Machine Language, Assemblers, and Compilers

Long long time ago, I can still remember how mnemonics used to make me smile... And I knew that with just the opcode names that I could play those BSim games and maybe hack some macros for a while. But 6.004 gave me shivers with every lecture they delivered. Bad news at the door step, I couldn't read one more spec. I can't remember if I tried to get factorial-optimized, but something touched my nerdish pride the day my Beta died. And I was singing...

References (on web site):
- Documentation
- BSIM reference
- Notes on C Language

Quiz 2 TOMORROW!

β Machine Language: 32-bit instructions

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>r_c</th>
<th>r_a</th>
<th>r_b</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- arithmetic: ADD, SUB, MUL, DIV
- compare: CMPEQ, CMPLT, CMPLE
- boolean: AND, OR, XOR
- shift: SHL, SHR, SRA

Ra and Rb are the operands, Rc is the destination. R31 reads as 0, unchanged by writes.

OPCODE: r_c r_a 16-bit signed constant

- arithmetic: ADDC, SUBC, MULC, DIVC
- compare: CMPEQC, CMPLTC, CMPLEC
- boolean: ANDC, ORC, XORC
- shift: SHLC, SHRC, SRAC

jump: JMP (const not used)

memory access: LD, ST (const = byte offset from Reg[ra])

How can we improve the programmability of the Beta?

Encoding Binary Instructions

32-bit (4-byte) ADD instruction:

```
000000001000000110000010000000000000000
```

OpCode  Rc  Ra  Rb  (unused)


But, most of us would prefer to write

```
ADD (R2, R3, R4)  (ASSEMBLER)
```

or, better yet,

```
a = b+c;  (High Level Language)
```

Software Approaches: INTERPRETATION, COMPILATION

Interpretation

Turing’s model of Interpretation:

- Start with some hard-to-program universal machine, say M1
- Write a single program for M1 which mimics the behavior of some easier machine, say M2
- Result: a “virtual” M2

“Layers” of interpretation:

- Often we use several layers of interpretation to achieve desired behavior, eg:
  - X86 (Pentium), running
    - Scheme, running
      - Application, interpreting
        - Data.

Structure  | Language
---|---
DATA | Scheme
Application Interprets | X86 Instructions
APPLICATION | HARDWARE

Software Approaches: INTERPRETATION, COMPILATION
Compilation

Model of Compilation:

- Given some hard-to-program machine, say $M_1$...
- Find some easier-to-program language $L_2$ (perhaps for a more complicated machine, $M_2$); write programs in that language.
- Build a translator (compiler) that translates programs from $M_2$’s language to $M_1$’s language. May run on $M_1$, $M_2$, or some other machine.

Interpretation & Compilation: two tools for improving programmability...

- Both allow changes in the programming model.
- Both afford programming applications in platform (e.g., processor) independent languages.
- Both are widely used in modern computer systems!

Interpretation vs Compilation

<table>
<thead>
<tr>
<th>How it treats input “x+2”</th>
<th>Interpretation</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>computes x+2</td>
<td>generates a program that computes x+2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>When it happens</th>
<th>Interpretation</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>During execution</td>
<td>Program Execution</td>
<td>Before execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What it complicates/slow</th>
<th>Interpretation</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Execution</td>
<td>Program Development</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decisions made at</th>
<th>Interpretation</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time</td>
<td>Compile Time</td>
<td></td>
</tr>
</tbody>
</table>

Major design choice we’ll see repeatedly: do it at Compile time or at Run time?

Software: Abstraction Strategy

Initial steps: compilation tools

Assembler (UASM):
- Symbolic representation of machine language
- Hides: bit-level representations, hex locations, binary values

Compiler (C):
- Symbolic representation of algorithm
- Hides: Machine instructions, registers, machine architecture

Subsequent steps: interpretive tools

- Operating system
  - Hides: Resource (memory, CPU, I/O) limitations and details
- Apps (e.g., Browser)
  - Hides: Network; location; local parameters

A Program for Writing Programs

UASM - the 6.004 (Micro) Assembly Language

UASM:
1. A Symbolic LANGUAGE for representing strings of bits
2. A PROGRAM (“Assembler” = primitive compiler) for translating UASM source to binary.
**UASM Source Language**

A UASM SOURCE FILE contains, in symbolic text, values of successive bytes to be loaded into memory... e.g. in

- 37  -3  255  \textit{decimal (default)};
- 0b100101  \textit{binary (note the “Ob” prefix)};
- 0x25  \textit{hexadecimal (note the “0x” prefix)};

Values can also be expressions; e.g., the source file

\[
37+0b10-0x10  \quad 24-0x1 \quad 4*0b110-1 \quad 0xF7&0x1F
\]

generates 4 bytes of binary output, each with the value 23!

