CPU Examples & RISC

- x86/ARM
- Instruction analysis
- RISC vs. CISC
- Register use

X86 Processor Registers

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

Pentium: FP / MMX Registers

- FP regs used as stack
- 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

Pentium: EFLAGS Register

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

Pentium: Interrupts

Calling interrupt handler; atomic hardware functionality!

If not in privileged mode (etuoikeutettu tila)

- PUSH(SS) stack segment selector to stack
- PUSH(ESP) stack pointer to stack
- PUSH(EFLAGS) status register to stack
- EFLAGS.IOPL = 00 several interrupts (keskeytystilat)
- EFLAGS.TF = 0 disable exceptions (poikkeus)
- PUSH(CS) code segment selector to stack
- PUSH(EIP) instruction pointer to stack (akkiyksilostot)

If needed

- number (interrupt vector [number], CS)
- EIP = interrupt vector [number], EIP

Return

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

Address translation:

Segment number and offset from interrupt vector => Address of the interrupt handler

See Sta10 Table 12.3 as subroutine call

ARM (Ch 12.6 Sta10)

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

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

Comp. Org II, Autumn 2010

2
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

Exception modes

- **User mode**
- **Supervisor mode**
- **Abort mode**
- **Undefined mode**
- **Fast interrupt mode**
- **Interrupt mode**
- **System mode**

ARM Register organization

- **SP** – stack pointer
- **LR** – link register
- **PC** – program counter
- **CP** – current program status register
- **SPSR** – saved program status register

Shaded regs replaced in exception modes!

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.

Program status regs (CPSR & SPSR)

<table>
<thead>
<tr>
<th>User flags</th>
<th>System-control flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>J – Jazelle instruction in use</td>
<td>T – normal / Thumb instr.</td>
</tr>
<tr>
<td>Gear byte code mode`</td>
<td>M{4:0} – processor mode</td>
</tr>
</tbody>
</table>

Exception type

- **Result**
- **Process mode**
- **Normal entry address**
- **Description**

Example:

- Data abort
  - Abort
  - ex00000010
  - Generator writes the system in abnormal

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.

Computer Organization II

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
- Microprogrammed control, 1964
- Easier control design and impl.
- Multiple processors, 1964
- Hyperthreading CPU, 2001
- Several register sets and virtual processors on chip
- Multicore CPU, 2005
- Several full processors on chip

Comp. Org II, Autumn 2010
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
- 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>Procedure</th>
<th>Dynamic</th>
<th>Machine-Instruction</th>
<th>Memory-Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Occurrence</td>
<td>Weighted</td>
<td>Weighted</td>
</tr>
<tr>
<td>ADM006</td>
<td>30%</td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>LOAD</td>
<td>5%</td>
<td>10%</td>
<td>5%</td>
</tr>
<tr>
<td>CALL</td>
<td>15%</td>
<td>20%</td>
<td>15%</td>
</tr>
<tr>
<td>IF</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>OUT2</td>
<td>3%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>OTHER</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

The table shows the percentage of dynamic operations and operands in the context of high-level language.%

Subroutine (procedure, function) calls?

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

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Dynamic</th>
<th>Local Variable</th>
<th>Local Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Occurrence</td>
<td>Steady State</td>
<td>Steady State</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>5%</td>
<td>20%</td>
</tr>
<tr>
<td>Array/Vector</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>Static Var</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Other</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Observations from Real Programs

- Most operations 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:
    - No speedup
    - Speedup: 2x

| ExTime_{old} = ExTime_{old} + (0.9 * 1.0 + 0.1 * 0.5) |
| = 0.95 x ExTime_{old} |
| Speedup = ExTime_{old} / ExTime_{new} = 1 / 0.95 = 1.053 << 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 last
    - 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 vs. CISC

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Computer Instructions</th>
<th>RISC Instructions</th>
<th>Superscalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year developed</td>
<td>IBM 360/50 1964</td>
<td>XAA 1973</td>
<td>1989</td>
</tr>
<tr>
<td>Number of operations</td>
<td>IBM 360/70</td>
<td>XAA 4,200</td>
<td>10,000</td>
</tr>
<tr>
<td>Instruction size (bits)</td>
<td>256</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Addressing modes</td>
<td>4</td>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>Number of general-purpose registers</td>
<td>16</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Clock rate in MHz</td>
<td>2.5</td>
<td>2.1</td>
<td>2</td>
</tr>
<tr>
<td>Cache size in K</td>
<td>32</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Cache hit ratio</td>
<td>90%</td>
<td>72%</td>
<td>50%</td>
</tr>
</tbody>
</table>

RISC (dark) vs. CISC (white background)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Pipelined</th>
<th>Multiple instructions</th>
<th>Instruction set</th>
<th>Load/store memory references</th>
<th>Instruction execution time</th>
<th>Superscalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC</td>
<td>Yes</td>
<td>No</td>
<td>32</td>
<td>0</td>
<td>0.01s</td>
<td>No</td>
</tr>
<tr>
<td>CISC</td>
<td>No</td>
<td>Yes</td>
<td>128</td>
<td>0</td>
<td>0.05s</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 there are only register r0-r31
  - CPU maps them to actual (true) registers r0-r539

Register Files

- Current Window Pointer
- Instruction
- Decoder
- Registers
- Data

Lecture 8: Pentium, ARM, RISC

24.11.2010

Comp. Org II, Autumn 2010
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

Sta10 Fig 13.1

Circular 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

Sta10 Fig 13.2

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>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 procedure nesting depth</td>
<td>Save/Restore based on cache replacement algorithm</td>
</tr>
<tr>
<td>Register addressing</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

Sta10 Table 13.5

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

Sta10 Fig 13.4

Allocation of registers (compiler-based register optimization)

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

Sta10 Fig 13.4a

RISC-pipeline

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

Sta10 Fig 13.6
RISC-pipeline, Delayed Branch

- Forget dependency problem here, concentrate on jump!

Traditional pipeline:

Clearing 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)

Bubble?

Bubble?

Forget dependency problem here, concentrate on jump!

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 ops/VLIW-instruction
    - Lower levels as RISC

Just in time (JIT) compilation

'compilation' at every execution

'compilation' just once per group

http://www.cs.clemson.edu/~mark/330/colwell/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?