CPU Examples & RISC

Ch 12.5-6 [Sta10]
- x86/ARM

Ch 13 [Sta10]
- Instruction analysis
- RISC vs. CISC
- Register use

Computer Organization II

X86 architecture (e.g., Pentium)
### X86 Processor Registers

#### (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>Contain segment selectors</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>1</td>
<td>32</td>
<td>Status and control bits</td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td>1</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>Contain segment selectors</td>
</tr>
<tr>
<td>RFLAGS</td>
<td>1</td>
<td>64</td>
<td>Status and control bits</td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td>1</td>
<td>64</td>
<td>Instruction pointer</td>
</tr>
</tbody>
</table>

#### (c) Floating-Point Unit

<table>
<thead>
<tr>
<th>Type</th>
<th>Number</th>
<th>Length (bits)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>8</td>
<td>80</td>
<td>Hold floating-point numbers</td>
</tr>
<tr>
<td>Control</td>
<td>1</td>
<td>16</td>
<td>Control bits</td>
</tr>
<tr>
<td>Status</td>
<td>1</td>
<td>16</td>
<td>Status bits</td>
</tr>
<tr>
<td>Tag Word</td>
<td>1</td>
<td>16</td>
<td>Specifies contents of all registers</td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td>1</td>
<td>48</td>
<td>Points to instruction interrupted by exception</td>
</tr>
<tr>
<td>Data Pointer</td>
<td>1</td>
<td>48</td>
<td>Points to operand interrupted by exception</td>
</tr>
</tbody>
</table>

**Note:**
- EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI
- CS, SS, DS, ES, FS, GS
- EFLAGS

*Source: Sta10 Table 12.2*
Pentium: FP / MMX Registers

- Aliasing
- FP regs used as stack
  - Intel 8087 coprocessor (1980)
- MMX multimedia instructions use the same registers, but use them directly
- MMX-usage: bits 64-79 are set to 1 → NaN
- FP Tag (word) indicate which usage is current
  - First MMX instr. set
  - EMMS (Empty MMX State) instruction reset

Programmer responsibility

Discussion?

Pentium: EFLAGS Register

- ID = Identification flag
- VIP = Virtual interrupt pending
- VIF = Virtual interrupt flag
- AC = Alignment check
- VM = Virtual 8086 mode
- RF = Resume flag
- NT = Nested task flag
- IOPL = I/O privilege level
- OF = Overflow flag
- DF = Direction flag
- IF = Interrupt enable flag
- TF = Trap flag
- SF = Sign flag
- ZF = Zero flag
- AF = Auxiliary carry flag
- PF = Parity flag
- CF = Carry flag

- Condition of the processor: carry, parity, auxiliary, zero, sign, and overflow
- Used in conditional branches
### Pentium: Control Registers

<table>
<thead>
<tr>
<th>CR4</th>
<th>CR3</th>
<th>CR2</th>
<th>CR1</th>
<th>CR0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Page Directory Base</td>
<td>Page Fault Linear Address</td>
<td>Not used!</td>
</tr>
</tbody>
</table>

#### Flags
- **PCE** = Performance Counter Enable
- **POE** = Page Global Enable
- **MCE** = Machine Check Enable
- **FAE** = Physical Address Extension
- **PSE** = Page Size Extension
- **DF** = Debug Extension
- **TSD** = Time Stamp Disable
- **PVI** = Protected Mode Virtual Interrupt
- **VME** = Virtual 8086 Mode Extensions
- **PCD** = Page-level Cache Disable
- **PWT** = Page-level Write Transparency

#### System Control Flags
- **CD** = Cache Disable
- **NW** = Not Write Through
- **AM** = Alignment Mask
- **WP** = Write Protect
- **NE** = Numa Node Error
- **ET** = Extension Type
- **TS** = Task Switched
- **EM** = Emulation
- **MP** = Monitor/Coprocessor
- **PF** = Prefetchable

See Sta10 Fig 12.23

### Pentium: Interrupts

- Calling interrupt handler; atomic hardware functionality!
- If not in privileged mode (*etuokeutettu tila*)
  - 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 (keskeyys)
  - EFLAGS.TF ← 0 disable exceptions (poikkeus)
  - PUSH(CS) code segment selector to stack
  - PUSH(EIP) instruction pointer to stack (käskyosoitin)
  - PUSH(error code) if needed

