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 Beta 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."

Today’s handouts:
- Lecture slides
- Lab 3 due Thursday!

- Beta BEQ/BNE/JMP instructions
- Review of assembly language
- Simple compilation strategy
Can We Solve Factorial With ALU Instructions?

- \texttt{factorial(N) = N! = N \times (N-1) \times \ldots \times 1}

- No! Factorial needs to \textit{loop}
- So far we can only encode sequences of operations on registers
- Need a way to change the PC based on data values!
  – Called “branching”. If the branch is taken, the PC is changed. If the branch is not taken, keep executing sequentially.
Beta Branch Instructions

The Beta’s *branch instructions* provide a way to conditionally change the PC to point to a nearby location...

... and, optionally, remembering (in Rc) where we came from (useful for procedure calls).

<table>
<thead>
<tr>
<th>BEQ or BNE</th>
<th>rc</th>
<th>ra</th>
<th>16-bit signed constant</th>
</tr>
</thead>
</table>

BEQ(ra,offset,rc): Branch if equal

NPC $\leftarrow$ PC + 4
Reg[rc] $\leftarrow$ NPC
if (Reg[ra] == 0)
  PC $\leftarrow$ NPC + 4*offset
else
  PC $\leftarrow$ NPC

BNE(ra,offset,rc): Branch if not equal

NPC $\leftarrow$ PC + 4
Reg[rc] $\leftarrow$ NPC
if (Reg[ra] != 0)
  PC $\leftarrow$ NPC + 4*offset
else
  PC $\leftarrow$ NPC

“offset” is a SIGNED CONSTANT encoded as part of the instruction!

offset = distance in words to branch target, counting from the instruction following the BEQ/BNE. Range: -32768 to +32767.
Can We Solve Factorial Now?

• Yes!

```c
int a = 1;
int b = N;
do {
    a = a * b;
    b = b - 1;
} while (b != 0)
```

```asm
// Assume r1 = N
ADDC(r31, 1, r0) // r0 = 1
L:MUL(r0, r1, r0) // r0 = r0 * r1
SUBC(r1, 1, r1) // r1 = r1 - 1
BNE(r1, L, r31) // if r1 != 0, run MUL next
// at this point, r0 = N!
```

```
```

```
```
```
```
```
```
```

6.004 Computation Structures

L10: Assembly Language & Compilers, Slide #4
**Beta JMP Instruction**

Branches transfer control to some predetermined destination specified by a constant in the instruction. It will be useful to be able to transfer control to a computed address.

\[
\begin{array}{|c|c|c|}
\hline
011011 & r_c & r_a \\
\hline
\end{array}
\]

\[\text{JMP}(Ra,Rc): \quad \text{Reg}[Rc] \leftarrow PC + 4 \]
\[PC \leftarrow \text{Reg}[Ra]\]

Useful for procedure call return…

\[\begin{array}{l}
\text{...} \\
[0x100] \text{BEQ}(R31, \text{sqrt}, R28) \\
\text{...} \\
[0x678] \text{BEQ}(R31, \text{sqrt}, R28) \\
\text{...}
\end{array}\]

\[\begin{array}{l}
\text{sqrt:} \\
\text{...} \\
\text{JMP}(R28, R31) \\
\sim \\
1^{\text{st}} \text{time}: PC \leftarrow 0x104 \\
2^{\text{nd}} \text{time}: PC \leftarrow 0x67C
\end{array}\]
Beta ISA Summary

• Storage:
  – Processor: 32 registers (r31 hardwired to 0) and PC
  – Main memory: Up to 4 GB, 32-bit words, 32-bit byte addresses, 4-byte-aligned accesses

<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>

32 bits

• Instruction formats:

<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>

• Instruction classes:
  – ALU: Two input registers, or register and constant
  – Loads and stores: access memory
  – Branches, Jumps: change program counter
Programming Languages

32-bit (4-byte) ADD instruction:

