Instruction sets (Käskykannat)

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

- CPU executes instructions “one after another”
- Execution of one instruction has several phases (see state diagram). The CPU repeats these phases.
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?
  - Where does the data come for the instruction?
    - Registers, memory, disk, I/O
  - Where is the result stored?
    - Registers, memory, disk, I/O

- What instruction is executed next?
  - Implicit? Explicit?

- I/O?
  - Memory-mapped I/O → I/O with memory reference operations

Covered on Comp. Org I

Access time? Access rate?
Instructions and data (käskyt ja data)

(a) Binary program

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0010 0010 0000 0001</td>
</tr>
<tr>
<td>102</td>
<td>0001 0010 0000 0010</td>
</tr>
<tr>
<td>103</td>
<td>0001 0010 0000 0111</td>
</tr>
<tr>
<td>104</td>
<td>0011 0010 0000 0100</td>
</tr>
</tbody>
</table>

(b) Hexadecimal program

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>2201</td>
</tr>
<tr>
<td>102</td>
<td>1202</td>
</tr>
<tr>
<td>103</td>
<td>1203</td>
</tr>
<tr>
<td>104</td>
<td>3204</td>
</tr>
<tr>
<td>201</td>
<td>0002</td>
</tr>
<tr>
<td>202</td>
<td>0003</td>
</tr>
<tr>
<td>203</td>
<td>0004</td>
</tr>
<tr>
<td>204</td>
<td>0000</td>
</tr>
</tbody>
</table>

(c) Symbolic program

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Symbolic name</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>LDA</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>STA</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>DAT</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>DAT</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>DAT</td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>DAT</td>
<td></td>
</tr>
</tbody>
</table>

(d) Assembly program

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMUL</td>
<td>LDA</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>J</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>K</td>
</tr>
<tr>
<td></td>
<td>STA</td>
<td>N</td>
</tr>
</tbody>
</table>

I   DATA   2
J   DATA   3
K   DATA   4
N   DATA   0

(Sta10 Fig 11.13)
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, COMP, 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, IRET, 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></td>
<td>If memory is involved:</td>
</tr>
<tr>
<td></td>
<td>Determine memory address</td>
</tr>
<tr>
<td></td>
<td>Perform virtual-to-actual-memory address transformation</td>
</tr>
<tr>
<td></td>
<td>Check cache</td>
</tr>
<tr>
<td></td>
<td>Initiate memory read/write</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>May involve data transfer, before and/or after</td>
</tr>
<tr>
<td></td>
<td>Perform function in ALU</td>
</tr>
<tr>
<td></td>
<td>Set condition codes and flags</td>
</tr>
<tr>
<td>Logical</td>
<td>Same as arithmetic</td>
</tr>
<tr>
<td>Conversion</td>
<td>Similar to arithmetic and logical. May involve special logic to perform conversion</td>
</tr>
<tr>
<td>Transfer of Control</td>
<td>Update program counter. For subroutine call/return, manage parameter passing and linkage</td>
</tr>
<tr>
<td>I/O</td>
<td>Issue command to I/O module</td>
</tr>
<tr>
<td></td>
<td>If memory-mapped I/O, determine memory-mapped address</td>
</tr>
</tbody>
</table>

(Sta10 Table 10.4)
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

<table>
<thead>
<tr>
<th>Operation Mnemonic</th>
<th>Name</th>
<th>Number of Bits Transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Load</td>
<td>32</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
<td>16</td>
</tr>
<tr>
<td>LR</td>
<td>Load</td>
<td>32</td>
</tr>
<tr>
<td>LER</td>
<td>Load (Short)</td>
<td>32</td>
</tr>
<tr>
<td>LE</td>
<td>Load (Short)</td>
<td>32</td>
</tr>
<tr>
<td>LDR</td>
<td>Load (Long)</td>
<td>64</td>
</tr>
<tr>
<td>LD</td>
<td>Load (Long)</td>
<td>64</td>
</tr>
<tr>
<td>ST</td>
<td>Store</td>
<td>32</td>
</tr>
<tr>
<td>STH</td>
<td>Store Halfword</td>
<td>16</td>
</tr>
<tr>
<td>STC</td>
<td>Store Character</td>
<td>8</td>
</tr>
<tr>
<td>STE</td>
<td>Store (Short)</td>
<td>32</td>
</tr>
<tr>
<td>STD</td>
<td>Store (Long)</td>
<td>64</td>
</tr>
</tbody>
</table>

IBM EAS/390 (Sta10 Table 10.5)
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)?

