Lecture A2: X-TOY Programming and Simulation

Useful X-TOY Idioms

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

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!

How to represent negative integers

X-TOY integers occupy 16 bits each.
- We can represent 0 to $2^{16} - 1$ if we like.
- But what about negative integers?
- Reserving half the possible bit-patterns for negative seems fair.

Highly desirable property:
- If X is a positive integer, then the representation of -X, when added to X, had better yield zero.

Two's Complement Integers

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.
Two's Complement Integers

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>FFF</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>14</td>
<td>0004</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>13</td>
<td>0003</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>12</td>
<td>0002</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>11</td>
<td>0001</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>10</td>
<td>0000</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>9</td>
<td>FFF</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>8</td>
<td>FFFE</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>7</td>
<td>FFFD</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>6</td>
<td>FFFC</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

**Properties of Two's Complement Integers**

**Nice properties:**
- $0000000000000000$ represents 0.
- -0 and +0 are the same.
- Addition is easy (see next slide).
- Checking for arithmetic overflow is easy.

**Not-so-nice properties.**
- Can represent one more negative integer than positive integer ($-32,768 = -2^{15}$ but not $32,768 = 2^{15}$).

Two's Complement Arithmetic

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

```
-3 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
+  4 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
  = 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

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

```
+32,767 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
+    2 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
  = -32,767 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```
Representing Other Primitive Data Types

Negative integers.
- X-TOY uses two's complement integers.

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 (8 bits / character).
- Can pack two characters into one 16-bit word.

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.

Array3

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);

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.

```c
max.c

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

- Arbitrary amounts of output.
  - dragon curve

Bitwise Operations

Bitwise AND. (opcode 3)

Bitwise XOR. (opcode 4)

```c
dragon-nonrecursive.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();
}
```

Logic operations are BITWISE:
- $1234_{16} \& FAD_{16} = 1210_{16}$
- $1234_{16} \text{ XOR } FAD_{16} = E8E_{16}$

<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>
**Dragon in XTOY**

Impress your Yale friends with these 10 lines of code!

- Connect stdout to turtle.
  - `Li` if zero
  - `R` if nonzero
- Keeps going until turtle runs out of energy.

---

**Function Call: A Failed Attempt**

**Goal:** \( x \times y \times z \).
- Need two multiplications: \( x \times y, (x \times y) \times z \).

**A failed attempt:**
- Write multiply loop at 30-36.
- Calling program agrees to store arguments in registers 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.**

---

**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.

---

**Multiplication Function Call**

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

---
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.
  - jump-and-link
- 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?

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" multiplication.

Grade school binary multiplication algorithm to compute \( c = a \times b \).
- 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.

Binary Multiplication

Shift left. (opcode 5)
- Move bits to the left, padding with zeros as needed.
- \( \text{1234}_{16} \ll 7_{16} = \text{1600}_{16} \)
Shift Right

Move bits to the right, padding with sign bit as needed.

1234_{16} \gg 7_{16} = 0124_{16}

How To Build a TOY Machine

Implement fetch-execute cycle in hardware.
- See Lecture A4-A6.

Implement fetch-execute cycle in software.
- Write a program to "simulate" the behavior of the TOY machine.
  - TOY simulator in Java.
  - TOY simulator in C.
  - TOY simulator in TOY! (It has been done.)

Fast Multiplication Function

```
int main(int argc, char *argv[]) {
  short R[16] = {0}; // registers
  short mem[256] = {0}; // memory
  unsigned char pc = 0x10; // pc
  int i, op, d, s, t;
  FILE *file;
  file = fopen(argv[1], "r");
  if (file == NULL) {
    printf("Error opening %s\n", file);
    exit(EXIT_FAILURE);
  }
  ...
  ...
while (fscanf(file, ">%dX%4hX", &i, &inst) != EOF)
  mem[i] = inst;
  ...
  ...
}```
do {
    inst = mem[pc++];
    op = (inst >> 12) & 15;
    d = (inst >> 8) & 15;
    s = (inst >> 4) & 15;
    t = (inst >> 0) & 15;
    addr = (inst >> 0) & 255;
    // see next slide
    
    R[0] = 0;
} while (op != 0);

return 0;

// X-TOY Simulator

**Shifting and Masking**

Extract destination register.

- Given 16 bit integer in C, isolate destination register (bits 8-11).
- Use bit operations in C.

\[
\begin{array}{ccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
\]

\[
\text{inst} = 3604_{16}
\]

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

\[
(\text{inst} >> 8) & 15 = 6
\]

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

**TOY Simulator**

**toy.c: execute instruction**

```c
switch (op) {
    case 0: break;
    case 1: R[d] = R[s] + R[t]; break;
    case 2: R[d] = R[s] - R[t]; break;
    case 3: R[d] = R[s] & R[t]; break;
    case 4: R[d] = R[s] ^ R[t]; break;
    case 5: R[d] = R[s] << R[t]; break;
    case 6: R[d] = R[s] >> R[t]; break;
    case 7: R[d] = addr; break;
    case 8: R[d] = mem[addr]; break;
    case 9: mem[addr] = R[d]; break;
    case 10: R[d] = mem[R[t]]; break;
    case 12: if (R[d] == 0) pc = addr; break;
    case 13: if (R[d] > 0) pc = addr; break;
    case 14: pc = R[d]; break;
    case 15: R[d] = pc; pc = addr; break;
}
```