```
1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
```

<table>
<thead>
<tr>
<th>opcode</th>
<th>rc</th>
<th>ra</th>
<th>rb</th>
<th>(unused)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

We’d rather write in *assembly language*:

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

or better yet a *high-level language*:

\[
a = b + c;
\]
Assembly Language

• Abstracts bit-level representation of instructions and addresses

• We’ll learn UASM (“microassembler”), built into BSim

• Main elements:
  – Values
  – Symbols
  – Labels (symbols for addresses)
  – Macros
Example UASM Source File

\[ N = 12 \quad \text{// loop index initial value} \]
\[ \text{ADDC}(r31, N, r1) \quad \text{// r1 = loop index} \]
\[ \text{ADDC}(r31, 1, r0) \quad \text{// r0 = accumulated product} \]
\[ \text{loop: MUL}(r0, r1, r0) \quad \text{// r0 = r0 * r1} \]
\[ \text{SUBC}(r1, 1, r1) \quad \text{/* r1 = r1 - 1 */} \]
\[ \text{BNE}(r1, \text{loop}, r31) \quad \text{// if r1 != 0, NextPC=loop} \]

- **Comments** after //, ignored by assembler (also /*...*/)
- **Symbols** are symbolic representations of a constant value (they are NOT variables!)
- **Labels** are symbols for addresses
- **Macros** expand into sequences of bytes
  - Most frequently, macros are instructions
  - We can use them for other purposes
How Does It Get Assembled?

Text input

\[
\begin{align*}
N &= 12 \\
\text{ADDC}(r31, N, r1) \\
\text{ADDC}(r31, 1, r0) \\
\text{loop: MUL}(r0, r1, r0) \\
\text{SUBC}(r1, 1, r1) \\
\text{BNE}(r1, \text{loop}, r31)
\end{align*}
\]

Binary output

\[
\begin{align*}
110000 \ 0000 \ 1111 \ 00000000 \ 00001100 [0x00] \\
110000 \ 0000 \ 1111 \ 00000000 \ 00000001 [0x04] \\
100010 \ 0000 \ 0000 \ 00001 \ 00000000000 [0x08] \\
\text{...}
\end{align*}
\]

Symbol table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>r31</td>
<td>31</td>
</tr>
<tr>
<td>N</td>
<td>12</td>
</tr>
<tr>
<td>loop</td>
<td>8</td>
</tr>
</tbody>
</table>
Registers are Predefined Symbols

- \( r_0 = 0, \ldots, r_{31} = 31 \)
- Treated like normal symbols:

\[
\text{ADDC}(r_{31}, N, r_1) \\
\text{ADDC}(31, 12, 1)
\]

Substitute symbols with their values

Expand macro

\[
110000 \ 00001 \ 1111 \ 00000000 \ 00001100
\]

- No “type checking” if you use the wrong opcode...

\[
\text{ADDC}(r_{31}, r_{12}, r_1) \\
\text{ADDC}(31, 12, 1)
\]

\[\text{Reg}[1] \leftarrow \text{Reg}[31] + 12\]

\[
\text{ADD}(r_{31}, N, r_1) \\
\text{ADD}(31, 12, 1)
\]

\[\text{Reg}[1] \leftarrow \text{Reg}[31] + \text{Reg}[12]\]
Labels and Offsets

**Input file**

\[ N = 12 \]
\[ \text{ADDC}(r31, N, r1) \]
\[ \text{ADDC}(r31, 1, r0) \]
\[ \text{loop: MUL}(r0, r1, r0) \]
\[ \text{SUBC}(r1, 1, r1) \]
\[ \text{BNE}(r1, \text{loop}, r31) \]

- **Label** value is the address of a memory location
- **BEQ/BNE** compute offset automatically
- Labels hide addresses!

**Symbol table**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>r31</td>
<td>31</td>
</tr>
<tr>
<td>N</td>
<td>12</td>
</tr>
<tr>
<td>loop</td>
<td>8</td>
</tr>
</tbody>
</table>

**Output file**

\[ 110000 \ 00001 \ 11111 \ 00000000 \ 00001100 \ [0x00] \]
\[ 110000 \ 00000 \ 11111 \ 00000000 \ 00000001 \ [0x04] \]
\[ 100010 \ 00000 \ 00001 \ 0000 \ 00000000000 \ [0x08] \]
\[ 110001 \ 00001 \ 00001 \ 00000000 \ 00000001 \ [0x0C] \]
\[ 011101 \ 11111 \ 00001 \ 11111111 \ 11111101 \ [0x10] \]

\[ \text{offset} = (\text{label} - \text{<addr of BNE/BEQ>})/4 - 1 \]
