Lecture 6: Instruction Sets

27.1.2012

Comp. Org II, Spring 2012

Instruction Sets (Käskykannat)

Ch 10-11 [Sta10]
• Operations
• Operands
• Operand references (osoitustavat)
• Pentium / ARM

Computer Instructions (konekäskyt)

• Instruction set (käskykanta) – Set of instructions CPU 'knows'
• Operation code (käskykoodi) – What does the instruction do?
• Data references (viitteet) – one, two, several?
  - Registers, memory, disk, I/O
• Where does the data come for the instruction?
• What instruction is executed next?
  - Implicit? Explicit?
• I/O?
  - Memory-mapped I/O – I/O with memory reference operations

Instructions and data (käskyt ja data)

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>0101</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>0102</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>0103</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>0104</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>ADD</td>
</tr>
<tr>
<td>0102</td>
<td>ADD</td>
</tr>
<tr>
<td>0104</td>
<td>ADD</td>
</tr>
<tr>
<td>0106</td>
<td>ADD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>DATA</td>
</tr>
<tr>
<td>0101</td>
<td>DATA</td>
</tr>
<tr>
<td>0102</td>
<td>DATA</td>
</tr>
<tr>
<td>0103</td>
<td>DATA</td>
</tr>
<tr>
<td>0104</td>
<td>DATA</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>I</td>
</tr>
<tr>
<td>0101</td>
<td>J</td>
</tr>
<tr>
<td>0102</td>
<td>K</td>
</tr>
<tr>
<td>0103</td>
<td>L</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Instruction cycle

• CPU executes instructions “one after another”
• Execution of one instruction has several phases (see state diagram). The CPU repeats these phases

Instruction types?

• Transfer between memory and registers
  - LOAD, STORE, MOVE, PUSH, POP, ...
• Controlling I/O
  - Memory-mapped I/O – like memory
  - I/O not memory-mapped – own instructions to control
• Arithmetic and logical operations
  - ADD, MUL, CLR, SET, CMP, AND, SHR, NOP, ...
• Conversions (esitystapamuunnokset)
  - TRANS, CONV, 16bTo32b, IntToFloat, ...
• Transfer of control (käskyjen suoritusjärjestyksen ohjaus), conditional, unconditional
  - JUMP, BRANCH, JEQU, CALL, EXIT, HALT, ...
• Service requests (palvelupyyntö)
  - SVC, INT, SYSENTER, SYSEXIT, ...
• Privileged instructions (etuoikeutetut käskyt)
  - DIS, IEN, flush cache, invalidate TLB, ...

What happens during instruction execution?

<table>
<thead>
<tr>
<th>Data Transfer</th>
<th>Transfer data from one location to another</th>
</tr>
</thead>
<tbody>
<tr>
<td>If memory is involved</td>
<td></td>
</tr>
<tr>
<td>Determine memory address</td>
<td></td>
</tr>
<tr>
<td>Perform virtual-to-actual memory address transformation</td>
<td></td>
</tr>
<tr>
<td>Check cache</td>
<td></td>
</tr>
<tr>
<td>Update memory map/write</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Perform function in ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set condition codes and flags</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical</th>
<th>Set condition codes and flags</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Perform arithmetic and logical operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>May involve special logic to perform conversion</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transfer of Control</th>
<th>Update program counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>For subroutine call/return, manage parameter passing and linkage</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I/O</th>
<th>Issue command to I/O module</th>
</tr>
</thead>
<tbody>
<tr>
<td>If memory-mapped I/O, determine memory-mapped address</td>
<td></td>
</tr>
</tbody>
</table>
Lecture 6: Instruction Sets

What kind of data?
- Integers, floating-points
- Boolean (tutuusarvoja)
- Characters, strings
- IRA (aka ASCII), EBCDIC
- Vectors, tables
- N elements in sequence
- Memory references

Different sizes
- 8/16/32/64b, ...
- Each type and size has its own operation code

Instruction representation (käskyformaatti)
- How many bits for each field in the instruction?
- How many different instructions?
- Maximum number of operands per instruction?
- Operands in registers or in memory?
- How many registers?
- Fixed or variable size (vakio vai vaihteleva koko)?

