Lecture A2: X-TOY Programming

DEC PDP 12

What We've Learned About X-TOY

X-TOY: what's in it, how to use it.
- Box with switches and lights.
- 4360 bits = $256 \times 16 + 16 \times 16 + 8$.
- 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.

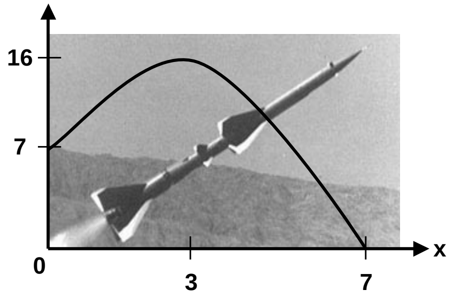
Bitwise operations.
- XOR, AND.

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."

$f(x) = -x^2 + 6x + 7$

x	f(x)	$\Delta f(x)$	$\Delta^2 f(x)$
0	7		
		5	
1	12		-2
		3	
2	15		-2
		1	
3	16		-2
		-1	
4	15		-2
		-3	
5	12		-2
		-5	
6	7		-2
		-7	
7	0		



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.

`t = -Δ²f(x)`

`delta = Δf(x)`

ballistic.c

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

Ballistic

ballistic.toy

```
0B: 0006                                b
0C: 0007                                c

10: 7101   R[1] ← 0001
11: 7202   R[2] ← 0002                  t ← 2
12: 7A20   R[A] ← 0020                  a[]
13: 7300   R[3] ← 0000                  x ← 0
14: 840B   R[4] ← mem[0B]
15: 2441   R[4] ← R[4] - R[1]           delta ← b-1
16: 850C   R[5] ← mem[0C]               y ← c
17: B50A   mem[R[A]] ← R[5]             a[0] ← f(0)

18: 1554   R[5] ← R[5] + R[4]           y += delta
19: 2442   R[4] ← R[4] - R[2]           delta += t
1A: 1331   R[3] ← R[3] + R[1]           x++
1B: 17A3   R[7] ← R[A] + R[3]           address of a[x]
1C: B507   mem[R[7]] ← R[5]             a[x] ← y
1D: D518   if (R[5] > 0) goto 18
1E: 0000   halt
```

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.

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);`

stdin-stdout.toy

```
10: 8AFF   read R[A]
11: 9AFF   write R[A]
12: 0000   halt
```

Standard output.

- Writing to memory location FF.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1
8_{16}				A_{16}				F_{16}				F_{16}			
opcode				dest d				addr							

Standard Input, Standard Output

Enables computer to process more information than fits in memory.

- Arbitrary amounts of input.

max.c

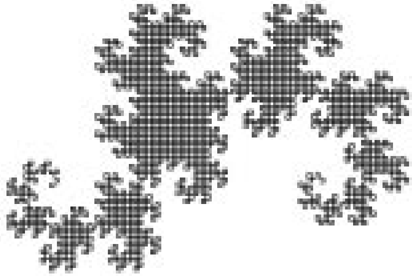
```
int x, max = 0;

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)



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();
   }
}
```

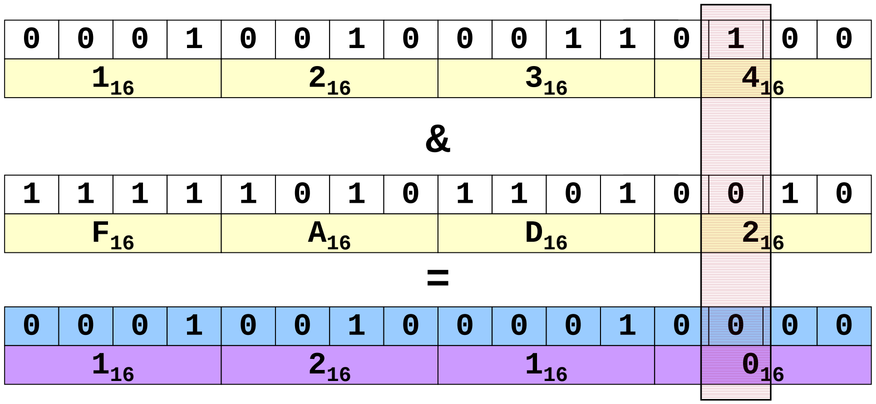
check bit to the left of rightmost 1

Bitwise AND

x	y	AND
0	0	0
0	1	0
1	0	0
1	1	1

Logic operations are BITWISE:

- 1234_{16} & $FAD2_{16}$ = 1210_{16}



0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
1_{16}				2_{16}				3_{16}				4_{16}			

&

1	1	1	1	1	0	1	0	1	1	0	1	0	0	1	0
F_{16}				A_{16}				D_{16}				2_{16}			

=

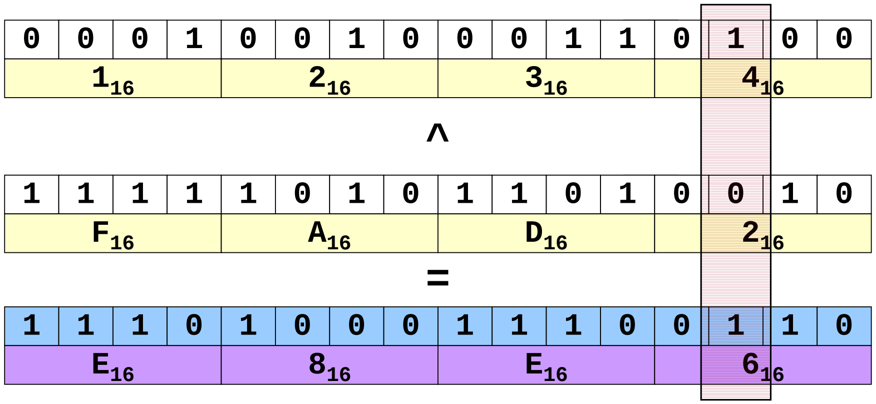
0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0
1_{16}				2_{16}				1_{16}				0_{16}			

Bitwise XOR

x	y	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Logic operations are BITWISE:

- 1234_{16} ^ $FAD2_{16}$ = $E8E6_{16}$



0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
1_{16}				2_{16}				3_{16}				4_{16}			

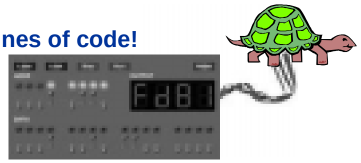
^

1	1	1	1	1	0	1	0	1	1	0	1	0	0	1	0
F_{16}				A_{16}				D_{16}				2_{16}			

=

1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	0
E_{16}				8_{16}				E_{16}				6_{16}			

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.

dragon.toy

```
10: 7101   R[1] <- 0001
11: 7201   R[2] <- 0001         k

12: 2321   R[3] <- R[2] - R[1]  k - 1
13: 4423   R[4] <- R[2] ^ R[3]  k ^ (k-1)
14: 1541   R[5] <- R[4] + R[1]  (k^(k-1))+1
15: 3552   R[5] <- R[5] & R[2]  k & ((k^(k-1))+1)
16: 95FF   write R[5]
17: 1221   R[2] <- R[2] + R[1]
18: C012   goto 12
19: 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`

dump.toy

```
F0: 7101   R[1] ← 0001
F1: 72EF   R[2] ← 00EF
F2: A302   R[3] ← mem[R[2]]
F3: 93FF   write R[3]
F4: 2221   R[2] ← R[2] - R[1]
F5: D2F2   if (R[2] > 0) goto F2
```

- 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

Function Call: A Failed Attempt

Goal: x × y × z.

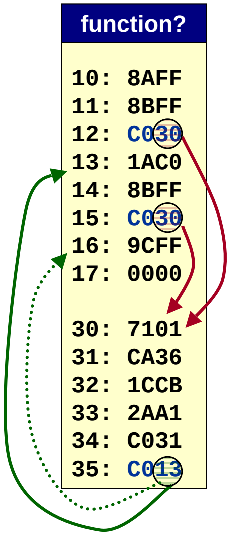
- Need two multiplications: x × y, (x × y) × z.
  - Solution 1: write multiply code 2 times.
  - Solution 2: write an X-TOY function.

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: C030
13: 1AC0
14: 8BFF
15: C030
16: 9CFF
17: 0000

30: 7101
31: CA36
32: 1CCB
33: 2AA1
34: C031
35: C013
```

Multiplication Function

Calling convention.