\[ = (8 - 16)/4 - 1 = -3 \]
### Pseudoinstructions

- Convenience instructions that expand into real instructions
- Make assembly language programs easier to read

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Real instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD(CC,RC)</td>
<td>LD(R31,CC,RC)</td>
</tr>
<tr>
<td>ST(RA,CC)</td>
<td>ST(RA,CC,R31)</td>
</tr>
<tr>
<td>CMOVE(CC,RC) // Reg[RC] &lt;- CC</td>
<td>ADDC(R31,CC,RC)</td>
</tr>
<tr>
<td>NOP(). // do nothing</td>
<td>ADD(R31,R31,R31)</td>
</tr>
<tr>
<td>BEQ(RA, LABEL)</td>
<td>BEQ(RA, LABEL, R31)</td>
</tr>
<tr>
<td>BNE(RA, LABEL)</td>
<td>BNE(RA, LABEL, R31)</td>
</tr>
<tr>
<td>BR(LABEL) // always branch</td>
<td>BEQ(R31, LABEL)</td>
</tr>
<tr>
<td>BR(LABEL,RC) // always branch</td>
<td>BEQ(R31, LABEL, RC)</td>
</tr>
<tr>
<td>BF(RA, LABEL,RC) // 0 is false</td>
<td>BEQ(RA, LABEL, RC)</td>
</tr>
<tr>
<td>BF(RA, LABEL)</td>
<td>BEQ(RA, LABEL)</td>
</tr>
<tr>
<td>BT(RA, LABEL,RC) // 1 is true</td>
<td>BNE(RA, LABEL, RC)</td>
</tr>
<tr>
<td>BT(RA, LABEL)</td>
<td>BNE(RA, LABEL)</td>
</tr>
</tbody>
</table>
Factorial with Pseudoinstructions

**Before**

\[
\begin{align*}
N &= 12 \\
\text{ADDC}(r31, N, r1) \\
\text{ADDC}(r31, 1, r0) \\
\text{loop: } &\text{MUL}(r0, r1, r0) \\
&\text{SUBC}(r1, 1, r1) \\
&\text{BNE}(r1, \text{loop}, r31)
\end{align*}
\]

**After**

\[
\begin{align*}
N &= 12 \\
&\text{CMOVE}(N, r1) \\
&\text{CMOVE}(1, r0) \\
\text{loop: } &\text{MUL}(r0, r1, r0) \\
&\text{SUBC}(r1, 1, r1) \\
&\text{BNE}(r1, \text{loop})
\end{align*}
\]
Raw Data

• LONG assembles a 32-bit value
  – Variables
  – Constants > 16 bits

N: \text{LONG}(12)

\text{factN:LONG}(0x\text{deadbeef})

Start:

LD(N, r1)
CMOVE(1, r0)

\text{loop: MUL}(r0, r1, r0)
SUBC(r1, 1, r1)
BT(r1, loop)
ST(r0, \text{factN})

Symbol table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>factN</td>
<td>4</td>
</tr>
</tbody>
</table>

\text{LD}(r31, N, r1)
\text{LD}(31, 0, 1)
Reg[1] ← Mem[Reg[31] + 0]
   ← Mem[0]
   ← 12
UASM Expressions and Layout

• Values can be written as expressions
  – Assembler evaluates expressions, they are not translated to instructions to compute the value!
    
    \[ A = 7 + 3 \times 0x0cc41 \]
    \[ B = A - 3 \]

• The “.” (period) symbol means the next byte address to be filled
  – Can read or write to it
  – Useful to control data layout or leave empty space (e.g., for arrays)
    
    \[ . = 0x100 \]  // Assemble into 0x100
    \[ \text{LONG}(0x\text{deadbeef}) \]
    \[ k = . \]  // Symbol “k” has value 0x104
    \[ \text{LONG}(0x00\text{dec0de}) \]
    \[ . = .+16 \]  // Skip 16 bytes
    \[ \text{LONG}(0xc0\text{ffeeee}) \]
Summary: Assembly Language

• Low-level language, symbolic representation of sequence of bytes. Abstracts:
  – Bit-level representation of instructions
  – Addresses
• Elements: Values, symbols, labels, instructions
• Values can be constants or expressions
• Symbols are symbolic representations of values
• Labels are symbols for addresses
• Can control where to assemble with “.” symbol
Programming Languages

32-bit (4-byte) ADD instruction:

\[
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

opcode    rc    ra    rb    (unused)


We’d rather write

\[
ADD(R2, R3, R4) \quad (Assembly)
\]

or better yet

\[
a = b + c; \quad (High-Level\ Language)
\]
High-Level Languages

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

