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 *(käslykanta)*:
  - 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

---

Instructions and data *(käskyt ja data)*

![Binary program](image)

![Hexadecimal program](image)

![Symbolic program](image)

![Assembly program](image)

---

Covered on *Comp. Org I*

Access time?

Access rate?

Symbolic name
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 (palvelupyynnöt)
  - 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>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>Initiate memory read/write</td>
<td></td>
</tr>
<tr>
<td>May involve data transfer, before and/or after</td>
<td></td>
</tr>
<tr>
<td>Perform function in ALU</td>
<td></td>
</tr>
<tr>
<td>Set condition codes and flags</td>
<td></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>If memory-mapped I/O, determine memory-mapped address</td>
<td></td>
</tr>
</tbody>
</table>
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</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>STR</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

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?

```plaintext
0x1200: 0x1201 0x1202 0x1203
```

```
Big-Endian: Most significant byte in lowest byte addr
```

```
Little-Endian: Least significant byte in lowest byte addr
```

```
LoadC R1, =0x11223344
Store R1,0x1200
```

```
0x11 0x22 0x33 0x44
```

```
0x00000044 = 0x44 0x00 0x00 0x00
```

---

**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

![Diagram](image)

**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*)

- Constant
- Variable, constant
- Pointer in register
- 1D-array, record
- Pointer in memory
- Register
- Stack in memory (in registers?)
### 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 ind.</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)

**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)*

- **JUMP** *+5*
- **CALL** SP, Summation(BX)
- **ADDF** F2,F2, Table(R5)
- **MUL** F4,F6, Salary(R8)
- **STORE** F2, -4(FP)
More Addressing Modes

- Autoincrement (before/after)
  - Example: `CurrIndex = i++;`
  - `EA = (R), R ← (R) + S`

- Autodecrement (before/after)
  - Example: `CurrIndex = –i;`
  - `R ← (R) - S, EA = (R)`

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

- Autoscale
  - Example: `Double X;`
  - `X = Tbl[i]; # scale 8`
  - `EA = A + (R) * S`

Operand size:
- = byte, 4 = word, …
Pentium: Registers

- General registers (extended registers), 32-b
  - EAX, EBX, ECX, EDX: accumulator, 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 (käskynosotin)
- EIP Extended Instruction Pointer
- Status register
  - EFLAGS
    - overflow, sign, zero, parity, carry....

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>32-bit general register (extended register)</td>
</tr>
<tr>
<td>EBX</td>
<td>32-bit general register (extended register)</td>
</tr>
<tr>
<td>ECX</td>
<td>32-bit general register (extended register)</td>
</tr>
<tr>
<td>EDX</td>
<td>32-bit general register (extended register)</td>
</tr>
<tr>
<td>ESP</td>
<td>32-bit stack pointer (extended register)</td>
</tr>
<tr>
<td>EBP</td>
<td>32-bit base pointer (extended register)</td>
</tr>
<tr>
<td>ESI</td>
<td>32-bit source index (extended register)</td>
</tr>
<tr>
<td>EDI</td>
<td>32-bit destination index (extended register)</td>
</tr>
</tbody>
</table>

x86: Data types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>Float</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, values 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.</td>
</tr>
<tr>
<td>Far pointer</td>
<td>A logical address consisting of a 16-bit segment selector and a 20-bit offset (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 positions 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 1 to 32 bits.</td>
</tr>
<tr>
<td>Byte string</td>
<td>A contiguous sequence of bytes, words, or doublewords, containing from 1 to 128 bits.</td>
</tr>
<tr>
<td>Floating point</td>
<td>Single / Double / Extended precision, IEEE 754 standard.</td>
</tr>
</tbody>
</table>
### Pentium: Operations

#### (just part of)

<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>HALT</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 instructions that immediately follow the LOCK.</td>
</tr>
<tr>
<td>ESC</td>
<td>Process exception 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 is cleared. Subsequent 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>

### 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>PADDQ [B, W, D]</td>
<td>Parallel add of packed eight-bytes, four 16-bit words, or two 32-bit integers, with saturation.</td>
</tr>
<tr>
<td></td>
<td>PADDS (B, W)</td>
<td>Add with saturation.</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 cleared.</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 cleared.</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>

| Convert        | PACKUSWB           | Pack words into byes with unsaturated saturation. |
|                | PACKSW (Wb, Dw)    | Pack words into byes, or doublewords into words, with unsaturated saturation. |
|                | FUNPCKH (BW, WD, DO)| Parallel unpack (unreduced merge) high-order bytes, words, or doublewords from MMX register. |
|                | FUNPCKL (BW, WD, DO)| Parallel unpack (unreduced merge) low-order bytes, words, or doublewords from MMX register. |
**Pentium: Addressing modes**

**muistin osoitustavat**

<table>
<thead>
<tr>
<th>x86 Addressing Mode</th>
<th>Algorithm</th>
<th>Registers:</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>Operand = (R)</td>
<td></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) × 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>LA = (SR) + (I) × S + (B) + A</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

---

**Pentium: Addressing Mode Calculation**

\[ LA = (SR) + (I)S + (B) + A \]
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?

Addressing

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

(Sfu10 Fig 11.9)
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.

- Opcode
  - Each instruction has its own bit sequence (incl. opcode)
  - Bits specify the size of the operand (8/16/32 b)
- ModR/m (optional)
  - Indicate, whether operand is in a register or in memory
  - What addressing mode (osoitmusuoto) 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

Computer Organization II

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)
  - 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

![Diagrams](Sta10 Fig 11.3)

**Discussion?**

---

**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

LDMax r10, {r0,r1,r4}
SDMax r10, {r0,r1,r4}

Base register 0x20C
Increment after (IA) Increment before (IB) Decrement after (DA) Decrement before (DB)
(r4) (r1) (r0) 0x214 0x218 0x210 0x20C
(r4) (r1) (r0) 0x214 0x218 0x210 0x20C
(r0) (r1) (r4) 0x208 0x204 0x204 0x200
(r0) (r1) (r4) 0x208 0x204 0x204 0x200

ARM Instruction Formats

<table>
<thead>
<tr>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>shift amount</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rs</td>
<td>shift</td>
<td>1</td>
<td>Rm</td>
</tr>
<tr>
<td>cond</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>rotate</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>B</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>immediate</td>
</tr>
<tr>
<td>cond</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>P</td>
<td>U</td>
<td>B</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>shift amount</td>
</tr>
<tr>
<td>cond</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
<td>Rd</td>
<td>register list</td>
</tr>
<tr>
<td>cond</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24-bit offset</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
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>MT</td>
<td>N = 1</td>
<td>Minus/negative</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>N = 0</td>
<td>Minus/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>That instruction can only be executed unconditionally</td>
</tr>
</tbody>
</table>

Condition flags: N, Z, C and V
N – Negative
Z – Zero
C – Carry
V – oVerflow

RISC vs. CISC

- **RISC**
  - High-level programming language
  - Easy to execute
  - HW

- **CISC**
  - High-level programming language
  - Support high-level languages
  - Difficult to execute
  - HW

- **RISC**
  - High-level programming language
  - Easy to execute
  - HW

We’ll return to this later (lecture 8)
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?