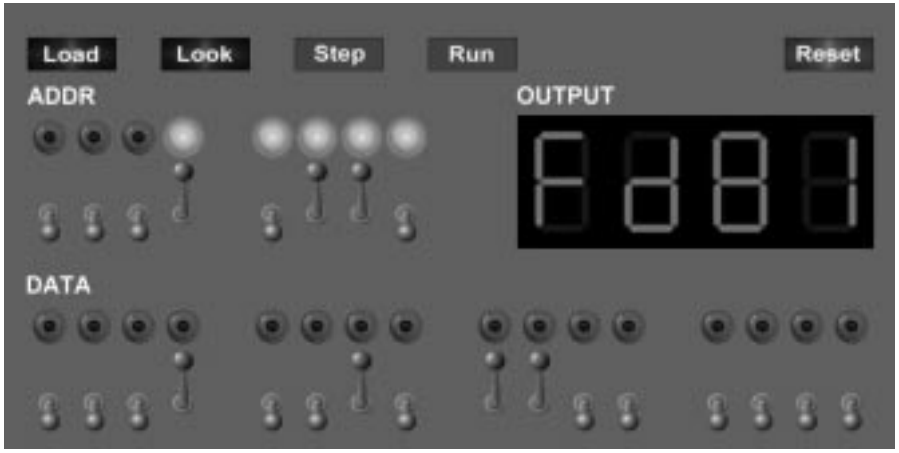
Lecture A3: TOY Simulator



Today

Relate TOY, C, and "real computers."

- von Neumann architecture.
- General purpose machine.

Programs that manipulate programs.

- Creating malicious programs.
- Bootstrapping.

Simulation.

- TOY simulator in C.
- TOY simulator in Java. (Visual X-TOY)
- TOY simulator in TOY.

Basic Characteristics of X-TOY Machine

X-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.

Unsafe Code at any Speed

Reverse.

- Read in a sequence of integers, and print them in reverse order.
- Why is this unsafe?
  - Buffer overflow: array `a[]` has fixed size which may be too small.

Might lose control of machine behavior . . .

reverse.c

```
n = 0;
while (scanf("%d", &a[n]) != EOF)
   n++;

while (n >= 0) {
   printf("%d\n", a[n]);
   n--;
}
```

TOY Implementation of Reverse

Read in a sequence of integers and print them in reverse order.

- Stop upon reading 0000.

reverse.toy

```
10: 7101  R[1] ← 0001
11: 7A30  R[A] ← 0030  // a[]
12: 7B00  R[B] ← 0000  // i

// read in integers
13: 8CFF  read R[C]
14: CC19  if (R[C] == 0)
             goto 19
15: 16AB  R[6] ← R[A] + R[B]
16: BC06  mem[R[6]] ← R[C]
17: 1BB1  R[B]++
18: C013  goto 13
```

reverse.toy (cont)

```
// print in reverse order
19: CB20  if (R[B] == 0)
             goto 20
1A: 16AB  R[6] ← R[A] + R[B]
1B: 2661  R[6]--
1C: AC06  R[C] ← mem[R[6]]
1D: 9CFF  write R[C]
1E: 2BB1  R[B]--
1F: C019  goto 19
20: 0000  halt
```


Food For Thought

What happens if we change `7A30` to `7A00`?

- Self modifying program.
- Malicious user could exploit buffer overflow and run any code!

Crazy 8s Input

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
8888 8810
98FF C011
```

reverse.toy

```
// malicious code
10: 8888  data
11: 8810  R[8] ← mem[10]
12: 98FF  print R[8]
13: C011  goto 11

14: CC19  if (R[C] == 0)
             goto 19
15: 16AB  R[6] ← R[A] + R[B]
16: BC06  mem[R[6]] ← R[C]
17: 1BB1  R[B]++
18: C013  goto 13
```

reverse.toy (cont)

```
// print in reverse order
19: CB20  if (R[B] == 0)
             goto 20
1A: 16AB  R[6] ← R[A] + R[B]
1B: 2661  R[6]--
1C: AC06  R[C] ← mem[R[6]]
1D: 9CFF  write R[C]
1E: 2BB1  R[B]--
1F: C019  goto 19
20: 0000  halt
```


What Can Happen When We Lose Control?

Robert Morris Internet Worm.

- Cornell grad student injected worm into Internet in 1988.
- Exploited buffer overflow hole in finger daemon `fingerd`.

buffer-overflow.c

