Lecture 3:

Simple Computer Architecture II

www.cl.cam.ac.uk/Teaching/2001/OSFounds/

Lecture 3: Wednesday 10th October 2001

Today's Lecture

Today we'll cover:

- What do machine instructions look like?
  - Instructions & condition codes
  - Branching,
  - Addressing.
- How do we store data in the machine?
  - Text,
  - Floating point,
  - Data structures.
- Fetch-Execute cycle:
  - "Tieing it all up"

Arithmetic & Logical Instructions

- Some common ALU instructions are:

Mnemonic		C/Java Equivalent
`and`	$d \leftarrow a, b$	`d = a & b;`
`xor`	$d \leftarrow a, b$	`d = a ^ b;`
`bis`	$d \leftarrow a, b$	`d = a | b;`
`bic`	$d \leftarrow a, b$	`d = a & (~b);`
`add`	$d \leftarrow a, b$	`d = a + b;`
`sub`	$d \leftarrow a, b$	`d = a - b;`
`rsb`	$d \leftarrow a, b$	`d = b - a;`
`shl`	$d \leftarrow a, b$	`d = a << b;`
`shr`	$d \leftarrow a, b$	`d = a >> b;`

  Both d and a *must* be registers; b can be a register or a (small) constant.

- Typically also have `addc` and `subc`, which handle carry or borrow (for multi-precision arithmetic), e.g.

```
add  d0, a0, b0    // compute "low" part.
addc d1, a1, b1    // compute "high" part.
```

- May also get:
  - Arithmetic shifts: `asr` and `asl`(?)
  - Rotates: `ror` and `rol`.

Conditional Execution

- Seen flags C, N, V; add Z (zero), logical NOR of all bits in output.
- Can predicate execution based on (some combination) of flags, e.g.

```
sub d, a, b     // compute d = a - b
beq proc1       // if equal, goto proc1
br  proc2       // otherwise goto proc2
```

  Java equivalent approximately:

```
if (a==b) proc1() else proc2();
```

- On ARM everything conditional, e.g.

```
sub   d, a, b     # compute d = a - b
moveq d, #5       # if equal, d = 5;
movne d, #7       # otherwise d = 7;
```

  Java equiv: `d = (a==b) ?  5 :  7;`

- "Silent" versions useful when don't really want result, e.g. `tst`, `teq`, `cmp`.
- Alt (MIPS): `beq reg1 reg2 L1`

Condition Codes

Suffix	Meaning	Flags
EQ, Z	Equal, zero	Z == 1
NE, NZ	Not equal, non-zero	Z == 0
MI	Negative	N == 1
PL	Positive (incl. zero)	N == 0
CS, HS	Carry, higher or same	C == 1
CC, LO	No carry, lower	C == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
HI	Higher	C == 1 && Z == 0
LS	Lower or same	C == 0 || Z == 1
GE	Greater than or equal	N == V
GT	Greater than	N == V && Z == 0
LT	Less than	N != V
LE	Less than or equal	N != V || Z == 1

- HS, LO, etc. used for unsigned comparisons (recall that $\overline{C}$ means "borrow").
- GE, LT, etc. used for signed comparisons: check both N and V so always works.

Loads & Stores

- Have variable sized values, e.g. bytes (8-bits), words (16-bits), longwords (32-bits) and quadwords (64-bits).
- Load or store instructions usually have a suffix to determine the size, e.g. 'b' for byte, 'w' for word, 'l' for longword.
- When storing > 1 byte, have two main options: big endian and little endian; e.g. storing longword 0xDEADBEEF into memory at address 0x4.

If read back a *byte* from address 0x4, get 0xDE if big-endian, or 0xEF if little-endian.

- Today have x86 & Alpha little endian; Sparc & 68K, big endian; MIPS & ARM either.

Addressing Modes

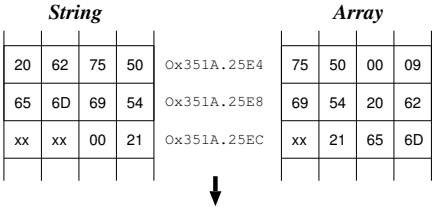
- An **addressing mode** tells the computer where the data for an instruction is to come from.
- Get a wide variety, e.g.

```
Register:              add  r1, r2, r3
Immediate:             add  r1, r2, #25
PC Relative:           beq  0x20
Register Indirect:     ldr  r1, [r2]
" + Displacement:      str  r1, [r2, #8]
Indexed:               movl r1, (r2, r3)
Absolute/Direct:       movl r1, $0xF1EA0130
Memory Indirect:       addl r1, ($0xF1EA0130)
```

- Most modern machines are *load/store* $\Rightarrow$ only support first five:
  - allow at most one memory ref per instruction (there are very good reasons for this)
- Note that CPU generally doesn't care *what* is being held within the memory.
- i.e. up to *programmer* to interpret whether data is an integer, a pixel or a few characters in a novel.

Representing Text

- Two main standards:
  1. ASCII: 7-bit code holding (English) letters, numbers, punctuation and a few other characters.
  2. Unicode: 16-bit code supporting practically all international alphabets and symbols.
- ASCII default on many operating systems, and on the early Internet (e.g. e-mail).
- Unicode becoming more popular (esp UTF-8!).
- In both cases, represent in memory as either strings or arrays: e.g. "Pub Time!"



