CS201 W'00 Lecture Notes
Lecture 3

Data Types and Encodings

Key idea of digital computer: **anything** can be encoded as a binary number.

Examples:

- Integers
- Floating-point numbers
- Characters
- Instructions
- Pointers
- Bit maps

Issues:

- Machine support
- Information content
- Compaction vs. ease of coding/decoding.

Machine Types

Machine types = data representations **directly supported** by machine instructions.

Examples on most machines:

Fixed-size integers (two's-complement and unsigned)

- Arithmetic instructions

Fixed-size bitmaps

- Bitwise logical instructions

Floating point numbers

- Arithmetic instructions

Pointers

- Jump instructions

Instructions

- Execution!

Less common examples:

Character strings

- Compare, move instructions

Binary-coded decimal

- Arithmetic instructions

Machine types vs. software types

Although type may be supported by instructions, machine code has no **type safety**:

- Data in memory or registers have no type.
- You can use any value as operand to any instruction.
- Allows **"puns"**, which are crucial!

— Address arithmetic puns pointers and integers.

— Fast multiplication puns integers and bit patterns.

— Compilers pun data and instructions!

- But easy to write nonsense!

Software types are those supported only by programmer convention.

- Examples: multi-precision integers, complex numbers, records.
- Vague dividing line: e.g., array support.
- Software-based types typically guarantee some safety properties.

Information content

Note that **all** types are fundamentally **discrete**.

- i.e., their values are 1-1 with the natural numbers $\{0, 1, 2, \ldots\}$.
- can represent values by numbers!

How much space does it take to represent a value of a particular type?

A boolean, with two values (true and false) requires just one bit.

The enumerated type

```
color = red | blue | green | yellow
```

with 4 discrete values, can be represented in two bits:

```
red    0 00
blue   1 01
green  2 10
yellow 3 11
```

So, e.g., we can fit 8 color values into a 16-bit word.

Can use bit manipulation instructions (AND,OR,SHIFT,etc.) to pack and unpack words fairly efficiently.

Similarly, the 256 ASCII characters fit into 8 bits, so we can use byte-oriented instructions. (Why **are** there 8 bits in a byte?)

Information Content(2)

In general: If a type has n discrete values, then it requires $\log_2 n$ bits.

What if we have only 3 colors?

```
color = red | green | blue
```

Then we can certainly continue using 4 bits per value, ignoring code 11; again get 8 values per 16 bit word.

E.g., the sequence
`red,blue,blue,green,green,red,blue,green`

which corresponds to `0,1,1,2,2,0,1,2`

could be represented by `0001011010000110`

But we should need only $\log_2 3 \approx 1.585$ bits, allowing us to store $\lfloor\frac{16}{1.585}\rfloor = 10$ values per 16 bit word!

How? Consider an alternative way of packing the color values, as a base-3 number:

01122012_3

$= 2 \times 3^0 + 1 \times 3^1 + 0 \times 3^2 + 2 \times 3^3 + \ldots.$

Since $2222222222_3 < 2^{16}$ we can fit 10 values in the word!

Of course, we have to do a lot more work to pack and unpack words into their values, so this may not be a good idea!

Example Time vs. Space Tradeoffs: Characters

Characters are normally represented by 8-bit ASCII codes.

- Fairly arbitrary, but incorporates alphabetical order.
- Originally a 7-bit code plus a parity bit.
- Good match with byte instructions.

If we want to represent strictly alphabetic data, need fewer than ($\log_2 256$) bits [homework problem].

But usually don't bother, because byte access is so much faster.

If we want to represent strictly numeric data, need very few bits ($\log_2 10 =\approx 3.3$ bits per decimal digit), so normally convert numeric strings to binary numbers before storage or computation.

But if we do a lot of I/O with numbers, can spend a lot of time doing conversions. May be better to use **binary coded decimal** representation. (4 bits per digit).

- Cheap to convert to/from ASCII (using bit operations).
- Some machines have built-in instructions for computing on BCD representation.

Compression

Is $\log_2 n$ bits per value the most compact representation we can use? In general, no!

These representations use a **fixed-length** encoding (the same number of bits to represent every value).

If all values are equally possible, this is the best we can do. But if some values are more likely than others, we can save space by using a **variable-length** encoding:

- More common values get short representations.
- Less common values get longer representations.
- An "average" value needs fewer bits than in in fixed-length scheme.

Example: 4 colors, with known frequency distribution:

```
red    50% 0
blue   30% 10
green  10% 110
yellow 10% 111
```

E.g., the sequence `red,blue,red,red,green,blue,yellow` is represented by `0100011010111`

Examples: • English text has many more `e`'s than `x`'s!
• Some machine instructions are executed much more often than others.

