Lecture A2: X-TOY Programming

What We've Learned About X-TOY

- X-TOY: what's in it, how to use it.
  - Box with switches and lights.
  - 4360 bits = 256 × 16 + 16 × 16 + 8.
  - von Neumann architecture.

- Data representation.
  - Binary and hex.

- X-TOY instruction set architecture.
  - 16 instruction types.

- Sample X-TOY machine language programs.
  - Arithmetic.
  - Loops.

What We Do Today

- Manipulate addresses.
  - Arrays.
  - Function calls.
  - Pointers.

- Represent data other than positive integers.
  - Negative numbers.

- Standard input, standard output.

- Bitwise operations.
  - XOR, AND.

Ballistic Tables

- Store table of f(x) = -x^2 + bx + c for various values of x.
  - Note: can't afford to multiply.
  - Use "differencing method."

\[ f(x) = -x^2 + 6x + 7 \]

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
<th>Δf(x)</th>
<th>Δ²f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>5</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>3</td>
<td>-2</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>-3</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>-5</td>
<td>-2</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>-7</td>
<td>-2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory Indirection

Static addressing.
- Until now, all load/store addresses hardwired in instruction.
  - Ex. 8A34: R[A] ← mem[34]
- More flexibility needed to implement arrays.

Indirect (dynamic) addressing.
- Work with names of data.
- Want to access variable memory location like $x$, instead of hardwiring 34.

Solution.
- Put memory address in register. (C "pointer")
- Use CONTENTS of register as address.
- Use store indirect, load indirect instructions to access.

Representing Other Primitive Data Types

Negative integers.
- X-TOY uses two's complement integers.
- Done in precept.

Big integers.
- Can use "multiple precision."
  - Use two 16-bit words per integer.

Real numbers.
- Can use "floating point" (like scientific notation).
- Double word for extra precision.

Characters.
- Can use ASCII code (7 bits / character).
- Can pack two characters into one 16-bit word.

Ballistic

```
int t = 2;
int y = c, x = 0;
int delta = b-1;
while (y > 0) {
  y += delta;
  delta += t;
  a[x] = y;
  x++;
}
```

```
ballistic.toy
```

```
ballistic.c
```

```
int t = -\Delta^2 f(x)
```

```
delta = \Delta f(x)
```

```
ballistic.toy
```

```
int t = 2;
int y = c, x = 0;
int delta = b-1;
while (y > 0) {
  y += delta;
  delta += t;
  a[x] = y;
  x++;
}
```

```
ballistic.c
```

```
t = -\Delta^2 f(x)
```

```
delta = \Delta f(x)
```

```
Ballistic
```

```
0B: 0006 b
0C: 0007 c
10: 7101 R[1] ← 0001
12: 7A20 R[A] ← 0020 a[]
13: 7300 R[3] ← 0000 x ← 0
14: 840B R[4] ← mem[0B]
16: 850C R[5] ← mem[0C] y ← c
17: B50A mem[R[A]] ← R[5] a[0] ← f(0)
1C: B507 mem[R[7]] ← R[5] a[x] ← y
1D: D518 if (R[5] > 0) goto 18
1E: 0000 halt
```

Standard Input, Standard Output

Standard input.
- Loading from memory address FF loads one (hexadecimal) integer from X-TOY stdin.
  - 8AFF means:
    - read an integer from standard input, and store it in register A
    - scanf("%hX", &a);
- 8AFF means: read R[A]
- 9AFF means: write R[A]
- 0000 means: halt

Standard output.
- Writing to memory location FF.
**Standard Input, Standard Output**

Enables computer to process more information than fits in memory.

- Arbitrary amounts of input.
  - dragon curve

```c
int x, max = 0;
while (scanf("%d", &x) != EOF) {
    if (x > max)
        max = x;
}
printf("Maximum = %d\n", max);
```

- Arbitrary amounts of output.

**Bitwise Operations**

**Bitwise AND.** (opcode 3)

**Bitwise XOR.** (opcode 4)

```c
void dragon(int m) {
    int k;
    F();
    for (k = 1; k <= m-1; k++) {
        if (k & ((k^(k-1))+1))
            R();
        else
            L();
        F();
    }
}
```

**Bitwise AND**

Logic operations are BITWISE:

- `1234_{16} & FAD_{216} = 1210_{16}`

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bitwise XOR**

Logic operations are BITWISE:

- `1234_{16} ^ FAD_{216} = E8E6_{16}`

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Dragon in XTOY

Impress your Yale friends with these 10 lines of code!
- Connect stdout to turtle.
  - L if zero
  - R if nonzero
- Keeps going until turtle runs out of energy.

```
10: 7101 R[1] <- 0001
11: 7201 R[2] <- 0001 k
15: 95FF write R[5]
17: 0000 halt
```

Standard Input, Standard Output: Booting

Booting.
- Work all day to develop operating system.
- How do you save it for tomorrow?
  - leave computer on?
    - Tubes would blow up.
  - write short program dump.toy to dump contents of memory onto tape
  - run dump.toy
- How do you get it back?
  - turn on computer, old memory values gone
  - key in short program boot.toy to read contents of memory from tape
  - run boot.toy
  - use operating system

```
dump.toy
F0: 7101 R[1] <- 0001
F1: 72EF R[2] <- 00EF
F3: 93FF write R[3]
F5: D2F2 if (R[2] > 0) goto F2
```

Function Call: A Failed Attempt

Goal: \( x \times y \times z \).
- Need two multiplications: \( x \times y \), \((x \times y) \times z\).
  - Solution 1: write multiply code 2 times.
  - Solution 2: write an X-TOY function.