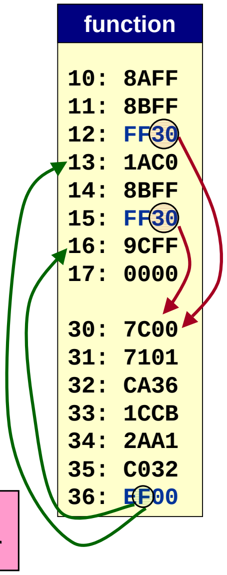
- Jump to line 30.
- Store a and b in registers A and B.
- Return address in register F.
- Put result c = a × b in register C.
- Register 1 is scratch.
- Overwrites registers A and B.

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



function

```
10: 8AFF
11: 8BFF
12: FF30
13: 1AC0
14: 8BFF
15: FF30
16: 9CFF
17: 0000

30: 7C00
31: 7101
32: CA36
33: 1CCB
34: 2AA1
35: C032
36: EF00
```

Multiplication Function Call

Client program to compute x × y × 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

function.toy (cont)

```
10: 82FF   read R[2]            x
11: 83FF   read R[3]            y
12: 84FF   read R[4]            z

13: 1A20   R[A] ← R[2]          x
14: 1B30   R[B] ← R[3]          y
15: FF30   R[F] ← pc; goto 30   x * y

16: 1AC0   R[A] ← R[C]          (x * y)
17: 1B40   R[B] ← R[4]          z
18: FF30   R[F] ← pc; goto 30   (x * y) * z

19: 9CFF   write R[C]
1A: 0000   halt
```

opcode F
jump and link

R[F] ← 16

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?

✎ 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 × b.
- Brute-force algorithm:
  - initialize c = 0
  - add b to c, a times

"Grade-school" binary multiplication.

Decimal

```
        1 2 3 4
      * 1 5 1 2
        2 4 6 8
      1 2 3 4
    6 1 7 0
  1 2 3 4
0 1 8 6 5 8 0 8
```

Binary

```
        1 0 1 1
      * 1 1 0 1
        1 0 1 1
      0 0 0 0
    1 0 1 1
  1 0 1 1
1 0 0 0 1 1 1 1
```

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

b_i = i^{th} bit of b

```
        1 0 1 1
      * 1 1 0 1
        1 0 1 1
      0 0 0 0
    1 0 1 1
  1 0 1 1
1 0 0 0 1 1 1 1
```

Implement with built-in TOY shift instructions.

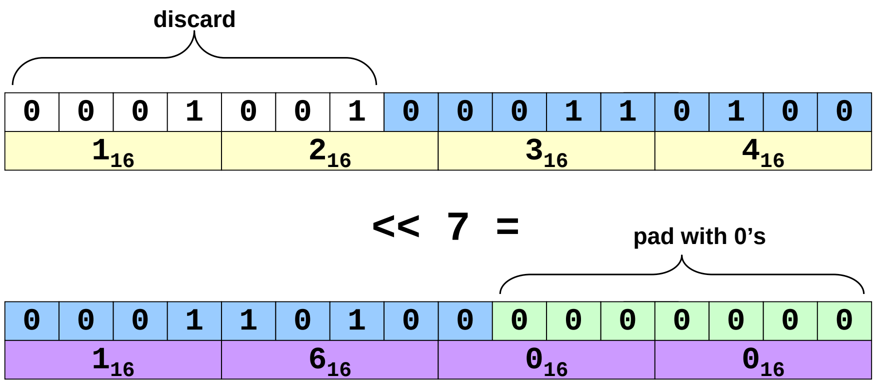
multiply-fast.c

```
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.
- `1234`$_{16}$ `<<` `7`$_{16}$ `=` `1600`$_{16}$



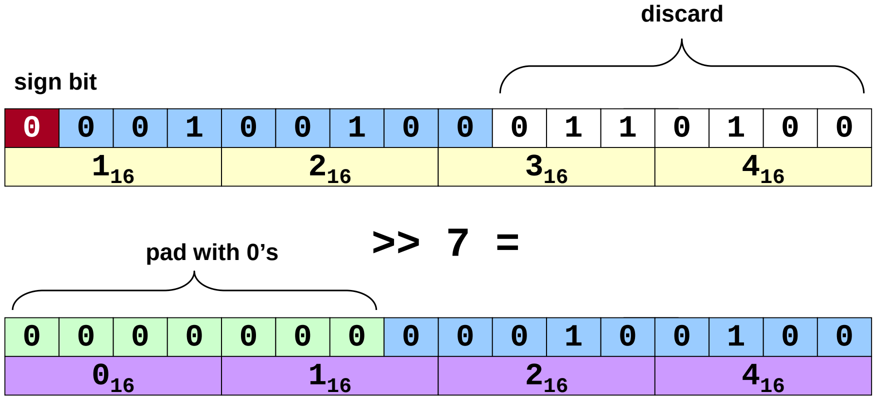
0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
1_{16}				2_{16}				3_{16}				4_{16}			

0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0
1_{16}				6_{16}				0_{16}				0_{16}			

Shift Right

Shift left. **(opcode 6)**

- Move bits to the right, padding with sign bit as needed.
- `1234`$_{16}$ `>>` `7`$_{16}$ `=` `0124`$_{16}$



0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
1_{16}				2_{16}				3_{16}				4_{16}			

0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
0_{16}				1_{16}				2_{16}				4_{16}			

Fast Multiplication Function

multiply-fast.toy