#### Interrupts
- **CS** ← interrupt vector [number].CS
- **EIP** ← interrupt vector [number].EIP

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

See Sta10 Table 12.3
### Exception and Interrupt Vector Table

<table>
<thead>
<tr>
<th>Vector Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error, division overflow or division by zero</td>
</tr>
<tr>
<td>1</td>
<td>Debug exception; includes various faults and traps related to debugging</td>
</tr>
<tr>
<td>2</td>
<td>NMI pin interrupt; issued on NMI pin</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint, caused by INT 3 instruction, which is a 1-byte instruction useful for debugging</td>
</tr>
<tr>
<td>4</td>
<td>INTO-detected overflow; occurs when the processor executes INTO with the OF flag set</td>
</tr>
<tr>
<td>5</td>
<td>BOUND range-exceeded; the BOUND instruction compares a register with boundaries stored in memory and generates an interrupt if the contents of the register is out of bounds</td>
</tr>
<tr>
<td>6</td>
<td>Undefined opcode</td>
</tr>
<tr>
<td>7</td>
<td>Device not available; attempt to use I$ or WAIT instructions fails due to lack of external device</td>
</tr>
<tr>
<td>8</td>
<td>Double fault; two interrupts occur during the same instruction and cannot be handled serially</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Invalid task state segment; segment describing a suspended task is not initialized or not valid</td>
</tr>
<tr>
<td>11</td>
<td>Segment not present; required segment not present</td>
</tr>
<tr>
<td>12</td>
<td>Stack fault: limit of stack segment exceeded or stack segment not present</td>
</tr>
<tr>
<td>13</td>
<td>General protection; protection violation that does not cause another exception (e.g., writing to a read-only segment)</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
</tr>
<tr>
<td>16</td>
<td>Floating-point error; generated by a floating-point arithmetic instruction</td>
</tr>
<tr>
<td>17</td>
<td>Alignment check; access to a word stored at an odd-byte address or a doubleword stored at an address not a multiple of 4</td>
</tr>
<tr>
<td>18</td>
<td>Machine check; model specific</td>
</tr>
<tr>
<td>19-31</td>
<td>Reserved</td>
</tr>
<tr>
<td>32-51</td>
<td>User interrupt vector; provided when INTn signal is asserted</td>
</tr>
</tbody>
</table>

Unmasked exceptions are provided when INTn signal is asserted. (Sta10 Table 12.3)
ARM features

- 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

- 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 Register organization

- SP – stack pointer
- LR – link register (return address & mode)
- PC – program counter
- CPSR – current program status register
- SPSR – saved program status register

Shaded regs replaced in exception modes!

Diagrams:
- Fig 12.26
ARM Program status regs (CPSR & SPSR)

```
<table>
<thead>
<tr>
<th>User flags</th>
<th>System control flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>N, Z, C, V - condition code</td>
<td>E - endianness in load/store</td>
</tr>
<tr>
<td>Q - overflow or saturation in SIMD-orient. instr.</td>
<td>A, I, F - interrupt disable bits (A - imprecise data aborts, I - normal IRQ, F - fast FIQ)</td>
</tr>
<tr>
<td>J - Jazelle instruction in use</td>
<td>T - normal / Thumb instr.</td>
</tr>
<tr>
<td>&quot;Java byte code mode&quot;</td>
<td>M[4:0] - processor mode</td>
</tr>
<tr>
<td>GE[3:0] - for SIMD as greater than or equal flags for individual bytes or halfwords of the result</td>
<td></td>
</tr>
</tbody>
</table>
```

**ARM Interrupt vector**

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

The vector contains the start addresses of the interrupt handlers.

### ARM Interrupt vector

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Processor Mode</th>
<th>Normal entry address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>0x00000000</td>
<td>Occurs when the system is initialized.</td>
</tr>
<tr>
<td>Data abort</td>
<td>Abort</td>
<td>0x00000010</td>
<td>Occurs when an invalid memory address has been accessed, such as if there is no physical memory for an address or the correct access permission is lacking.</td>
</tr>
<tr>
<td>FIQ (Fast interrupt)</td>
<td>FIQ</td>
<td>0x0000001C</td>
<td>Occurs when an external device asserts the FIQ pin on the processor. An interrupt cannot be interrupted except by an FIQ. FIQ is designed to support a data transfer or channel process and has sufficient private registers to accommodate the need for register saving in such applications, therefore minimizing the overhead of context switching. A fast interrupt cannot be interrupted.</td>
</tr>
<tr>
<td>IRQ (Interrupt)</td>
<td>IRQ</td>
<td>0x00000018</td>
<td>Occurs when an external device asserts the IRQ pin on the processor. An interrupt cannot be interrupted except by an FIQ.</td>
</tr>
<tr>
<td>Prefetch abort</td>
<td>Abort</td>
<td>0x0000000C</td>
<td>Occurs when an attempt to fetch an instruction results in a memory fault. The exception is raised when the instruction enters the execute stage of the pipeline.</td>
</tr>
<tr>
<td>Undefined instructions</td>
<td>Undef</td>
<td>0x00000004</td>
<td>Occurs when an instruction not in the instruction set reaches the execute stage of the pipeline.</td>
</tr>
<tr>
<td>Software interrupt</td>
<td>Supervisor</td>
<td>0x00000008</td>
<td>Generally used to allow user mode programs to call the OS. The user program executes a SWI instruction with an argument that identifies the function the user wishes to perform.</td>
</tr>
</tbody>
</table>