<table>
<thead>
<tr>
<th>Number of Addresses</th>
<th>Symbolic Representation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>OP A, B, C</td>
<td>A ← B OP C</td>
</tr>
<tr>
<td>2</td>
<td>OP A, B</td>
<td>A ← A OP B</td>
</tr>
<tr>
<td>1</td>
<td>OP A</td>
<td>AC ← AC OP A</td>
</tr>
<tr>
<td>0</td>
<td>OP</td>
<td>T ← (T - 1) OP T</td>
</tr>
</tbody>
</table>

AC = accumulator  
A, B, C = memory or register locations  
T = top of stack  
(T - 1) = second element of stack

(Sta10 Table 10.1)
How many registers?

- Minimum 16 to 32
  - Work data in 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
    - (We’ll come back to 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

---

See: Appendix 10A in Ch10 [Sta10]

Example: JVM

LOAD   R3, C
LOAD   R2, B
ADD    R1, R2, R3
STORE  R1, A
Byte ordering (tavujärjestys): Big vs. Little Endian

How to store a multibyte scalar value?

0x1200: 0x1200 0x1201 0x1202 0x1203

(sanaosoite) Word

"Isoimmassa lopputavu"

Big-Endian:
Most significant byte in lowest byte addr

Little-Endian:
Least significant byte in lowest byte addr

"Pienimmässä lopputavu"

0x00000044 = 0x44 0x00 0x00 0x00

STORE 0x11223344, 0x1200

Big-Endian:
Most significant byte in lowest byte addr

Little-Endian:
Least significant byte in lowest byte addr

See: Appendix 10B (Sta10)
Big vs. Little Endian

- 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 (*pistokekirjasto*) has routines `htoi()` and `itoh()`
    (Host to Internet & Internet to Host)
Data alignment (kohdentaminen)

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

\[
\begin{align*}
\text{load } r1, 0(r4) & \quad \text{or} \quad \text{load } r1, 2(r4) \\
\text{shl } r1, =16 & \\
\text{load } r2, 4(r4) & \\
\text{shr } r2, =16 & \\
\text{or } r1, r2 & \\
\end{align*}
\]
Memory references

(Muistin osoitustavat)

Ch 11 [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 (*osoitusmuodot*)

- **(a) Immediate**
  - Instruction
    - Operand
  - Memory
  - Operand

- **(b) Direct**
  - Instruction
    - A
  - Memory
  - Operand
  - Memory
  - Registers
  - Operand

- **(c) Indirect**
  - Instruction
    - A
  - Memory
  - Operand
  - Memory
  - Operand
  - Registers

- **(d) Register**
  - Instruction
    - R
  - Memory
  - Registers
  - Operand

- **(e) Register Indirect**
  - Instruction
    - R
  - Memory
  - Registers
  - Operand

- **(f) Displacement**
  - Instruction
    - R
  - Memory
  - Registers
  - Operand

- **(g) Stack**
  - Instruction
    - Implicit
  - Top of Stack
  - Register

*(Sta10 Fig 11.1)*
### 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>Operand = A</td>
<td>No memory reference</td>
<td>Limited operand magnitude</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 references</td>
</tr>
<tr>
<td>Register</td>
<td>EA = R</td>
<td>No memory reference</td>
<td>Limited address space</td>
</tr>
<tr>
<td>Register indirect</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>

- **EA** = Effective Address
- **(A)** = content of memory location A
- **(R)** = content of register R
- One register for the top-most stack item’s address
- Register (or two) for the top stack item (or two)

*(Sta10 Table 11.1)*
Displacement Address (*siirtymä*)

- 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
    
    *(e.g., in activation record)*

(Rahollinen muistiosoite)
More addressing modes

- **Autoincrement (before/after)**
  - Example `CurrIndex=i++;`
  - EA = (R), \( R \leftarrow (R) + S \)

- **Autodecrement (before/after)**
  - Example `CurrIndex=--i;`
  - \( R \leftarrow (R) - S \), EA = (R)

- **Autoincrement deferred**
  - Example `Sum = Sum + (*ptrX++);`
  - EA = Mem(R), \( R \leftarrow (R) + S \)