```c
/* Compute greatest common divisor * using Euclid's method */
int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```

• 6.004 uses C, a common systems programming language. Modern popular alternatives include C++, Java, Python, and many others

• Advantages over assembly
  – Productivity (concise, readable, maintainable)
  – Correctness (type checking, etc)
  – Portability (run same program on different hardware)

• Disadvantages over assembly?
  – Efficiency?
Compilers

• Bare minimum for a functional compiler:

  Correct input source program (e.g., C) \rightarrow \text{Compiler} \rightarrow \text{Functionally equivalent target program (e.g., ASM)}

• Good compilers:
  – Produce meaningful errors on incorrect programs
    • Even better: meaningful warnings
  – Produce fast, optimized code

• This lecture:
  – Simple techniques to compile a C programs into assembly
A Simple Compilation Strategy

• Programs are sequences of statements, so repeatedly call `compile_statement(statement)`:
  – Unconditional: `expr;`
  – Compound: `{ statement_1; statement_2; ... }`
  – Conditional: `if (expr) statement_1; else statement_2;`
  – Iteration: `while (expr) statement;`

• Also need `compile_expr(expr)` to generate code to compute value of `expr` into a register
  – Constants: `1234`
  – Variables: `a, b[expr]`
  – Assignment: `a = expr`
  – Operations: `expr_1 + expr_2, ...`
  – Procedure calls: `proc(expr, ...)`

We’ll cover this in L11.
compile_expr(expr) \Rightarrow Rx

- **Constants**: 1234 \Rightarrow Rx
  - CMOVE(1234, Rx)
  - LD(c1, Rx)
    ... 
    c1: LONG(123456)

- **Variables**: a \Rightarrow Rx
  - LD(a, Rx)
    ...
    a: LONG(0)

- **Assignment**: a = expr \Rightarrow Rx
  - compile_expr(expr) \Rightarrow Rx
    ST(Rx, a)

- **Variables**: b[expr] \Rightarrow Rx
  - compile_expr(expr) \Rightarrow Rx
    MULC(Rx, bsize, Rx)
    LD(Rx, b, Rx)
    ...
    // reserve array space
    b: . = . + bsize * blen