(Sta10 Fig 12.27)
RISC-architecture

Ch 13 [Sta10]
- Instructions
- RISC vs. CISC
- Register allocation

Hardware milestones
- 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
      - Huge improvement in performance
- 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

IBM, Intel
- Maurice Wilkes
  - Intel, Sony-Toshiba-IBM
  - Maurice Wilkes
  - Tom Kilburn
  - J.P. Eckert, John Mauchly
  - J.L. Hennessy & D.A. Patterson
  - John Cocke, IBM 801
  - J.L. Hennessy & D.A. Patterson
  - John Cocke, 1965
  - CDC, 1964
  - Intel
  - IBM
CISC - Complex Instruction Set Computer

- Goal: Shrink the semantic gap (semanttinen kuilu) between high-level language and machine instruction set
  - Expressiveness of high-level languages had increased (in 1970’s...)
  - Wanted “simple” compilations
    - Language structures match nicely with instructions
  - 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

Which Operations and Operands Are Used?

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

<table>
<thead>
<tr>
<th>Dynamic Occurrence</th>
<th>Machine-Instruction Weighted</th>
<th>Memory-Reference Weighted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal C</td>
<td>Pascal C</td>
<td>Pascal C</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>15%</td>
<td>38%</td>
</tr>
<tr>
<td>LOOPE</td>
<td>5%</td>
<td>43%</td>
</tr>
<tr>
<td>CALL</td>
<td>15%</td>
<td>21%</td>
</tr>
<tr>
<td>IF</td>
<td>26%</td>
<td>44%</td>
</tr>
<tr>
<td>GOTO</td>
<td>3%</td>
<td>3%</td>
</tr>
<tr>
<td>OTHER</td>
<td>6%</td>
<td>3%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weighted Relative Dynamic Frequency of HLL Operations [PATT92a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(HLL=High Level Language)</td>
</tr>
<tr>
<td>Dynamic Percentage of Operands</td>
</tr>
<tr>
<td>Integer Constant</td>
</tr>
<tr>
<td>Scalar Variable</td>
</tr>
<tr>
<td>Array/Structure</td>
</tr>
</tbody>
</table>
Observations on Subroutine (procedure, function) calls?

- Lots of subroutine calls
- Calls rarely have many parameters
- Nested (sisäkkäinen) calls are rare

<table>
<thead>
<tr>
<th>Percentage of Executed Procedure Calls With</th>
<th>Compiler, Interpreter, and Typesetter</th>
<th>Small Nonnumeric Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;3 arguments</td>
<td>0–7%</td>
<td>0–5%</td>
</tr>
<tr>
<td>&gt;5 arguments</td>
<td>0–3%</td>
<td>0%</td>
</tr>
<tr>
<td>&gt;8 words of arguments and local scalars</td>
<td>1–20%</td>
<td>0–6%</td>
</tr>
<tr>
<td>&gt;12 words of arguments and local scalars</td>
<td>1–6%</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 (Occamin partaveitsi)

"Entia non sunt multiplicanda praeter necessitatem" ("Entities should not be multiplied more than necessary")

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:

  \[
  \text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 \times 1.0 + 0.1 \times 0.5) \\
  = 0.95 \times \text{ExTime}_{\text{old}} \\
  \text{Speedup} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{0.95} = 1.053 \ll 2
  \]

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.*

RISC Approach

- Optimize design for execution speed, instead of ease of compilation
  - Compilers are good, machines are efficient
    - Compiler can and has time to do the optimization
  - 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
    - Many machine instructions

⇒ 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 address references
  - No indirect memory reference
  - Fast address translation
- Limited set of different operands
  - 32b integers, floating-point
- One or more instructions are **completed** on each cycle

**RISC architecture**

- CPU easier to implement
  - Pipeline control and optimization simpler
  - Hardwired (langoitettu) control
- Smaller chip (lastu) size
  - More chips per die (kiekko)
  - Smaller waste%
- Cheaper manufacturing
- Faster marketing
  - Design-to-market time

---

http://www.gadget-paradise.com/0127/samsung/compssd/64gb-flash-memory/

Large chips vs. small chips?

