COS 126 Lecture 5: TOY programming

Box with switches and lights
- CPU (central processing unit)
- 256 16-bit words of memory
- 8 registers (1st eight words of memory)

Hexadecimal
- shorthand notation for bits

to LOAD a memory word
- set address switches
- set memory switches
- press LOAD

Program: sequence of instructions

to RUN a program
- set address switches (to initialize PC)
- press GO (or STEP for just one instr.)

to LOOK at a memory word
- set address switches
- press LOOK
- examine lights

Same basic principles as
- early computers
- modern microprocessors

TOY instructions

Instruction formats

FORMAT 1: opcode, r0, r1, and r2

FORMAT 2: opcode, r0, and 8-bit addr

TRANSFER between registers and memory

9:	**load**	**r0 <- mem[addr]**
A:	**store**	**mem[addr] -> r0**
B:	**load address**	**r0 <- addr**

ARITHMETIC operations

1:	**add**	**r0 <- r1 + r2**
2:	**subtract**	**r0 <- r1 - r2**
3:	**multiply**	**r0 <- r1 * r2**

LOGICAL operations

C:	**xor**	**r0 <- r1 ^ r2**
D:	**and**	**r0 <- r1 & r2**
E:	**shift right**	**r0 <- r0 >> addr**
F:	**shift left**	**r0 <- r0 << addr**

CONTROL

0:	**halt**	**halt**
4:	**system call**	**print r0 on tty**
5:	**jump**	**pc <- addr**
6:	**jump if positive**	**if (r0 > 0) pc <- addr**
7:	**jump and count**	**r0 <- r0 - 1** **if (r0 != 0) pc <- addr**
8:	**jump and link**	**r0 <- pc; pc <- addr**

To: Toy machine programmers

From: Hardware Design Team

Subject: Toy v6.0

Version 6.0 of the Toy architecture is incompatible with previous versions. Among other things, we have changed the way adressing is done and we have bagged the divide instruction.

All software written for old machines must be reimplemented. People who took cs126 last year learned about some obsolete version.

Don’t tell the customers.

Indexed Addressing

Load, store

    addresses hardwired into instructions
    flexibility needed for memory access

Solution:

    use CONTENTS of register as address

How to specify instruction?

    * eight registers
    * four bits to specify register
    * only *need* three

For all Format 1 instructions

    define leading bit of second digit
    to be the INDEX bit
        if it’s 1, add r1 and r2 to get addr
        if it’s 0, take address as before

Ex:

    if R2 is 0030 and R4 = 0002
    then 9F24 loads mem[32] into r7

Note: B923 is equivalent to 1123
(these things happen in machine design!)

Sample Toy program 3

Put Fibonacci numbers

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

into memory, starting at location 30

(memory use corresponds to C array)

```
10: B10E    R1 <- 000E
11: B001    R0 <- 0001
12: B230    R2 <- 0030          a
13: A030    mem[30] <- 1        a[0] = 1
14: A031    mem[31] <- 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
```

Result of pressing GO with PC at 10:

```
0010: B10E B001 B230 A030 A031 B300 B401 B502
0018: 9E23 9F24 1667 AE25 1330 1440 1550 7118
0030: 0001 0001 0002 0003 0005 0008 000D 0015
0038: 0022 0037 0059 0090 00E9 0179 0262 03DB
```

Food for thought:

what happens if we change mem[12] to B210 ??

Branches and Looping

Press GO, computer either
- * executes some instructions and halts
- * gets caught in a loop

"infinite loop"
    puzzles and/or panics programmers
    (it will happen to you!)

```
     xx:
               .
               .
               .
               50xx
```

Often more complicated

Can't know whether or not a program
    will loop, in general

Control structures (for and while) help
    manage branching, avoid looping

Can always stop TOY by pulling the plug!

Profound implications (stay tuned)

Logical Instructions

C: xor
D: and
E: shift right
F: shift left

xor, and: bit-by-bit operations
shift: move bits

MASKing with "and instruction"

1010010x01111010
0000000100000000
0000000x00000000

Can implement other logical operations

	and	xor	or
a b	&	^	(a & b) ^ (a ^ b)
0 0	0	0	0
0 1	0	1	1
1 0	0	1	1
1 1	1	0	1

Sample TOY program 4

Ex: suppose that memory locations 10-15 contain

```
10: 911F B000 1210 1310 E203 E30A C323 B401
18: D334 F101 C113 0000 0000 0000 0000 0684
```

Set PC to 10. Press GO. What happens?

Step-by-step:

```
10: 911F    R1 <- 0684        0000011010000100
11: B000    R0 <- 0000
12: 1210    R2 <- R1 + R0     0000011010000100
13: 1310    R3 <- R1 + R0     0000011010000100
14: E203    R2 <- R2 >> 3     0000000011010000
15: E30A    R3 <- R3 >> 10    0000000000000001
16: C323    R3 <- R2 ^ R3     0000000011010001
17: B401    R4 <- 0001        0000000000000001
18: D334    R3 <- R3 & R4     0000000000000001
19: F101    R1 <- R1 << 1     0000110100001000
1A: C113    R1 <- R1 ^ R3     0000110100001001
1B: 0000    halt
```

Simulates one step of LFBSR of Lecture 1

Function implementations

To implement functions

- assume parameter values in registers
- assume return value in register
- use indexed jump to return

Ex: function to compute a to the b-th power

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

Implementation computes a to the b-th power

- by looping b times
- multiplying R3 by a each time

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

CALLING convention

agreement between function and calling program on parameters and return address

Function calls

Adhere to calling conventions to
get function to perform computation
with different parameter values

Ex: program that calls the function
on the previous slide twice
to compute x^4 + y^5
x in mem loc 1E
y in mem loc 1F

```
10: B000    R0 <- 0
11: 911E    R1 <- x
12: B204    R2 <- 4
13: 8420    R3 <- x^4 (using function)
14: 1530    R5 <- R3
15: 911F    R1 <- y
16: B205    R2 <- 5
17: 8420    R3 <- y^5 (using function)
18: 1535    R5 <- x^4 + y^5
```

Precious resource: registers

Call a function from within a function?
(stay tuned)

Representations

We've been assuming nonnegative integers

negative integers: use "two's complement"

```
0000000000000100  4
0000000000000011  3
0000000000000010  2
0000000000000001  1
0000000000000000  0
1111111111111111 -1
1111111111111110 -2
1111111111111101 -3
1111111111111100 -4
```

big integers: could use "multiple precision"
 multiple words per integer
 required for multiply, divide?
real numbers: could use "floating point"
 like scientific notation
character strings: could use ASCII code
 8 bits/character (packed/unpacked)

Details (for real machines): stay tuned

Does a C program know the representation?
 ideally: NO
 in practice: YES

TOY perspective

MACHINE LANGUAGE:
    hex codes we've used
ASSEMBLY LANGUAGE:
    names for opcodes, regs, memory locs
    1-1 translation to machine language

TOY has the same basic computational tools that we use in C
    data representations
        integer, float, char
    control
        assignment, loop, conditional functions
"All" programming languages have these fundamental characteristics

COMPILER translates a program from
    high-level language like C to
    low-level machine language like TOY
Different ways to express same
    underlying computation (algorithm)

Contents of memory
    DATA if referenced by instruction
    INSTRUCTION if referenced by PC
Essential characteristic of
    VON NEUMANN MACHINE