- **Autoscale**
  - Example `Double X;`
  - EA = \( A + (R) \times S \)
  - \( X = \text{Tbl}[i]; \)
Pentium
Pentium: Registers

- General registers (yleisrekisterit), 32-b
  - EAX, EBX, ECX, EDX  accu, base, count, data
  - ESI, EDI  source & destination index
  - ESP, EBP  stack pointer, base pointer
- Part of them can be used as 16-bit registers
  - AX, BX, CX, DX, SI, DI, SP, BP
- Or even as 8-bit registers
  - AH, AL, BH, BL, CH, CL, DH, DL
- Segment registers 16b
  - CS, SS, DS, ES, FS, GS
    - code, stack, data, data, ...
- Program counter (käskynosoitin)
  - EIP Extended Instruction Pointer
- Status register
  - EFLAGS
    - overflow, sign, zero, parity, carry,...
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.</td>
</tr>
<tr>
<td>Integer</td>
<td>A signed binary value contained in a byte, word, or doubleword, using two's complement representation.</td>
</tr>
<tr>
<td>Ordinal</td>
<td>An unsigned integer contained in a byte, word, or doubleword.</td>
</tr>
<tr>
<td>Unpacked binary coded decimal (BCD)</td>
<td>A representation of a BCD digit in the range 0 through 9, with one digit in each byte.</td>
</tr>
<tr>
<td>Packed BCD</td>
<td>Packed byte representation of two BCD digits; value in the range 0 to 99.</td>
</tr>
<tr>
<td>Near pointer</td>
<td>A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.</td>
</tr>
<tr>
<td>Far pointer</td>
<td>A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.</td>
</tr>
<tr>
<td>Bit field</td>
<td>A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.</td>
</tr>
<tr>
<td>Bit string</td>
<td>A contiguous sequence of bits, containing from zero to $2^{32} - 1$ bits.</td>
</tr>
<tr>
<td>Byte string</td>
<td>A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{32} - 1$ bytes.</td>
</tr>
<tr>
<td>Floating point</td>
<td>Single / Double / Extended precision</td>
</tr>
<tr>
<td>Packed SIMD (single instruction, multiple data)</td>
<td>Packed 64-bit and 128-bit data types</td>
</tr>
</tbody>
</table>

IEEE 754 standard

(Sta10Table 10.2)
### Pentium: Operations

(just part of)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTER</td>
<td>Creates a stack frame that can be used to implement the rules of a block-structured high-level language.</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Reverses the action of the previous ENTER.</td>
</tr>
<tr>
<td>BOUND</td>
<td>Check array bounds. Verifies that the value in operand 1 is within lower and upper boundaries.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS</td>
<td>Load pointer into D segment register.</td>
</tr>
<tr>
<td>HLT</td>
<td>Halt.</td>
</tr>
<tr>
<td>LOCK</td>
<td>Asserts a hold on shared memory so that the Pentium has exclusive use of it during the instruction that immediately follows the LOCK.</td>
</tr>
<tr>
<td>ESC</td>
<td>Processor extension escape. An escape code that indicates the succeeding instructions are to be executed by a numeric coprocessor that supports high-precision integer and floating-point calculations.</td>
</tr>
<tr>
<td>WAIT</td>
<td>Wait until BUSY# negated. Suspends Pentium program execution until the processor detects that the BUSY pin is inactive, indicating that the numeric coprocessor has finished execution.</td>
</tr>
<tr>
<td>SGDT</td>
<td>Store global descriptor table.</td>
</tr>
<tr>
<td>LSL</td>
<td>Load segment limit. Loads a user-specified register with a segment limit.</td>
</tr>
<tr>
<td>VERR/VERW</td>
<td>Verify segment for reading/writing.</td>
</tr>
<tr>
<td>INVD</td>
<td>Flushes the internal cache memory.</td>
</tr>
<tr>
<td>WBINVD</td>
<td>Flushes the internal cache memory after writing dirty lines to memory.</td>
</tr>
<tr>
<td>INVLPG</td>
<td>Invalidates a translation lookaside buffer (TLB) entry.</td>
</tr>
</tbody>
</table>