- **Operations**:
  \( expr_1 + expr_2 \Rightarrow Rx \)
  - compile_expr(expr_1) \Rightarrow Rx
    compile_expr(expr_2) \Rightarrow Ry
    ADD(Rx, Ry, Rx)
Compiling Expressions

C code:

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

Beta assembly code:

```assembly
x:  LONG(0)
y:  LONG(0)
c1: LONG(123456)
...
LD(x, r1)
CMOVE(3, r2)
SUB(r1, r2, r1)  \(\Rightarrow\) SUBC(r1, 3, r1)
LD(y, r2)
LD(c1, r3)
ADD(r2, r3, r2)
MUL(r2, r1, r1)
ST(r1, y)
```

```assembly
 compile_expr(y = (x-3)*(y+123456))
 compile_expr((x-3)*(y+123456))
 compile_expr(x-3)
 compile_expr(x)
   LD(x, r1)
 compile_expr(3)
   CMOVE(3, r2)
   SUB(r1, r2, r1)
 compile_expr(y+123456)
 compile_expr(y)
   LD(y, r2)
 compile_expr(123456)
   LD(c1, r3)
   ADD(r2, r3, r2)
   MUL(r1, r2, r1)
   ST(r1, y)
```
compile_statement

• Unconditional: \textit{expr};

  Beta assembly:
  \texttt{compile\_expr(expr)}

• Compound: \{ \textit{statement}_1; \textit{statement}_2; \ldots \}

  Beta assembly:
  \texttt{compile\_statement(statement}_1)\texttt{compile\_statement(statement}_2)\ldots
**compile_statement: Conditional**

**C code:**
```c
if (expr)
    statement;
```

**Beta assembly:**
```assembly
compile_expr(expr) ⇒ Rx
BF(rx, Lendif)
compile_statement(statement)
```

**Lendif:**

**C code:**
```c
if (expr)
    statement_1;
else
    statement_2;
```

**Beta assembly:**
```assembly
compile_expr(expr) ⇒ Rx
BF(rx, Lelse)
compile_statement(statement_1)
BR(Lendif)
```

**Lelse:**
```assembly
compile_statement(statement_2)
```

**Lendif:**
compile_statement: Iteration

C code:
while (expr)
     statement;

Beta assembly:
Lwhile:
    compile_expr(expr)⇒Rx
    BF(rx, Lendwhile)
    compile_statement(statement)
    BR(Lwhile)
Lendwhile:

Better Beta assembly:
    BR(Ltest)
    Lwhile:
    compile_expr(expr)⇒Rx
    BF(rx, Lwhile)
    compile_statement(statement)
    BR(Lwhile)
    BT(rx, Lwhile)

Saves an instruction each iteration

C code:
for (init; test; increment)
     statement;

Example:
for (i=0; i < 10; i = i + 1)
    sum = sum + b[i];

is equivalent to:
init;
while (test) {
    statement;
    increment;
}
Putting It All Together: Factorial

```plaintext
int n = 20;
int r = 0;

start:
    CMOVE(1, r0)
    ST(r0, r)
    BR(test)

r = 1;

while (n > 0) {
    loop:
        LD(r, r3)
        LD(n, r1)
        MUL(r1, r3, r3)
        ST(r3, r)
        LD(n, r1)
        SUBC(r1, 1, r1)
        ST(r1, n)

    r = r*n;
    MUL(r1, r3, r3)
    ST(r3, r)
    LD(n, r1)
    SUBC(r1, 1, r1)
    ST(r1, n)

    n = n-1;
    LD(n, r1)
    CMPLT(r31, r1, r2)
    BT(r2, loop)
}

done:
```

Easy translation

Slow code

(10 instructions in the loop)
int n = 20, r;

r = 1;

while (n > 0) {
    r = r*n;
    n = n - 1;
}

Optimization: keep values in regs

Optimization: Keep n, r in registers ⇒ move LDs/STs out of loop!

4 instructions in the loop

Optimization:

4 instructions in the loop