Lecture A4: TOY Programming

DEC PDP 12

What We've Learned About TOY

TOY: what's in it, how to use it.
- Von Neumann architecture.
- box with switches and lights.

Data representation.
- Binary and hexadecimal.

TOY instructions.
- Instruction set architecture.

Sample TOY machine language programs.
- 1 + 2 +3 + . . . n.
- LFBSR.
- Polynomial evaluation.

What We Learn Today

How to represent data other than positive integers?
- Negative numbers.

How to represent data structures?
- Arrays.

How to make function calls?

What is relationship among TOY, C, and "real computers"?

Representing Negative Numbers (Two's Complement)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+32767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
•••																
+4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
+3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
+2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
-3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
-4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
•••																
-32768	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

Two's Complement Integers Properties

Nice properties:

- 0000000000000000 represents 0.
- -0 and +0 are the same.
- Addition is easy (see next slide).

- 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}).

Alternatives other than two's complement exist.

- Many C compilers use two's complement.
- But not all, so do not assume they do.
- Unsafe C code to test if a is odd: `if (a & 1)`

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 can occur:
  - carry into sign 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

Representing Other Primitive Data Types

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.

Character strings.

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

Indexed Addressing

Static addressing.

- So far, all load/store addresses hardwired inside instruction.
- Ex. 9234: `R2 ← mem[34]`
- Need more flexibility to implement arrays, functions, etc.

indexed addressing and arrays

```
a[0] = 0;
a[1] = 1;
for (i = 2; i < 100; i++)
   a[i] = a[i-1] + a[i-2];
```

Indexed (dynamic) addressing.

- Want to be able to make memory index a variable, instead of hardwiring '34'.

Solution.

- Put memory address in register. (C "pointer")
- Use CONTENTS of register as address.
- Augment instruction format to use address register.

Review: Format 2 Instructions

Register-memory / register-immediate.

- Bits 12-15 encode opcode.
- Bits 8-11 encode destination register.
- Bits 0-7 encode memory address or arithmetic constant.

Ex: 9234 means

- Load contents of memory location 34_{16} into register R2.
- `R2 ← mem[34]`

Format 2 Instructions	
5:	jump
6:	jump if greater
7:	jump and count
8:	jump and link
9:	load
A:	store
B:	load address
E:	shift left
F:	shift right

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
9_{16}				2_{16}				34_{16}							
opcode				dest				addr							

Indexed Addressing

Bits 11 signifies "indexed addressing."

- If Bit 11 is 0 then Format 2 as usual.
- If Bit 11 is 1 then replace addr by R1 + R2
- 9234 means `R2 ← mem[34]`
- 9A34 means `R2 ← mem[R3 + R4]`

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
9_{16}				2_{16}				34_{16}							
opcode				dest				addr							

Why “Stealing” Bit 11 is OK

Bits 11 signifies "indexed addressing."

- We only have 8 registers.
- Only 3 bits (8-10) needed to distinguish among 8 values.
- Can “steal” bit 11.

Could we do the same for Format 1 instructions?

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
9_{16}				2_{16}				34_{16}							
opcode				dest				addr							

Sample C Program: Array

Goal: put Fibonacci numbers into array a[ ].

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

fibonacci.c

```
int main(void) {
  int n, i, j, k, a[16];
  n = 15;
  a[0] = 1; a[1] = 1;
  i = 0; j = 1; k = 2;
  do {
    a[k] = a[i] + a[j];
    i++; j++; k++;
    n--;
  } while (n > 0)
  return 0;
}
```

implement in TOY using indexed addressing (→ `a[k] = a[i] + a[j];`)

do-while more natural to implement in TOY (→ `} while (n > 0)`)

Sample TOY Program 3: Array

use indexed addressing three times

fibonacci.toy

```
10: B10E   R1 <- 000B
11: B001   R0 <- 0001
12: B2D0   R2 <- 00D0            a
13: A0D0   mem[D0] <- 1          a[0] = 1
14: A0D1   mem[D1] <- 1          a[1] = 1
15: B300   R3 <- 0               i = 0
16: B401   R4 <- 1               j = 1
17: B502   R5 <- 2               k = 2
18: 9E23     R6 <- mem[R2 + R3]  a[i]
19: 9F24     R7 <- mem[R2 + R4]  a[j]
1A: 1667     R6 <- R6 + R7
1B: AE25     mem[R2 + R5] <- R6  a[k]
1C: 1330     R3++                i++
1D: 1440     R4++                j++
1E: 1550     R5++                k++
1F: 7118     to 18 if --R1 > 0
```


Food for Thought

What happens if mem[12] = B210 instead of B2D0?

mystery.toy

```
10: B10E   R1 <- 000B
11: B001   R0 <- 0001
12: B2D0   R2 <- 00D0
13: A0D0   mem[D0] <- 1
14: A0D1   mem[D1] <- 1
15: B300   R3 <- 0
16: B401   R4 <- 1
17: B502   R5 <- 2
18: 9E23   R6 <- mem[R2 + R3]
19: 9F24   R7 <- mem[R2 + R4]
1A: 1667   R6 <- R6 + R7
1B: AE25   mem[R2 + R5] <- R6
1C: 1330   R3++
1D: 1440   R4++
1E: 1550   R5++
1F: 7118   to 18 if --R1 > 0
```

Overwrites mem location 12, then 13, then 14, . . .

Branches and Loops

Press GO, TOY machine either:

- Executes some instructions and halts.
- Gets caught in an infinite loop.

infinite loop

```
10: B101   R1 <- 0001
11: 5010   to 10
```

Infinite loop.