- **Stdin**: insert following code before execute switch statement.
  - if address is FF and opcode is load or load indirect
    ```c
    if ((addr == 255 && op == 8) || (R[t] == 255 && op == 10))
        scanf("%4hX", &mem[255]);
    ```

- **Stdout**: insert following code after execute switch statement.
  - if address is FF and opcode is store or store indirect
    ```c
    if ((addr == 255 && op == 9) || (R[t] == 255 && op == 11))
        printf("%04hX\n", mem[255]);
    ```
Simulation

Consequences of simulation.
- Test out new machine (or microprocessor) using simulator.
  - cheaper and faster than building actual machine
- Easy to add new functionality to simulator.
  - trace, single-step, breakpoint debugging
  - simulator more useful than TOY itself
- Reuse software for old machines.

Ancient programs still running on modern computers.
- Ticketron.
- Lode Runner on Apple IIe.

Basic Characteristics of X-TOY Machine

TOY is a general-purpose computer.
- Sufficient power to perform any computation.
- Limited only by amount of memory (and time).

Stored-program computer. (von Neumann memo, 1944)
- Data and instructions encoded in binary.
- Data and instructions stored in SAME memory.
- Can change program (control) without rewiring.
  - immediate applications
  - profound implications
- EDSAC (Wilkes 1949).
  - first stored-program computer
- Outgrowth of Turing’s work.

All modern computers are general-purpose computers and have same (von Neumann) architecture.

Lecture A2: Supplemental Notes

TOY Cheat Sheet

<table>
<thead>
<tr>
<th>#</th>
<th>Operation</th>
<th>Fmt</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>halt</td>
<td>1</td>
<td>$\text{exit}(0)$</td>
</tr>
<tr>
<td>1</td>
<td>add</td>
<td>1</td>
<td>$R[d] \leftarrow R[s] + R[t]$</td>
</tr>
<tr>
<td>2</td>
<td>subtract</td>
<td>1</td>
<td>$R[d] \leftarrow R[s] - R[t]$</td>
</tr>
<tr>
<td>3</td>
<td>and</td>
<td>1</td>
<td>$R[d] \leftarrow R[s] &amp; R[t]$</td>
</tr>
<tr>
<td>4</td>
<td>xor</td>
<td>1</td>
<td>$R[d] \leftarrow R[s] ^ R[t]$</td>
</tr>
<tr>
<td>5</td>
<td>shift left</td>
<td>1</td>
<td>$R[d] \leftarrow R[s] &lt;&lt; R[t]$</td>
</tr>
<tr>
<td>6</td>
<td>shift right</td>
<td>1</td>
<td>$R[d] \leftarrow R[s] &gt;&gt; R[t]$</td>
</tr>
<tr>
<td>7</td>
<td>load addr</td>
<td>2</td>
<td>$R[d] \leftarrow \text{addr}$</td>
</tr>
<tr>
<td>8</td>
<td>load</td>
<td>2</td>
<td>$R[d] \leftarrow \text{mem}[\text{addr}]$</td>
</tr>
<tr>
<td>9</td>
<td>store</td>
<td>2</td>
<td>$\text{mem}[\text{addr}] \leftarrow R[d]$</td>
</tr>
<tr>
<td>A</td>
<td>load indirect</td>
<td>1</td>
<td>$R[d] \leftarrow \text{mem}[R[t]]$</td>
</tr>
<tr>
<td>B</td>
<td>store indirect</td>
<td>1</td>
<td>$\text{mem}[R[t]] \leftarrow R[d]$</td>
</tr>
<tr>
<td>C</td>
<td>branch zero</td>
<td>2</td>
<td>$\text{if } (R[d] == 0) \text{ pc } \leftarrow \text{addr}$</td>
</tr>
<tr>
<td>D</td>
<td>branch positive</td>
<td>2</td>
<td>$\text{if } (R[d] &gt; 0) \text{ pc } \leftarrow \text{addr}$</td>
</tr>
<tr>
<td>E</td>
<td>jump register</td>
<td>2</td>
<td>$\text{pc } \leftarrow R[d]$</td>
</tr>
<tr>
<td>F</td>
<td>jump and link</td>
<td>2</td>
<td>$R[d] \leftarrow \text{pc}; \text{pc } \leftarrow \text{addr}$</td>
</tr>
</tbody>
</table>
Shift Right

Shift right. (opcode 6)
- Move bits to the right, padding with sign bit as needed.
- \( \text{FFCA}_{16} \gg 2_{16} = \text{FF2}_{16} \)
- \(-53_{10} \gg 2_{10} = -13_{10} \)

<table>
<thead>
<tr>
<th>sign bit</th>
<th>discard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 0 0 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>( F_{16} )</td>
<td>( F_{16} )</td>
</tr>
</tbody>
</table>

pad with 1’s

\( \gg 2 = \)

<table>
<thead>
<tr>
<th>pad with 1’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>( F_{16} )</td>
</tr>
</tbody>
</table>

Other Logical Operations

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

Build OR from AND and XOR.
- \((x \& y) \ xor (x \ xor y)\)

<table>
<thead>
<tr>
<th>a b</th>
<th>x y</th>
<th>x &amp; y</th>
<th>x ^ y</th>
<th>a ^ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>1 1</td>
<td>0 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

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