How many registers?
- Minimum 16 to 32
- Work data in registers
- Why not 2000 or 64000?
- Longer instructions, larger programs, slower registers
- Different register (sets) for different purpose?
- Integers vs. floating points, indices vs. data, code vs. data
- All sets can start register numbering from 0
- Opcode determines which set is used
- More registers than can be referenced?
- CPU allocates them internally
  - Register window – virtual register names
  - Example: function parameters passed in registers
  - Programmer thinks that registers are always r8-r15,
    - CPU maps r8-r15 somewhere to r8-r132 (more of this later)

Architectures
- Accumulator-based architecture (akkukone)
  - Just one register, accumulator, implicit reference to it
- Stack-based (pinokone)
  - Operands in stack, implicit reference
  - PUSH, POP
- Register-based (yleisrekisterikone)
  - All registers of the same size
  - Instructions have 2 or 3 operands
  - Load/Store architecture
  - Only LOAD/STORE have memory refs
  - ALU-operations have 3 regs

Byte ordering (tavujärjestys)
Big vs. Little Endian
- How to store a multibyte scalar value?

Big-Endian: Most significant byte in lowest byte addr
- LoadC R1, +0x1123344
- Store R1,0x1200

Little-Endian: Least significant byte in lowest byte addr
- LoadC R1, +0x1200
- Store R1,0x1200

Big vs. Little Endian
ARM
- ALU uses only one of them
  - Little-endian: x86, Pentium, VAX
  - Big-endian: IBM 370/390, Motorola 680x0 (Mac),
    most RISC-architectures
  - ARM, a bi-endian machine, accepts both
    - System control register has 1 bit (E-bit) to indicate the endian mode
    - Program controls which to use
- Byte order must be known, when transferring data from one machine to another
  - Internet uses big-endian format
  - Socket library (pastokoko) uses routines htonl() and ntohs()
    (Host to Internet & Internet to Host)
Data alignment  

- 16b data starts with even (parillinen) (byte)address  
- 32b data starts with address divisible (jaollinen) by 4  
- 64b data starts with address divisible by 8  
- Aligned data is easier to access  
- 32b data can be loaded by one operation accessing the word address (sanassote)

Unaligned data would contain no ‘wasted’ bytes, but  
- For example, loading 32b unaligned data requires two loads from memory (word address) and combining it:

\[
\begin{align*}
\text{load } r1, 2(r4) & \\
\text{shl } r1, =16 & \\
\text{load } r2, 4(r4) & \\
\text{shr } r2, =16 & \\
\text{or } r1, r2 & \\
\end{align*}
\]

Unaligned data would contain no ‘wasted’ bytes, but  
- For example, loading 32b unaligned data requires two loads from memory (word address) and combining it:

\[
\begin{align*}
\text{load } r1, 0(r4) & \\
\text{or} & \\
\end{align*}
\]

Memory references  

(Muistin osoitustavat)  
Ch II [Sta10]

Where are the operands?  

- In the memory  
  - Variable of the program, stack (pino), heap (keko)
- In the registers  
  - During the instruction execution, for speed
- Directly in the instruction  
  - Small constant values
- How does CPU know the specific location?  
  - Bits in the operation code
  - Several alternative addressing modes allowed

Addressing modes  

<table>
<thead>
<tr>
<th>Mode</th>
<th>Algorithm</th>
<th>Principal Advantage</th>
<th>Principal Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>O = A</td>
<td>No memory reference</td>
<td>Limited-width immediates</td>
</tr>
<tr>
<td>Direct</td>
<td>EA = A</td>
<td>Simple</td>
<td>Limited-address space</td>
</tr>
<tr>
<td>Indirect</td>
<td>EA = (A)</td>
<td>Large address space</td>
<td>Multiple memory reference</td>
</tr>
<tr>
<td>Register</td>
<td>EA = R</td>
<td>No memory reference</td>
<td>Limited address space</td>
</tr>
<tr>
<td>Register Relative</td>
<td>EA = (R)</td>
<td>Large address space</td>
<td>Extra memory reference</td>
</tr>
<tr>
<td>Displacement</td>
<td>EA = A + (R)</td>
<td>Flexibility</td>
<td>Complexity</td>
</tr>
<tr>
<td>Stack</td>
<td>EA = top of stack</td>
<td>No memory reference</td>
<td>Limited applicability</td>
</tr>
</tbody>
</table>