- 25% yield (good chips)
- 75% wasted
- 55% yield (good chips)
- 45% wasted
### RISC vs. CISC

#### Table: Characteristics of Some CISCs, RISCs, and Superscalar Processors

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>IBM 370/168</th>
<th>VAX 11/780</th>
<th>PDP 11/40</th>
<th>SPARC</th>
<th>MIPS R4000</th>
<th>PowerPC</th>
<th>UltraSPARC</th>
<th>MIPS R10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of instructions</td>
<td>366</td>
<td>303</td>
<td>271</td>
<td>59</td>
<td>94</td>
<td>225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction size (bytes)</td>
<td>2-6</td>
<td>2-57</td>
<td>1-11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing modes</td>
<td>4</td>
<td>22</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of general-purpose registers</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>40-530</td>
<td></td>
<td></td>
<td>40-530</td>
<td></td>
</tr>
<tr>
<td>Control memory size (kBytes)</td>
<td>430</td>
<td>480</td>
<td>246</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache size (Kilobytes)</td>
<td>64</td>
<td>64</td>
<td>8</td>
<td>32</td>
<td>128</td>
<td>16-32</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

#### RISC vs. CISC (dark) vs. CISC (white background)

#### Table: RISC vs. CISC

<table>
<thead>
<tr>
<th>Processor</th>
<th>Number of instructions</th>
<th>Minimum memory size (bits)</th>
<th>Maximum memory size (bits)</th>
<th>iPad addressing</th>
<th>Load/store addressing</th>
<th>Maximum number of memory operands</th>
<th>Unaligned addressing allowed</th>
<th>MMU/cache</th>
<th>Number of bits for register specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD 6000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Intel 486</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>IBM 3800</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>Intel 80386</td>
<td>12</td>
<td>22</td>
<td>18</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>Sparc 10</td>
<td>31</td>
<td>32</td>
<td>32</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>Sparc 20</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>VAX 86</td>
<td>56</td>
<td>56</td>
<td>56</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>Cray 2</td>
<td>43</td>
<td>51</td>
<td>51</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>4</td>
</tr>
</tbody>
</table>

### Notes
- a: RISC that does not conform to this characteristic
- b: CISC that does not conform to this characteristic

---

Computer Organization II, Spring 2012, Tiina Niklander

Lecture 8: Pentium, ARM, RISC

7.2.2012
Register Files

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 \( r0-r539 \)
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

![Overlapped Register Windows Diagram](Sta10 Fig 13.1)

Circural Buffer for Overlapped Register Window

- Too many nested calls?
  - Most recent calls in registers
  - Older 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)

![Circural Buffer for Overlapped Register Window Diagram](Sta10 Fig 13.2)
Register File vs. Cache

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

<table>
<thead>
<tr>
<th>Large Register File</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>All local scalars</td>
<td>Recently-used local scalars</td>
</tr>
<tr>
<td>Individual variables</td>
<td>Blocks of memory</td>
</tr>
<tr>
<td>Compiler-assigned global variables</td>
<td>Recently-used global variables</td>
</tr>
<tr>
<td>Save/Restore based on save stack depth</td>
<td>Save/Restore based on cache replacement algorithm</td>
</tr>
<tr>
<td>Register addressing</td>
<td>Number of bits</td>
</tr>
<tr>
<td></td>
<td>Memory addressing</td>
</tr>
</tbody>
</table>

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

See course on Models of Computation
Allocation of registers
(compiler-based register optimization) (4)

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

So, use the same physical register for A and D, and for C and E.

RISC-pipeline

I: instruction fetch
E: execute  E1: reg read, E2: Alu + reg write
D: memory op

Compiler solved RAW dependency

Two port MEM, or faster mem
(split cache enough?)
RISC-pipeline, Delayed Branch

Traditional pipeline clear pipeline

Forget dependency problem here, concentrate on jump!

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 μops/VLIW-instruction

  - Lower levels as RISC

CISC-to-RISC ‘compilation’ at every execution

Just in time (JIT) compilation

http://www.cs.clemson.edu/~mark/330/coldwell/pentium.gif
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?