```
// Input:  R[A] contains a, R[B] contains b
// Output: R[C] contains a * b

30: 7101   R[1] <- 0001
31: 7210   R[2] <- 0010               i = 16
32: 7C00   R[C] <- 0000               result

33: C23B   if (R[2] == 0) goto 3B     while (i > 0) {
34: 2221   R[2]--                        i--
35: 53A2   R[3] ← R[A] << R[2]           a left shifted i
36: 64B2   R[4] ← R[B] >> R[2]
37: 3441   R[4] ← R[4] & R[1]            b_i = ith bit of b

38: C43A   if (R[4] == 0) goto 3A
39: 1CC3   R[C] += R[3]                  add to result

3A: C033   goto 33                    }
3B: EF00   goto R[F]                  return
```

16-bit integers

Lecture A2: Extra Slides

Two's Complement Integers

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dec	**Hex**	**Binary**															
+32767	7FFF	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		•••															
+4	0004	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
+3	0003	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
+2	0002	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
+1	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
+0	0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	FFFF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-2	FFFE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
-3	FFFD	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
-4	FFFC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
		•••															
-32768	8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

2. Flip bits.

	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

3. Add 1.

-4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

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.

- N = ~N + 1

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	1	0	1

+

4	0	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	0	1

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	1	1	1	1

+

2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

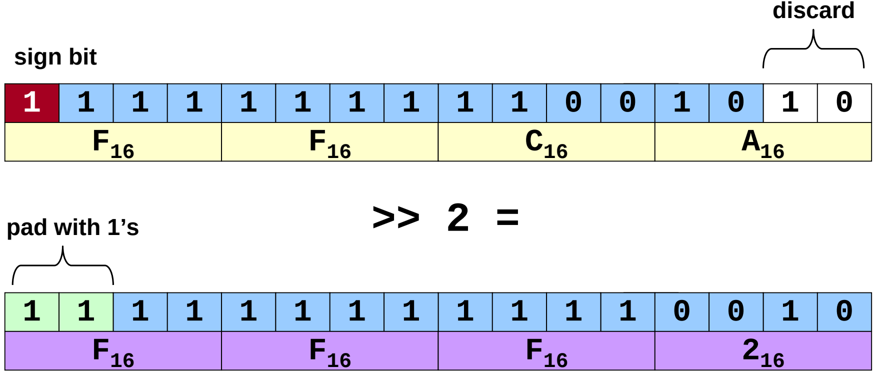
=

-32,767	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Shift Right

Shift right. (opcode 6)

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



Useful X-TOY Idioms

Jump absolute.

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

Jump absolute

```
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

Register assignment

```
17: 1230   R[2] ← R[3]
```

No-op.

- Instruction that does nothing.
- Plays the role of whitespace in C programs.
  - numerous other possibilities!

No-op

```
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) ^ (x ^ y)

		a	b		
x	y	x & y	x ^ y	a ^ b	OR
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	0	1	1

Build NOT from XOR.

- 1 ^ x = x'
- FFFF ^ x = x' (bitwise NOT)

x	1	x ^ 1	NOT
0	1	1	1
1	1	0	0