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…

When I find my code in tons of trouble, Friends and colleagues come to me, Speaking words of wisdom: "Write in C."

References: β Documentation; Lab #5B; Notes on C Language
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, SAR
  
  Ra and Rb are the operands, Rc is the destination. R31 reads as 0, unchanged by writes.

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>r_c</th>
<th>r_a</th>
<th>16-bit signed constant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **arithmetic:** ADDC, SUBC, MULC, DIVC
- **compare:** CMPEQC, CMPLTC, CMPLEC
- **boolean:** ANDC, ORC, XORC
- **shift:** SHLC, SHRC, SARC
- **branch:** BNE/BT, BEQ/BF (const = word displacement from PC<sub>next</sub>)
- **jump:** JMP (const not used)
- **memory access:** LD, ST (const = byte offset from Reg[ra])

  Two’s complement 16-bit constant for numbers from -32768 to 32767; sign-extended to 32 bits before use.

**How can we improve the programmability of the Beta?**
Encoding Binary Instructions

32-bit (4-byte) ADD instruction:

```
10000000 1000001 1000011 00000000 00000000
```

OpCode  Rc  Ra  Rb  (unused)

Means, to BETA,  \( \text{Reg}[4] = \text{Reg}[2] + \text{Reg}[3] \)

But, most of us would prefer to write

\[
\text{ADD} (R2, R3, R4) \quad \text{(ASSEMBLER)}
\]

or, better yet,

\[
a = b+c; \quad \text{(High Level Language)}
\]

Software Approaches: INTERPRETATION, COMPILATION
Interpretation

Turing’s model of Interpretation:

- Start with some hard-to-program universal machine, say $M_1$
- Write a single program for $M_1$ which mimics the behavior of some easier machine, say $M_2$
- Result: a “virtual” $M_2$

“Layers” of interpretation:

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

There are some characteristic differences between these two powerful tools...

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

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

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

<table>
<thead>
<tr>
<th>Decisions made at</th>
<th>Interpretation</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run Time</td>
<td>Compile Time</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
Abstraction step 1:

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 “Ox” prefix)};

Values can also be expressions; eg, the source file

37+0b10-0x10   24-0x1   4*0b110-1   0xF7&0x1F

\textit{generates 4 bytes of binary output, each with the value 23!}
Symbolic Gestures

We can also define SYMBOLS for use in source programs:

```
x = 0x1000  | A variable location
y = 0x1004  | Another variable
```

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

Special variable “.” (period) means next byte address to be filled:
```
. = 0x100  | Assemble into 100
  1  2  3  4
five = .  | Symbol “five” is 0x104
  5  6  7  8
. = . +16  | Skip 16 bytes
  9 10 11 12
```
Labels (Symbols for Addresses)

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

\[ x : \text{ is an abbreviation for } "x = ." \]

An Example--

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>09 04 01 00</td>
</tr>
<tr>
<td>1004</td>
<td>31 24 19 10</td>
</tr>
<tr>
<td>1008</td>
<td>79 64 51 40</td>
</tr>
<tr>
<td>100c</td>
<td>E1 C4 A9 90</td>
</tr>
<tr>
<td>1010</td>
<td>00 00 00 10</td>
</tr>
</tbody>
</table>

. = 0x1000

sqr: 0 1 4 9

16 25 36 49

64 81 100 121

144 169 196 225

slen: LONG(. - sqr)
Mighty Macroinstructions

*Macros are parameterized abbreviations, or shorthand*

| Macro to generate 4 consecutive bytes: |
| .macro consec(n) n n+1 n+2 n+3 |

| Invocation of above macro: |
| consec(37) |

Has same effect as:

37 38 39 40

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

| Assemble into bytes, little-endian: |
| .macro WORD(x) x%256 (x/256)%256 |
| .macro LONG(x) WORD(x) WORD(x >> 16) |
| . = 0x100 |
| LONG(0xdeadbeef) |

Has same effect as:

0xef 0xbe 0xad 0xde

Mem: 0x100 0x101 0x102 0x103
Assembly of Instructions

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>RC</th>
<th>RA</th>
<th>RB</th>
<th>UNUSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>110000</td>
<td>00000011111000000000000000000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-32768 = 10000000000000000000000000000000
ADDC = 0x30 = 110000 15 = 01111 0 = 00000

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

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

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

For Example:
ADDC(R15, -32768, R0) --> betaopc(0x31,15,-32768,0)
Finally, Beta Instructions

<table>
<thead>
<tr>
<th>BETA Instructions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>.macro ADD(RA, RB, RC) betaop(0x20, RA, RB, RC)</td>
</tr>
<tr>
<td>.macro ADDC(RA, C, RC) betaopc(0x30, RA, C, RC)</td>
</tr>
<tr>
<td>.macro AND(RA, RB, RC) betaop(0x28, RA, RB, RC)</td>
</tr>
<tr>
<td>.macro ANDC(RA, C, RC) betaopc(0x38, RA, C, RC)</td>
</tr>
<tr>
<td>.macro MUL(RA, RB, RC) betaop(0x22, RA, RB, RC)</td>
</tr>
<tr>
<td>.macro MULC(RA, C, RC) betaopc(0x32, RA, C, RC)</td>
</tr>
<tr>
<td>.macro LD(RA, CC, RC) betaopc(0x18, RA, CC, RC)</td>
</tr>
<tr>
<td>.macro LD(CC, RC) betaopc(0x18, R31, CC, RC)</td>
</tr>
<tr>
<td>.macro ST(RC, CC, RA) betaopc(0x19, RA, CC, RC)</td>
</tr>
<tr>
<td>.macro ST(RC, CC) betaopc(0x19, R31, CC, RC)</td>
</tr>
<tr>
<td>.macro BEQ(RA, LABEL, RC) betabr(0x1D, RA, RC, LABEL)</td>
</tr>
<tr>
<td>.macro BEQ(RA, LABEL) betabr(0x1D, RA, r31, LABEL)</td>
</tr>
<tr>
<td>.macro BNE(RA, LABEL, RC) betabr(0x1E, RA, RC, LABEL)</td>
</tr>
<tr>
<td>.macro BNE(RA, LABEL) betabr(0x1E, RA, r31, LABEL)</td>
</tr>
</tbody>
</table>

(from beta.uasm)

Convenience macros so we don’t have to specify R31…
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
```
Don’t have it? Fake it!

Convenience macros can be used to extend our assembly language:

```
.macro MOVE(RA,RC)  ADD(RA,R31,RC) | Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC) ADDC(R31,C,RC) | Reg[RC] <- C
.macro COM(RA,RC) XORC(RA,-1,RC) | Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC) SUB(R31,RB,RC) | Reg[RC] <- -Reg[RB]
.macro NOP() ADD(R31,R31,R31) | do nothing

.macro BR(LABEL) BEQ(R31,LABEL) | always branch
.macro BR(LABEL,RC) BEQ(R31,LABEL,RC) | always branch
.macro CALL(LABEL) BEQ(R31,LABEL,LP) | call subroutine
.macro BF(RA,LABEL,RC) BEQ(RA,LABEL,RC) | 0 is false
.macro BF(RA,LABEL) BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC) BNE(RA,LABEL,RC) | 1 is true
.macro BT(RA,LABEL) BNE(RA,LABEL)
```

<table>
<thead>
<tr>
<th>Multi-instruction sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>.macro PUSH(RA) ADDC(SP,4,SP) ST(RA,-4,SP)</td>
</tr>
<tr>
<td>.macro POP(RA) LD(SP,-4,RA) ADDC(SP,-4,SP)</td>
</tr>
</tbody>
</table>

(from beta.uasm)
Abstraction step 2:

High-level Languages

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

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

/* Annual raise program. */
Raise(struct Employee P[100])
{ int i = 0;
  while (i < 100)
  { struct Employee *e = &P[i];
    e->Salary =
      e->Salary + 100 + e->Points;
    e->Points = 0;  /* Start over! */
    i = i+1;
  }
}
```

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.