Effective address = \((R1) + A\)

Displacement Address  

(tehollinen muistiosoite)

- Effective address = \((R1) + A\)
  
  register content + constant in the instruction

- Constant relatively small (8 b, 16 b?)
- Usage  
  - Relational to PC  
  - Relational to Base  
  - Indexing a table  
  - Ref to record field  
  - Stack content  
  
  \(\text{e.g., } \text{in activation record} \)

\[\text{load } r1, 2(r4) \quad \text{or} \quad \text{load } r2, 4(r4) \quad \text{shl } r1, =16 \quad \text{load } r2, 4(r4) \quad \text{shr } r2, =16 \quad \text{or } r1, r2\]
More Addressing Modes

- Autoincrement (before/after)
  - Example: `CurrIndex = ++i;`

- Autodecrement (before/after)
  - Example: `CurrIndex = --i;`

- Autoincrement deferred
  - Example: `Sum = Sum + (*ptrX++);`

- Autoscale
  - Example: `Double X;...X=Tbl[i]; if scale 8`

Lecture 6: Instruction Sets 27.1.2012

Comp. Org II, Spring 2012 4

Pentium: Registers

- General registers (extended registers), 32-b
  - EAX, EBX, ECX, EDX: accu, base, count, data
  - EDI, ESI, ESP, EBP: stack pointer, base pointer

- Part of them can be used as 16-bit registers
  - AH, AL, BH, BL, CH, CL

- Segment registers 16b
  - CS, SS, DS, ES

- Program counter (`käskynosoitin`)
  - EIP: Extended Instruction Pointer

- Status register
  - EFLAGS: overflow, sign, zero, parity, carry...

Pentium: Operations

- Data transfers, arithmetic, moves, jumps, stores, etc.

- SSE: Single/Double/Extended precision

- MMX: SIMD operations

- No under/overflow.
  - Use closest representation

Pentium: MMX Operations

- SIMD: Single Instruction Multiple Data

- SIMD instructions with packed types

- Use closest representation
Pentium: Addressing modes (muistin osoitustavat)

<table>
<thead>
<tr>
<th>x86 Addressing Mode</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Operand = A</td>
</tr>
<tr>
<td>Register/Operand</td>
<td>(R) - 1, 2, 4, 8B</td>
</tr>
<tr>
<td>Displacement</td>
<td>LA = (BR) + A</td>
</tr>
<tr>
<td>Base</td>
<td>LA = (BR)</td>
</tr>
<tr>
<td>Base with Displacement</td>
<td>LA = (BR) + D</td>
</tr>
<tr>
<td>Scaled Index with Displacement</td>
<td>LA = (BR) + D x S</td>
</tr>
<tr>
<td>Base with Index and Displacement</td>
<td>LA = (BR) + (D x S) + A</td>
</tr>
</tbody>
</table>

Labels:
- LA = linear address
- CR = contents of X
- IS = segment index
- PC = program counter
- A = contents of an address field in the instruction

Pentium: Addressing Mode Calculation

LA = (SR) + (I) x S + B + A

Discussion?

Pentium: Instruction format

- CISC
  - Complex Instruction Set Computer
- Lots of alternative fields
  - Part may be present or absent in the bit sequence
  - Prefix 0-4 bytes
  - Interpretation of the rest of the bit sequence depends on the content of the preceding fields
- Plenty of alternative addressing modes (osoitustapa)
  - At most one operand can be in the memory
  - 24 different
- Backward compatibility
  - OLD 16-bit 8086-programs must still work
  - How to handle old instructions: emulate, simulate?