General method: Huffman codes

Example : Instruction formats

Design of instruction format is based on many parameters:

- Number of operators (**opcodes**)
- Number of **operands**
- Length and **mode** of operands

Instruction length = opcode + operands

Generally, **shorter** instructions are better

- Require less memory
- Faster to fetch from memory

(Variable-length instructions also possible.)

But **regularity** of format is also good

- Faster parallel **decoding** in hardware

Other important goals:

Orthogonality

- All opcodes should be available in all (sensible) modes and all registers should be usable with any opcode.

Simplicity

- Main client is compiler-writer, so a few clear choices with easily computed costs are desirable.

SPARC Instruction Formats

Every instruction has same length and similar layout.

[See text Figure 5-14]

x86 Instruction Formats

Instructions have variable lengths and irregular layout.

Driven by backward compatibility requirements.

[See text Figure 5-13]

Java Virtual Machine (JVM) Instruction Formats

Instructions regular but variable length.

[See text Figure 5-15]

For important ops, most common arg values are encoded in (single byte) opcode, e.g. `ILOAD 1` vs. `ILOAD_1`

Opcodes

Must reserve enough bits in instruction to hold needed number of opcodes.

n opcodes require $\log_2 n$ bits.

Can use **expanding** opcodes to avoid having huge opcode field in every instruction.

SPARC Example:

- Bits 30-31 indicate overall format (1-3):

00	branches; `sethi` (format 2)	specific op in bits 22-24
01	`call` (format 1)	
10	ALU instructions, etc. (format 3)	specific op in bits 19-24
11	memory instructions (format 3)	specific op in bits 19-24

Another use is to shorten instruction stream

Example: JVM uses `WIDE` prefix byte to mark opcode variants that take larger operand fields.

Operands

(Terminology often vague; should distinguish between **operands** and **memory addresses**.)

How many operands per instruction? How many bits per operand?

Many instructions (e.g, arithmetic) naturally 3-operand.

- SPARC, other RISCs are explicitly 3-operand.
- By making destination same as one source, get **destructive** 2-operand instructions (e.g., X86).
- Some early machines used a single **accumulator** register, so were 1-operand.
- Stack machines (like JVM) are 0-operand.

Fewer operands means fewer bits per instructions, but may mean more instructions! [See homework problem.]

What kind of operands?

In principle, every operand could be an arbitrary memory address, but

- this is slow!
- it takes lots of bits to specify each address
- every address is static

Instead, all modern machines take note of

- repeated references ⇒ put data in small set of registers
- locality of references ⇒ address data with small offset from base address

So most operands are register specifiers (e.g., 5 bits on SPARC, 3 bits on x86) or an offset specifier (e.g., 13 bits on SPARC, 1-4 bytes on x86).

Addressing Modes

Immediate

SPARC: `mov 6,%l3`

x86: `MOV EAX,6`

JVM: `ICONST 6`

- Operand is (static) value is embedded in instruction, so no extra fetch is needed.
- Only small values fit (unless variable-length instructions are used).

Direct

x86: `MOV EBX, COUNT`

- Operand is memory location at (static) absolute address embedded in instruction.
- Used for globals; requires relocation at link-time.
- Takes lots of instruction bits.

Register

SPARC: `add %l0,%l1,%l0`

x86: `ADD EAX,EBX`

- Operand is (dynamic) register value.
- Fastest possible access.

Register indirect

SPARC: `ld [%o1],%l2` (actually indexed)

x86: `MOV EAX,[EBX]`

- Operand is memory location at (dynamic) absolute address in register.
- Doesn't waste instruction bits.
- Allows stepping through memory

Indexed

SPARC: `ld [%fp+12],%l2`

x86: `MOV EAX,[EBP+12]`

JVM: `ILOAD 12`

- Operand is memory location at address formed by sum of (dynamic) register value and (static) offset in instruction.
- Very convenient for referencing elements in stack frame.

Based Indexed

SPARC: `ld[%fp+%o2],%l2`

x86: `MOV EAX,[EBP+ESI]`

- Operand is memory location at address formed by sum of two (dynamic) register values.

Stack Machines

JVM: `ISUB`

- Operands are implicitly at fixed positions relative to top of stack.

Branch Instructions

Targets of branch instructions could be specified using any of the memory-address modes, e.g., direct, register indirect, indexed.

Usually (implicitly) use indexed mode relative to **program counter**, which gives **position-independent code**.

- Can be relocated without alteration during linking.
- Since branch targets are usually nearby, don't need large offset.

Need at least one way to jump to arbitrary addresses, though.