What compilers do is not all that complicated. (at least, in principle)
Compiling Expressions

C code:

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

Beta assembly code:

```
x:  LONG(0)
y:  LONG(0)
c:  LONG(123456)
...
LD(x, r1)
SUBC(r1,3,r1)
LD(y, r2)
LD(C, r3)
ADD(r2,r3,r2)
MUL(r2,r1,r1)
ST(r1,y)
```

- 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:

```
; hist: =.+4*100 | Leave room for 100 ints
...
<score in r1>
MULC(r1,4,r2) | index -> byte offset
LD(r2,hist,r0) | hist[score]
ADDC(r0,1,r0) | increment
ST(r0,hist,r2) | hist[score]
```

Address:

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

```c
struct Point
    { int x, y;
    } P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 157;
```

might translate to:

```assembly
P1: .+=.+8
P2: .+=.+8
x=0 | Offset for x component
y=4 | Offset for y component
...
ADDC(r31,157,r0) | r0 <- 157
ST(r0,P1+x) | P1.x = 157
...
<p in r3>
ST(r0,y,r3) | p->y = 157;
```

Address:
VARIABLE base address + CONSTANT component offset
Conditionals

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

Beta assembly:
(compile expr into rx)
BF(rx, Lendif)
(compile STUFF)
Lendif:
(compile STUFF1)
BR(Lendif)
Lelse:
(compile STUFF2)
Lendif:

There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:

if (y > 32)
{
  x = x + 1;
}

compiles to:

LD(y,R1)
CMPLEC(R1,32,R1)
BT(R1,Lendif)
ADDC(R2,1,R2)
Lendif:

there's no >32 instruction!
Loops

C code:
while (expr) {
    STUFF
}

Beta assembly:
Lwhile:
    (compile expr into rx)
    BF(rx,Lendwhile)
    (compile STUFF) 
    BR(Lwhile)
Lendwhile:

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

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”
Optimizing Our Favorite Program

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

r = 1;
{ADDC(r31, 1, r0)
ST(r0, r)}

loop:
{LD(n, r1)
CMPLT(r31, r1, r2)
BF(r2, done)
LD(r, r3)
LD(n, r1)
MUL(r1, r3, r3)
ST(r3, r)
LD(n, r1)
SUBC(r1, 1, r1)
ST(r1, n)
BR(loop)}

done:

Cleverness:
None… straightforward compilation

(11 instructions in loop...)
```

Cleverness:
None… straightforward compilation

```
int n = 20, r;
r = 1;
while (n > 0)
{
    r = r*n;
    n = n-1;
}
```
Optimizations

```c
int n = 20, r;

n: LONG(20)
r: LONG(0)

r = 1;

ADDC(r31, 1, r0)
ST(r0, r)
LD(n, r1) ; keep n in r1
LD(r, r3) ; keep r in r3

loop:

while (n > 0) {
  CMPLT(r31, r1, r2)
  BF(r2, done)
  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…

\begin{align*}
\text{int } n &= 20, r; \\
n &: \text{ LONG}(20) \\
r &: \text{ LONG}(0) \\
\end{align*}

\begin{align*}
r &= 1; \\
\text{LD}(n, r1) &; \text{ keep } n \text{ in } r1 \\
\text{ADDC}(r31, 1, r3) &; \text{ keep } r \text{ in } r3 \\
\text{BEQ}(r1, \text{done}) &; \text{ why?}
\end{align*}

\begin{align*}
\text{while } (n > 0) \\
\text{loop:} \\
\{ & \text{ r = r*n; } \\
& \text{ n = n-1; } \\
\}
\text{done:} \\
& \text{ST}(r1, n) &; \text{ save final } n \\
& \text{ST}(r3, r) &; \text{ save final } r
\end{align*}

Cleverness:
We avoid overhead of conditional!
(Now 3 instructions in loop...)

\textbf{UNFORTUNATELY}, \ 20! = 2,432,902,008,176,640,000 > 2^{61} \\
12! = 479,001,600 = 0x1c8cfc00
Coming Attractions:

Procedures & Stacks