---

**Labels (Symbols for Addresses)**

LABELS are symbols that represent memory addresses. They can be set with the following special syntax:

- \texttt{x:} is an abbreviation for \texttt{”x = .”}

An Example--

\[
\begin{array}{c}
\text{1000:} & 09 & 04 & 01 & 00 \\
\text{1004:} & 31 & 24 & 19 & 10 \\
\text{1008:} & 79 & 64 & 51 & 40 \\
\text{100c:} & E1 & C4 & A9 & 90 \\
\text{1010:} & 10 & \ldots & \ldots & \ldots
\end{array}
\]

\[
\begin{array}{c}
\text{sqrs:} & 01 & 4 & 9 \\
\text{slen:} & 16 & 25 & 36 & 49 \\
\text{4*0b110-1:} & 64 & 81 & 100 & 121 \\
\text{0xF7&0x1F:} & 144 & 169 & 196 & 225
\end{array}
\]

---

**Symbolic Gestures**

We can also define SYMBOLS for use in source programs:

\[
\begin{align*}
x &= 0x1000 & \text{A variable location} \\
y &= 0x1004 & \text{Another variable}
\end{align*}
\]

| Symbolic names for registers: 
| R0 = 0 \\
| R1 = 1 \\
| ... \\
| R31 = 31

Special variable "." (period) means next byte address to be filled:

\[
\begin{align*}
. &= 0x100 & \text{Assemble into 100} \\
1 & 2 & 3 & 4 \\
\text{five} &= . & \text{Symbol “five” is 0x104} \\
5 & 6 & 7 & 8 \\
. &= .+16 & \text{Skip 16 bytes} \\
9 & 10 & 11 & 12
\end{align*}
\]

---

**Mighty Macroinstructions**

Macros are parameterized abbreviations, or shorthand

- | Macro to generate 4 consecutive bytes:
  \texttt{.macro consec(n) \ n \ n+1 \ n+2 \ n+3}
  \texttt{Invocation of above macro:
  \texttt{consec(37)}}

Has same effect as:

\[
37 \quad 38 \quad 39 \quad 40
\]

Here are macros for breaking multi-byte data types into byte-sized chunks

\[
\begin{align*}
. &= 0x100 & \text{Assemble into bytes, little-endian (least-sig byte last)}
\end{align*}
\]

\[
\begin{align*}
. &= 0x100 & \text{Macro \texttt{WORD(x)} as \texttt{(x\%256),(x/256)\%256}}
\end{align*}
\]

\[
\begin{align*}
. &= 0x100 & \text{Macro \texttt{LONG(x)} as \texttt{WORD(x) WORD(x >> 16)}}
\end{align*}
\]

Boo, that’s hard to read.

Maybe, those big-endian types do have a point.

\[
\begin{align*}
\text{Mem:} & 0x100 \quad 0x101 \quad 0x102 \quad 0x103
\end{align*}
\]
Assembly of Instructions

<table>
<thead>
<tr>
<th>OP CODE</th>
<th>RC</th>
<th>RA</th>
<th>RB</th>
<th>UNUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100000000001111110000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Assemble Beta op instructions
  ```assemble
  .macro betaop(OP,RA,RB,RC) {     .align 4
      LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
  }
  ```

- Assemble Beta opc instructions
  ```assemble
  .macro betaopc(OP,RA,CC,RC) {     .align 4
      LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000))
  }
  ```

- Assemble Beta branch instructions
  ```assemble
  .macro betabr(OP,RA,RC,LABEL) betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
  ```

For Example:
```
ADDC(R15, -32768, R0) --> betaopc(0x30,15,-32768,0)
```

**Example Assembly**

```
ADDC(R3, 1234, R17)     
  expand ADDC macro with RA=R3, C=1234, RC=R17
betaopc(0x30,R3,1234,R17)  
  expand betaopc macro with OP=0x30, RA=R3, CC=1234, RC=R17
  .align 4
  LONG((0x30<<26)+((R17%32)<<21)+((R3%32)<<16)+(1234 % 0x10000))
  expand LONG macro with X=0xC22304D2
  WORD(0xC22304D2) WORD(0xC22304D2 >> 16)
  expand first WORD macro with X=0xC22304D2
  0xC22304D2%256  (0xC22304D2/256)%256  WORD(0xC223)
  evaluate expressions, expand second WORD macro with X=0xC223
  0xD2  0x04  0xC223%256  (0xC223/256)%256  evaluate expressions
  0xD2  0x04  0x23  0xC2
```
Abstraction step 2:

High-level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm:

```c
class Employee {
    char *Name; // Employee's name.
    long Salary; // Employee's salary.
    long Points; // Brownie points.
}
```

We've used (and will continue to use throughout 6.004) C, a "mature" and common systems programming language. Modern popular alternatives include C++, Java, Python, and many others.