```
int main(void) {
   char buffer[100];
   scanf("%s", buffer);
   printf("%s\n", buffer);
   return 0;
}
```

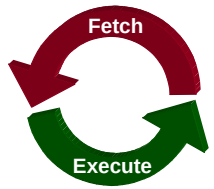

How To Build a TOY Machine

Implement fetch-execute cycle in hardware.

- See Lecture A4-A6.

X-TOY Machine

```
DO {
   fetch instruction
   execute instruction
} UNTIL (halt signal)
```



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 C

TOY SIMULATOR: toy.c

```
int main(int argc, char *argv[]) {
   short R[16] = {0};        // registers
   short mem[256] = {0};     // memory
   unsigned char pc = 0x10;  // pc
   unsigned int inst, addr;
   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, "%2hX%4hX", &i, &inst) !=  EOF)
      mem[i] = inst;

   . . .
```

short = 16 bit 2's complement integer (on arizona)

open file for reading

Unix

```
% gcc toy.c
% a.out dragon.toy
```

read TOY code from file

X-TOY Simulator

xtoy.c: parse instruction

```
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;
}
```

fetch and increment

s = bits 4-7

addr = bits 0-7

execute

ensure R[0] = 0

Shifting and Masking

Extract destination register.

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

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
inst = 3604_{16}	0	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0
inst >> 8	0	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0
15_{10}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
(inst >> 8) & 15 = 6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

X-TOY Simulator

xtoy.c: execute instruction

```
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 11: mem[R[t]] = R[d];                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;
}
```

halt

bitwise AND

load address

load indirect

jump register

X-TOY Simulator

Missing details for stdin, stdout.

- Stdin: insert following code before execute switch statement.
  - if address is FF and opcode is load or load indirect

```
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

```
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.

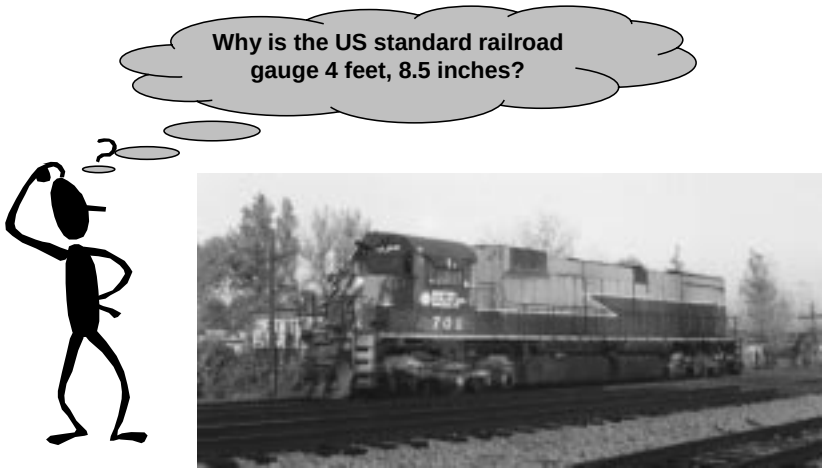
Ancient programs still running on modern computers.

- Ticketron.
- Lode Runner on Apple IIe.

Skip 6

Apple IIe Simulator

Backwards Compatibility



C and X-TOY

Correspondence between C constructs and X-TOY mechanisms.

C	X-TOY
assignment	load, store
arithmetic expressions	add, subtract
logical expressions	xor, and, shifts
loops (for, while)	jump absolute, branch
branches (if-else, switch)	branch if zero, positive
arrays, linked lists	indirect addressing
function call	jump and link, jump indirect
recursion	implement stack with arrays
whitespace	no-op 1000
. . .	. . .

Bootstrapping

Translate X-TOY program into C?

- **Easy.**

Translate C program into X-TOY?

- **Straightforward, if tedious.**

Translate X-TOY simulator from C to X-TOY?

- **Sure!**

Bootstrapping.

- **Build "first" machine.**
- **Implement simulator of itself.**
  - **C compiler written in C.**
- **Modify simulator to try new designs.**
  - **still going on!**