### Segment Register

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS</td>
<td>Load pointer into D segment register.</td>
</tr>
<tr>
<td>HLT</td>
<td>Halt.</td>
</tr>
<tr>
<td>LOCK</td>
<td>Asserts a hold on shared memory so that the Pentium has exclusive use of it during the instruction that immediately follows the LOCK.</td>
</tr>
<tr>
<td>ESC</td>
<td>Processor extension escape. An escape code that indicates the succeeding instructions are to be executed by a numeric coprocessor that supports high-precision integer and floating-point calculations.</td>
</tr>
<tr>
<td>WAIT</td>
<td>Wait until BUSY# negated. Suspends Pentium program execution until the processor detects that the BUSY pin is inactive, indicating that the numeric coprocessor has finished execution.</td>
</tr>
</tbody>
</table>

### Protection

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGDT</td>
<td>Store global descriptor table.</td>
</tr>
<tr>
<td>LSL</td>
<td>Load segment limit. Loads a user-specified register with a segment limit.</td>
</tr>
<tr>
<td>VERR/VERW</td>
<td>Verify segment for reading/writing.</td>
</tr>
</tbody>
</table>

### Cache Management

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVD</td>
<td>Flushes the internal cache memory.</td>
</tr>
<tr>
<td>WBINVD</td>
<td>Flushes the internal cache memory after writing dirty lines to memory.</td>
</tr>
<tr>
<td>INVLPG</td>
<td>Invalidates a translation lookaside buffer (TLB) entry.</td>
</tr>
</tbody>
</table>

*(Sta10 Table 10.8)*
## Pentium: MMX Operations

(just part of)

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>PADD [B, W, D]</td>
<td>Parallel add of packed eight bytes, four 16-bit words, or two 32-bit doublewords, with wraparound.</td>
</tr>
<tr>
<td></td>
<td>PMULHW</td>
<td>Parallel multiply of four signed 16-bit words, with high-order 16 bits of 32-bit result chosen.</td>
</tr>
<tr>
<td></td>
<td>PMULLW</td>
<td>Parallel multiply of four signed 16-bit words, with low-order 16 bits of 32-bit result chosen.</td>
</tr>
<tr>
<td></td>
<td>PMADDWD</td>
<td>Parallel multiply of four signed 16-bit words; add together adjacent pairs of 32-bit results.</td>
</tr>
</tbody>
</table>

| Conversion | PACKUSWB | Pack words into bytes with unsigned saturation. |
|           | PACKSS [WB, DW] | Pack words into bytes, or doublewords into words, with signed saturation. |
|           | PUNPCKH [BW, WD, DQ] | Parallel unpack (interleaved merge) high-order bytes, words, or doublewords from MMX register. |
|           | PUNPCKL [BW, WD, DQ] | Parallel unpack (interleaved merge) low-order bytes, words, or doublewords from MMX register. |

(Sta10 Table 10.11)
## Pentium: Addressing modes
(muistin osoitustavat)

### x86 Addressing

<table>
<thead>
<tr>
<th>Mode</th>
<th>Algorithm</th>
<th>Addressing modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Operand = A</td>
<td>1, 2, 4, 8B</td>
</tr>
<tr>
<td>Register Operand</td>
<td><strong>Operand = (R)</strong></td>
<td>Registers: 1, 2, 4, 8B</td>
</tr>
<tr>
<td>Displacement</td>
<td>LA = (SR) + A</td>
<td></td>
</tr>
<tr>
<td>Base</td>
<td>LA = (SR) + (B)</td>
<td></td>
</tr>
<tr>
<td>Base with Displacement</td>
<td>LA = (SR) + (B) + A</td>
<td></td>
</tr>
<tr>
<td>Scaled Index with Displacement</td>
<td>LA = (SR) + (I) \times S + A</td>
<td></td>
</tr>
<tr>
<td>Base with Index and Displacement</td>
<td>LA = (SR) + (B) + (I) + A</td>
<td></td>
</tr>
<tr>
<td>Base with Scaled Index and Displacement</td>
<td><strong>LA = (SR) + (I) \times S + (B) + A</strong></td>
<td></td>
</tr>
<tr>
<td>Relative</td>
<td>LA = (PC) + A</td>
<td></td>
</tr>
</tbody>
</table>

*LA = linear address  R = register  
(X) = contents of X  B = base register  
SR = segment register I = index register  
PC = program counter  S = scaling factor  
A = contents of an address field in the instruction*