- Puzzles and/or panics programmers. Why doesn't compiler detect and tell me?
- Control structures (while, for) help manage control flow and avoid looping.
- Can always top machine by pulling plug! (Ctrl-c)

Function Calls

Functions can be used and written by different people.

Issues:

- How to pass parameter values?
- How to know where to return? (may have multiple calls)

Solution: adhere to CALLING conventions.

- Agreement between function and calling program on where to store parameters and return address.
- Assume parameter value(s) in certain register(s).
- Assume return value in specific register.
- Use indexed jump to return.

Other possible solutions.

TOY Program 4: Function Call

Goal: create function to compute a^b.

Calling convention. Store:

- 0 in R0
- a in R1
- b in R2
- addr in R4
- result in R3

function.toy

```
20: B301    R3 <- 0001
21: 1223    R2++
22: 5024    jump to 24
23: 3331    R3 <- R3 * R1
24: 7223    to 23 if --R2 > 0
25: 5804    jump to addr in R4
```

Handle b = 0

pc <- R0 + R4 = R4

How to compute a^b?

- Set R3 = 1.
- Loop b times.
  - multiply R3 by a each time

TOY Program 4: Function Call

Client program to compute $x^4 + y^5$. Assume

x in memory location D0

y in memory location D1

opcode 8
jump and link

function.toy

```
10: B000    R0 <- 0
11: 91D0    R1 <- x
12: B204    R2 <- 4
13: 8420    R3 <- x^4 (using function)
14: 1530    R5 <- R3
15: 91D1    R1 <- y
16: B205    R2 <- 5
17: 8420    R3 <- y^5 (using function)
18: 1535    R5 <- x^4 + y^5
```

R4 <- 14
pc <- 20

How To Build a TOY Machine

Hardware.

- See Lecture A5.

Simulate in software.

- Write a program to "simulate" the behavior of the TOY machine.
- Java TOY simulator.
- C TOY simulator.

TOY SIMULATOR: toy.c

- short = 16 bit 2's comp integer (on arizona)
- initialize memory to 0
- read program
- fetch and increment
- r1 = bits 4, 5, 6
- indexed addressing
- execute

```
int main(void) {
  short int inst, R[8], mem[256];
  unsigned char pc = 0X10;
  int i, op, addr, r0, r1, r2, c;
  for (i = 0; i < 256; i++)
     mem[i] = 0;
  while (scanf("%hX%hX",&i, &inst)!=EOF)
     mem[i] = inst;
  do {
    inst = mem[pc++];
    op   = (inst >> 12) &  15;
    r0   = (inst >>  8) &   7;
    r1   = (inst >>  4) &   7;
    r2   = (inst >>  0) &   7;
    addr = (inst >>  0) & 255;
    if ((inst >> 11) & 1)
       addr = (R[r1] + R[r2]) & 255;
    . . .
  } while (op != 0)
  return 0;
}
```


TOY SIMULATOR: toy.c (cont)

- halt
- multiply
- jump and count
- load address
- right shift

```
switch (op) {
  case  0:                              break;
  case  1: R[r0] = R[r1] + R[r2];       break;
  case  2: R[r0] = R[r1] - R[r2];       break;
  case  3: R[r0] = R[r1] * R[r2];       break;
  case  4: printf("%04X\n", R[r0]);     break;
  case  5: pc = addr;                   break;
  case  6: if (R[r0] > 0) pc = addr;    break;
  case  7: if (--R[r0]) pc = addr;      break;
  case  8: R[r0] = pc; pc = addr;       break;
  case  9: R[r0] = mem[addr];           break;
  case 10: mem[addr] = R[r0];           break;
  case 11: R[r0] = addr;                break;
  case 12: R[r0] = R[r1] ^ R[r2];       break;
  case 13: R[r0] = R[r1] & R[r2];       break;
  case 14: R[r0] = R[r0] >> addr;       break;
  case 15: R[r0] = R[r0] << addr;       break;
}
```


Simulation

Consequences of simulation.

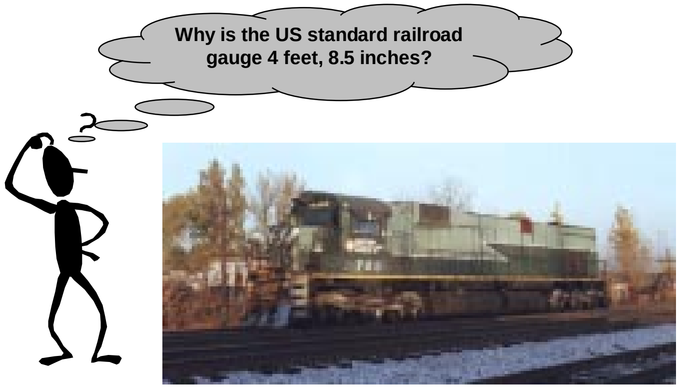
- Test out new machine (or microprocessor) using simulator.
  - cheaper and faster than building actual machine
- Easy to add other functions to simulator.
  - trace, single-step, breakpoint debugging
  - simulator more powerful than TOY itself
- Reuse software for old machines.

Ancient programs still running on modern computers.

- Ticketron - 5 cents per ticket.
- Lode Runner on Apple IIe.

Apple IIe Simulator

Simulation



Bootstrapping

Translate TOY program into C?
- Easy.

Translate C program to TOY?
- Straightforward, if tedious.

Translate TOY simulator into TOY? (!)
- Yes.

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