine RISC with CISC?

Review Questions

- Main features and characteristics of RISC-architecture?
- What makes RISC RISC?
- Which addressing format is not RISC?
- Which operation type is not RISC?
- Which instruction format is not RISC?
- Which operand type is not RISC?
- Why would large L1 cache be better than large register file?
- How are register windows used?
- When would n overlapped registers be enough?
- What happens if n overlapped registers is not enough?