A failed attempt:
- Write multiply loop at 30-36.
- Calling program agrees to store arguments in register A and B.
- Function agrees to leave result in register C.
- Call function with jump absolute to 30.
- Return from function with jump absolute.

Reason for failure.
- Need to return to a VARIABLE memory address.

```
function?
10: 8AFF
11: 8BFF
12: C0E0
13: 1AC0
14: 8BFF
15: C0E0
16: 9CFF
17: 0000
30: 7101
31: CA36
32: 1CCB
33: 2AA1
34: C031
35: C032
```

Multiplication Function

Calling convention.
- Jump to line 30.
- Store a and b in registers A and B.
- Return address in register F.
- Put result \( c = a \times b \) in register C.
- Register 1 is scratch.
- Overwrites registers A and B.

```
function
10: 8AFF
11: 8BFF
12: FF00
13: 1AC0
14: 8BFF
15: FF00
16: 9CFF
17: 0000
30: 7C00
31: 7101
32: CA36
33: 1CCB
34: 2AA1
35: C032
36: EF00
```

```
function.toy
30: 7C00 R[C] <- 00
31: 7101 R[1] <- 01
32: CA36 if (R[A] == 0) goto 36
33: 1CCB R[C] += R[B]
34: 2AA1 R[A]--
35: C032 goto 32
36: EF00 pc <- R[F]
return
```

opcode E
jump register
Multiplication Function Call

Client program to compute \( x \times y \times z \).
- Read \( x, y, z \) from standard input.
- Note: PC is incremented before instruction is executed.
- value stored in register F is correct return address

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: 82FF</td>
<td>read R[2] ( x )</td>
</tr>
<tr>
<td>11: 83FF</td>
<td>read R[3] ( y )</td>
</tr>
<tr>
<td>12: 84FF</td>
<td>read R[4] ( z )</td>
</tr>
<tr>
<td>13: 1A20</td>
<td>( R[A] \leftarrow R[2] \times x )</td>
</tr>
<tr>
<td>14: 1B30</td>
<td>( R[B] \leftarrow R[3] \times y )</td>
</tr>
<tr>
<td>15: FF30</td>
<td>( R[F] \leftarrow \text{pc}; \text{goto 30} (x \times y) \times z )</td>
</tr>
<tr>
<td>16: 1AC0</td>
<td>( R[A] \leftarrow R[C] \times (x \times y) \times z )</td>
</tr>
<tr>
<td>17: 1B40</td>
<td>( R[B] \leftarrow R[4] \times z )</td>
</tr>
<tr>
<td>18: FF30</td>
<td>( R[F] \leftarrow \text{pc}; \text{goto 30} \times (x \times y) \times z )</td>
</tr>
<tr>
<td>19: 9CFF</td>
<td>write R[C]</td>
</tr>
<tr>
<td>1A: 0000</td>
<td>halt</td>
</tr>
</tbody>
</table>

Function Call: One Solution

Contract between calling program and function:
- Calling program stores function parameters in specific registers.
- Calling program stores return address in a specific register.
- Calling program sets PC to address of function.
- Function stores return value in specific register.
- Function sets PC to return address when finished.
- jump register

What if you want a function to call another function?
- Use a different register for return address.
- More general: store return addresses on a stack.

An Efficient Multiplication Algorithm

Inefficient multiply.
- Load in integers \( a \) and \( b \), and store \( c = a \times b \).
- Brute-force algorithm:
  - initialize \( c = 0 \)
  - add \( b \) to \( c \), \( a \) times

"Grade-school" binary multiplication.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>* 1 5 1 2</td>
<td>* 1 1 0 1</td>
</tr>
<tr>
<td>2 4 6 8</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>6 1 7 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>0 1 8 6 5 8 0 8</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 0 0 0 1 1 1</td>
<td>1 0 0 0 1 1 1</td>
</tr>
</tbody>
</table>

Binary Multiplication

Grade school binary multiplication algorithm.
- Initialize \( c = 0 \).
- Loop over \( i \) bits of \( b \):
  - if \( b_i = 0 \), do nothing
  - if \( b_i = 1 \), shift a left \( i \) bits and add to \( c \)

Implement with built-in TOY shift instructions.

```
int c = 0;
for (i = 15; i >= 0; i--) {
    if ((b >> i) & 1) {
        c += (a << i);
    }
}
```
### Shift Left