*(Sta10 Table 11.2)*

indexing arrays?  
arrays in stack?  
two dimensional arrays?
Pentium: Addressing Mode Calculation

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

Diagram showing the addressing mode calculation with labels A, B, and C.
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

1. Operand (register) or form part of the addressing-mode
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

- **Opcode**
  - Each instruction has its own bit sequence (incl. opcode)
  - Bits specify the size of the operand (8/16/32b)

- **ModR/m (optional)**
  - Indicate, whether operand is in a register or in memory
  - What addressing mode (*osoitusmuoto*) to be used
  - Sometimes enhance the opcode information (with 3 bits)

- **SIB = Scale/Index/Base (optional)**
  - Some addressing modes need extra information
  - Scale: scale factor for indexing (element size)
  - Index: index register (number)
  - Base: base register (number)
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 Instructions
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)

- Fixed data size

- **Thump** instruction set uses 16 bit instructions
ARM Data Types

- 8 (byte), 16 (halfword), 32 (word) bits - word aligned
- Unsigned integer and twos-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

LDMxx r10, {r0, r1, r4}
STMxx r10, {r0, r1, r4}
### ARM Instruction Formats

<table>
<thead>
<tr>
<th>Condition (cond)</th>
<th>Immediate (S)</th>
<th>Opcode</th>
<th>Register (Rn/Rd)</th>
<th>Shift Amount</th>
<th>Shift</th>
<th>Register (Rm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Processing Immediate Shift (cond 0 0 0)</td>
<td>0</td>
<td>0 0</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Shift Amount</td>
</tr>
<tr>
<td>Data Processing Register Shift (cond 0 0 0)</td>
<td>0</td>
<td>0 0</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rs</td>
</tr>
<tr>
<td>Data Processing Immediate (cond 0 0 1)</td>
<td>0</td>
<td>0 1</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rotate</td>
</tr>
<tr>
<td>Load/Store Immediate Offset (cond 0 1 0)</td>
<td>0 1 0</td>
<td>P</td>
<td>U</td>
<td>B</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Load/Store Register Offset (cond 0 1 1)</td>
<td>0 1 1</td>
<td>P</td>
<td>U</td>
<td>B</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Load/Store Multiple (cond 1 0 0)</td>
<td>1 0 0</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>Branch/Branch with Link (cond 1 0 1)</td>
<td>1 0 1</td>
<td>L</td>
<td>24-bit Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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

---

Discussion?
### 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>Z = 1</td>
<td>Equal</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Z = 0</td>
<td>Not equal</td>
</tr>
<tr>
<td>0010</td>
<td>CS/HS</td>
<td>C = 1</td>
<td>Carry set/unsigned higher or same</td>
</tr>
<tr>
<td>0011</td>
<td>CC/LO</td>
<td>C = 0</td>
<td>Carry clear/unsigned lower</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>N = 1</td>
<td>Minus/negative</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>N = 0</td>
<td>Plus/positive or zero</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>V = 1</td>
<td>Overflow</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>V = 0</td>
<td>No overflow</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>C = 1 AND Z = 0</td>
<td>Unsigned higher</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>C = 0 OR Z = 1</td>
<td>Unsigned lower or same</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>N = V</td>
<td>Signed greater than or equal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![N = 1 AND V = 1 OR (N = 0 AND V = 0)]</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>N ≠ V</td>
<td>Signed less than</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![N = 1 AND V = 0 OR (N = 0 AND V = 1)]</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>(Z = 0) AND (N = V)</td>
<td>Signed greater than</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>(Z = 1) OR (N ≠ V)</td>
<td>Signed less than or equal</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>—</td>
<td>Always (unconditional)</td>
</tr>
<tr>
<td>1111</td>
<td>—</td>
<td>—</td>
<td>This instruction can only be executed unconditionally</td>
</tr>
</tbody>
</table>

**Condition flags:**
- **N** – Negative
- **Z** – Zero
- **C** – Carry
- **V** – oVerflow

*(Sta10 Tbl 10.12)*
RISC vs. CISC

We’ll return to this later (lecture 8)

- **RISC**
  - Easy to execute
  - HW

- **CISC**
  - Support high-level languages
  - Difficult to execute
  - HW (Pentium)
  - SW (Crusoe)

- **High-level programming language**

---

Computer Organization II, Autumn 2010, Teemu Kerola
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?