Why use these, not assembler?

- readable
- concise
- unambiguous
- portable (algorithms frequently outlast their HW platforms)
- Reliable (type checking, etc)

Reference: C handout (6.004 web site)

How Compilers Work

Contemporary compilers go far beyond the macro-expansion technology of UASM. They

- Perform sophisticated analyses of the source code
- Invoke arbitrary algorithms to generate efficient object code for the target machine
- Apply "optimizations" at both source and object-code levels to improve run-time efficiency.

Compilation to unoptimized code is pretty straightforward... following is a brief glimpse.

Compiling Expressions

C code:

```c
int x, y;
y = (x-3)*(y+123456)
```

Beta assembly code:

```assembly
LD(x, r1)
SUBC(r1,3,r1)
LD(y, r2)
LD(C, r3)
ADD(r2,r3,r2)
MUL(r2,r1,r1)
```

| VARIABLES are assigned memory locations and accessed via LD or ST |
| OPERATORS translate to ALU instructions |
| SMALL CONSTANTS translate to "literal-mode" ALU instructions |
| LARGE CONSTANTS translate to initialized variables |

Data Structures: Arrays

The C source code:

```c
int Hist[100];
Hist[score] += 1;
```

might translate to:

```assembly
<score in r1> |
MULC(r1,4,r2) | index -> byte offset |
ADD(r0,1,r0) | increment |
ST(r0,hist,r2) | hist[score] |
```

Address:

CONSTANT base address + VARIABLE offset computed from index
### Data Structures: Structs

```
struct Point {
    int x, y;
} P1, P2, *p;
```

might translate to:

```
P1: .x = 157
P2: .x = 157
```

### Memory:

```
<table>
<thead>
<tr>
<th>Offset for x component</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Offset for y component</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
```

### Conditionals

**C code:**
```
if (expr) {
    STUFF
}
else {
    STUFF2
}
```

**Beta assembly:**
```
if (compile expr into rx)
    BF(rx, Lendif)
    (compile STUFF)
Lendif:
else {
    (compile STUFF2)
    BR(Lendif)
}
```

There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:
```
if (y > 32) {
    x = x + 1;
}
```

might compile to:
```
LD(y, R1)
CMPLEC(R1, 32, R1)
BT(R1, Lendif)
ADDCC(R2, 1, R2)
Lendif:
```

### Loops

**Beta assembly:**
```
Lwhile:
    (compile expr into rx)
    BF(rx, Lendwhile)
    (compile STUFF)
Ltest:
    (compile expr into rx)
    BT(rx, Lwhile)
Lendwhile:
```

**Alternate Beta assembly:**
```
BR(Ltest)
Lwhile:
    (compile STUFF)
Ltest:
    (compile expr into rx)
    BT(rx, Lwhile)
Lendwhile:
```

**C code:**
```
while (expr) {
    STUFF
}
```

Compilers spend a lot of time optimizing in and around loops.
- moving all possible computations outside of loops
- "unrolling" loops to reduce branching overhead
- simplifying expressions that depend on "loop variables"

### Our Favorite Program

```
int n = 20, r;
for (int i = 0; i < n; i++) {
    r = r * n;
}
```

Cleverness: None... straightforward compilation

(11 instructions in loop...)

### Optimizations

- moving all possible computations outside of loops
- "unrolling" loops to reduce branching overhead
- simplifying expressions that depend on "loop variables"
Optimizations

```
int n = 20, r;
n: LONG(20)
r: LONG(0)

r = 1;
start:
  ADDC(r31, 1, r0)
  ST(r0, r)
  LD(n, r1)  | keep n in r1
  LD(r, r3)  | keep r in r3
loop:
  CMPLT(r31, r1, r2)  | done
  BF(r2, done)  | why?
  MUL(r1, r3, r3)
  SUBC(r1, 1, r1)
  BR(loop)
done:
  ST(r1, n)  | save final n
  ST(r3, r)  | save final r
```

Cleverness:
We move LDs/STs out of loop!
(Still, 5 instructions in loop...)

Really Optimizing...

```
int n = 20, r;
n: LONG(20)
r: LONG(0)

r = 1;
start:
  LD(n, r1)  | keep n in r1
  ADDC(r31, 1, r3)  | keep r in r3
  BEQ(r1, done)  | why?
loop:
  while (n > 0)
  {   MUL(r1, r3, r3)
      SUBC(r1, 1, r1)
      n = n - 1;
      done:
      ST(r1, n)  | save final n
      ST(r3, r)  | save final r
```