- 0x492077697368206974207761732 03a2d28

Floating Point

- In many cases want to deal with very large or very small numbers.
- Use idea of "scientific notation", e.g. $n = m \times 10^e$
  - m is called the **mantissa**
  - e is called the **exponent**.

  e.g. $C = 3.01 \times 10^8$ m/s.
- For computers, use binary i.e. $n = m \times 2^e$, where m includes a "binary point".
- Both m and e can be positive or negative; typically
  - sign of mantissa given by an additional *sign* bit.
  - exponent is stored in a *biased* (*excess*) format.

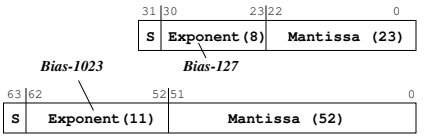
$\Rightarrow$ use $n = (-1)^s m \times 2^{e-b}$, where $0 \leq m < 2$ and b is the bias.

- e.g. 4-bit mantissa & 3-bit bias-3 exponent allows positive range $[0.001_2 \times 2^{-3}, 1.111_2 \times 2^4]$

$=$ [$(\frac{1}{8})(\frac{1}{8})$, $(\frac{15}{8})16$], or [$\frac{1}{64}$, 30]

Floating Point cont.

- In practice use IEEE floating point with *normalised* mantissa $m = 1.xx\ldots x_2$
  $\Rightarrow$ use $n = (-1)^s((1+m) \times 2^{e-b})$,
- Both single (`float`) and double (`double`) precision:



- IEEE fp reserves $e = 0$ and $e =$ max:
  - ± 0 (!): both e and m zero.
  - $\pm\infty$: $e =$ max, m zero.
  - NaNs : $e =$ max, m non-zero.
  - *denorms* : $e = 0$, m non-zero
- Normal positive range $[2^{-126}, \sim 2^{128}]$ for single, or $[2^{-1022}, \sim 2^{1024}]$ for double.
- NB: still only $2^{32}/2^{64}$ values — just spread out.

Data Structures

- Records / structures: each field stored as an offset from a *base address*.
- Variable size structures: explicitly store addresses (*pointers*) inside structure, e.g.

```
datatype rec = node of int * int * rec
             |     leaf of int;

val example = node(4, 5, node(6, 7, leaf(8)));
```

Imagine `example` is stored at address 0x1000:

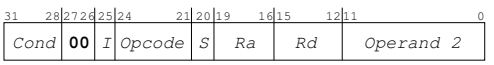
Address	Value	Comment
0x0F30	0xFFFF	Constructor tag for a leaf
0x0F34	8	Integer 8
⋮		
0x0F3C	0xFFFE	Constructor tag for a node
0x0F40	6	Integer 6
0x0F44	7	Integer 7
0x0F48	0x0F30	Address of inner node
⋮		
0x1000	0xFFFE	Constructor tag for a node
0x1004	4	Integer 4
0x1008	5	Integer 5
0x100C	0x0F3C	Address of inner node

Instruction Encoding

- An instruction comprises:

  a. an **opcode**: specify what to do.
  b. zero or more **operands**: where to get values

  e.g. `add r1, r2, r3` $\equiv$ | 1010111 | 001 | 010 | 011 |
- Old machines (and x86) use variable length encoding motivated by low code density.
- Most modern machines use fixed length encoding for simplicity. e.g. ARM ALU operations.



`and r13, r13, #31` = `0xe20dd01f` =

1110	00	1	0000	0	1101	1101	000000011111

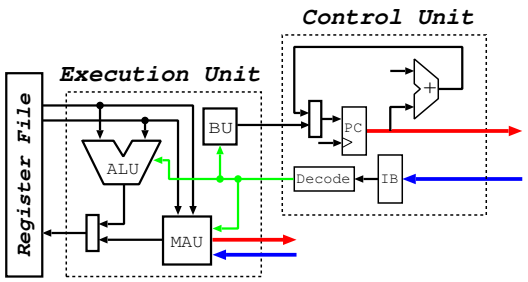
`bic r3, r3, r2` = `0xe1c33002` =

1110	00	0	1110	0	0011	0011	000000000010

`cmp r1, r2` = `0xe1510002` =

1110	00	0	1010	1	0001	0000	000000000010

Fetch-Execute Cycle Revisited



1. CU fetches & decodes instruction and generates (a) control signals and (b) operand information.
2. Inside EU, control signals select functional unit ("instruction class") and operation.
3. If ALU, then read one or two registers, perform operation, and (probably) write back result.
4. If BU, test condition and (maybe) add value to PC.
5. If MAU, generate address ("addressing mode") and use bus to read/write value.
6. Repeat *ad infinitum*.

Summary

You should now understand:

- Different forms of machine instructions,
- Different forms of addressing,
- Representing text and data structures,
- Floating point representation.

Next lecture: Buses and I/O devices

Background Reading:

- Hennessy/Patterson:
  - Chapter 3 - Machine Instructions (MIPS)
  - Section 4.8 - Floating Point