Pentium: Instruction format

- Instruction prefix (optional)
  - LOCK – exclusive use of shared memory in multiprocessor env.
  - REP – repeat operation to all characters of a string
- Segment override (optional)
  - Use the segment register explicitly specified in the instruction.
  - Else use the default segment register (implicit assumption).
- Operand size override (optional)
  - Switch between 16 or 32 bit operand, override default size.
- Address size override (optional)
  - Switch between 16 or 32 bit addressing, Override the default, which could be either.
Pentium: Instruction format

- Displacement (optional)
  - Certain addressing modes need this
  - 0, 1, 2 or 4 bytes (0, 8, 16 or 32 bits)
- Immediate (optional)
  - Certain addressing modes need this, value for operand
  - 0, 1, 2 or 4 bytes

ARM: Instruction set (käskykanta)

- RISC
  - Reduced Instruction Set Computer
- Fixed instruction length (32b), regular format
  - All instructions have the condition code (4 bits)
- Small number of different instructions
  - Instruction type (3 bit) and additional opcode/modifier (5 bit)
  - Easier hardware implementation, faster execution
  - Longer programs?
- Load/Store-architecture
  - 16 visible general registers (4 bits in the instruction)
  - PC = R15
- Thump instruction subset uses 16 bit instructions

ARM Data Types

- 8 (byte), 16 (halfword), 32 (word) bits - word aligned
- Unsigned integer and two's-complement signed integer
- Majority of implementations do not provide floating-point hardware
- Little and Big Endian supported
  - Bit E in status register defines which is used

ARM Addressing modes

- Load/Store
- Indirect
  - base reg + offset
- Indexing alternatives
  - Offset
    - Address is base + offset
  - Preindex
    - Form address
      - Write address to base
  - Postindex
    - Use base as address
      - Calculate new address
to base

ARM Addressing mode

- Data Processing instructions
  - Register addressing
    - Value in register operands may be scaled using a shift operator
  - Or mixture of register and immediate addressing
- Branch instructions
  - Immediate
  - Instruction contains 24 bit value
  - Shifted 2 bits left
    - On word boundary
    - Effective range +/-32MB from PC.
ARM Load/Store Multiple Addressing
- Load/store subset of general-purpose registers
- 16-bit instruction field specifies list of registers
- Sequential range of memory addresses
- Base register specifies main memory address

ARM Instruction Formats
- \( S = \) For data processing instructions, updates condition codes
- \( S = \) For load/store multiple instructions, execution restricted to supervisor mode
- \( P, U, W = \) distinguish between different types of addressing mode
- \( B = \) Unsigned byte (\( B==1 \)) or word (\( B==0 \)) access
- \( L = \) For load/store instructions, Load (\( L==1 \)) or Store (\( L==0 \))
- \( L = \) For branch instructions, is return address stored in link register

ARM Condition codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Symbol</th>
<th>Condition Tested</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>( x = y )</td>
<td>Equal</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>( x \neq y )</td>
<td>Not equal</td>
</tr>
<tr>
<td>0010</td>
<td>SL</td>
<td>( x \leq y )</td>
<td>Signed less or equal</td>
</tr>
<tr>
<td>0011</td>
<td>CH</td>
<td>( x &gt; y )</td>
<td>Greater than</td>
</tr>
<tr>
<td>0100</td>
<td>GT</td>
<td>( x = y )</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>0101</td>
<td>LH</td>
<td>( x &lt; y )</td>
<td>Less than</td>
</tr>
<tr>
<td>0110</td>
<td>HE</td>
<td>( x \geq y )</td>
<td>Signed greater or equal</td>
</tr>
<tr>
<td>0111</td>
<td>LT</td>
<td>( x &lt; y )</td>
<td>Less than or equal</td>
</tr>
</tbody>
</table>

ARM vs. CISC
- RISC: Easy to execute
- CISC: Support high-level languages
- High-level programming language

Summary
- Instruction set types: Stack, register, load-store
- Data types: int, float, char
- Addressing modes: indexed, others?
- Operation types?
  - Arithmetic & logical, shifts, conversions, vector
  - Comparisons
  - Control
    - If-then-else, loops, function calls/returns
    - Conditional instructions
  - Loads/stores, stack ops, vector ops
  - Privileged, os instructions
- Instruction formats
- Intel and ARM case studies

Review Questions / Kertauskysymyksiä
- Fields of the instruction?
- How does CPU know if the integer is 16 b or 32 b?
- Meaning of Big-Endian?
- Benefits of fixed instruction size vs. variable size instruction format?