**Shift left. (opcode 5)**
- Move bits to the left, padding with zeros as needed.
- \(1_{16}2_{16}3_{16}4_{16} \ll 7_{16} = 1600_{16}\)

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
1_{16} & 2_{16} & 3_{16} & 4_{16}
\end{array}
\]

\[\ll 7 = \text{pad with 0's}\]

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
1_{16} & 6_{16} & 0_{16} & 0_{16}
\end{array}
\]

### Shift Right

**Shift left. (opcode 6)**
- Move bits to the right, padding with sign bit as needed.
- \(1_{16}2_{16}3_{16}4_{16} \gg 7_{16} = 0124_{16}\)

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
1_{16} & 2_{16} & 3_{16} & 4_{16}
\end{array}
\]

\[\gg 7 = \text{discard sign bit}\]

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0_{16} & 1_{16} & 2_{16} & 4_{16}
\end{array}
\]

### Fast Multiplication Function

**multiply-fast.toy**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30:</td>
<td>7101 R[1] &lt;- 0001</td>
<td>i = 16 result</td>
</tr>
<tr>
<td>31:</td>
<td>7210 R[2] &lt;- 0010</td>
<td>i = 16 result</td>
</tr>
<tr>
<td>32:</td>
<td>7C00 R[C] &lt;- 0000</td>
<td>i = 16 result</td>
</tr>
<tr>
<td>33:</td>
<td>C23B if (R[2] == 0) goto 3B</td>
<td>while (i &gt; 0) {</td>
</tr>
<tr>
<td>34:</td>
<td>2221 R[2]--</td>
<td>i--</td>
</tr>
<tr>
<td>38:</td>
<td>C43A if (R[4] == 0) goto 3A</td>
<td>add to result</td>
</tr>
<tr>
<td>39:</td>
<td>1CC3 R[C] += R[3]</td>
<td></td>
</tr>
<tr>
<td>3A:</td>
<td>C033 goto 33</td>
<td>}</td>
</tr>
<tr>
<td>3B:</td>
<td>EF00 goto R[F]</td>
<td>return</td>
</tr>
</tbody>
</table>

### Lecture A2: Extra Slides
### Two's Complement Integers

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>+32767</td>
<td>7FFF</td>
<td>0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

**Properties:**
- Leading bit (bit 15) signifies sign.
- Negative integer \(-N\) represented by \(2^{16} - N\).
- Trick to compute \(-N\):
  1. Start with \(N\).
  2. Flip bits.
  3. Add 1.

- \(N = 0000000000000000\) represents 0.
- \(-0\) and +0 are the same.
- Addition is easy (see next slide).
- Checking for arithmetic overflow is easy.

### Two's Complement Arithmetic

Addition is carried out as if all integers were positive.
- It usually works:

\[
\begin{align*}
-3 & \quad 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 \\
+ & \quad 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 \\
= & \quad 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
\end{align*}
\]
**Two’s Complement Arithmetic**

Addition is carried out as if all integers were positive.
- It usually works.
- But overflow / underflow can occur:
  - Carry into sign (left most) bit with no carry out.

```
+32,767  0 1 1 1 1 1 1 1 1 1 1 1
+  
2  0 0 0 0 0 0 0 0 0 0 0
=  
-32,767  1 0 0 0 0 0 0 0 0 0 0
```

**Shift Right**

Shift right. (opcode 6)
- Move bits to the right, padding with sign bit as needed.
  - $\text{FFCA}_{16} \gg 2_{16} = \text{FFF2}_{16}$
  - $-53_{10} \gg 2_{10} = -13_{10}$

```
<table>
<thead>
<tr>
<th>F_{16}</th>
<th>F_{16}</th>
<th>C_{16}</th>
<th>A_{16}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 0</td>
<td>0 1 0 1</td>
</tr>
</tbody>
</table>
```

- **Discard**
- **Pad with 1’s**

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

**Useful X-TOY Idioms**

**Jump absolute.**
- Jump to a fixed memory address.
  - branch if zero with destination
  - register 0

```
17: C014  pc ← 14
```

**Register assignment.**
- No instruction that transfers contents of one register into another.
- Pseudo-instruction that simulates assignment:
  - add with register 0 as one of two source registers

```
```

**No-op.**
- Instruction that does nothing.
- Plays the role of whitespace in C programs.
  - numerous other possibilities!

```
17: 1000  no-op
```

**Other Logical Operations**

Any logical operation can be implemented with AND and XOR.
- See Boolean circuit lecture.

**Build OR from AND and XOR.**
- $(x \& y) \text{ OR } (x \oplus y)$

```
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x &amp; y</th>
<th>x \oplus y</th>
<th>a &amp; b</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Build NOT from XOR.**
- $1 \oplus x = x'$
- $\text{FFFF} \oplus x = x'$ (bitwise NOT)

```
<table>
<thead>
<tr>
<th>x</th>
<th>x \oplus 1</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```