CS 126 Lecture A1: TOY Machine

Outline

- **Introduction**
- Toy machine
- Machine language instructions
- Example machine language programs
- Conclusions

Brief History Leading to the Dominance of von Neumann Architecture

- 1940s, Atanasoff, Iowa State, first **special-purpose** electronic computer, binary representation of numbers
- ~1946, ENIAC, Eckert and Mauchly, UPenn, first general-purpose electronic computer
  - 100 ft long, 8.5 ft high, several ft wide, 18000 vacuum tubes
  - conditional jumps, **programmable**
  - code: setting switches, data: punch cards
  - Used to compute artillery firing tables
- 1944, von Neumann, visited ENIAC, the **"von Neumann Memo"**, concept of a "stored-program" computer
- 1949, Wilkes, EDSAC, first **stored-program** computer
- 1946, von Neumann, Goldstine, Burks, IAS machine, Princeton, the report pioneered most modern computer architecture concepts

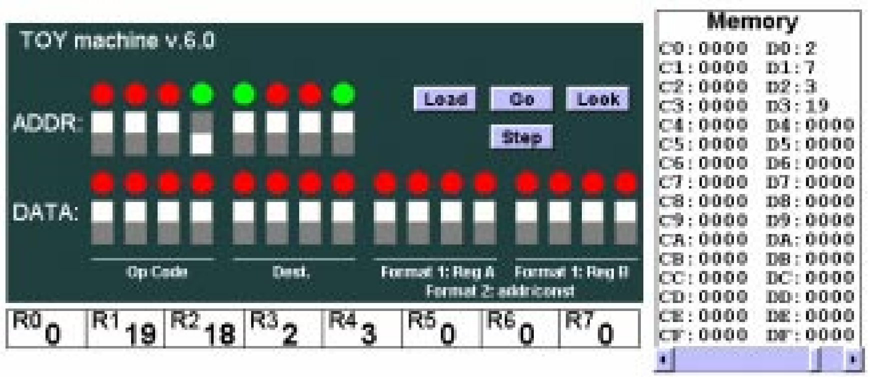
Why Study Machine Language Programming Today

- Learn how computers really work
- There are still (a few) situations where machine language programming is necessary
- The first step towards understanding how to build better computers

Outline

- ~~Introduction~~
- **Toy machine**
- Machine language instructions
- Example machine language programs
- Conclusions

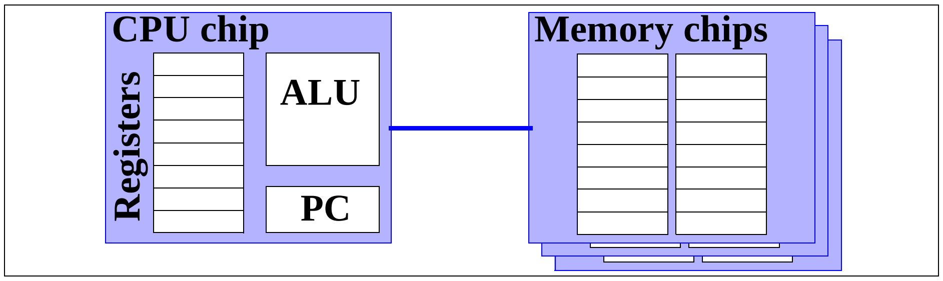
Toy Machine



```
Memory
C0:0000  D0:2
C1:0000  D1:7
C2:0000  D2:3
C3:0000  D3:19
C4:0000  D4:0000
C5:0000  D5:0000
C6:0000  D6:0000
C7:0000  D7:0000
C8:0000  D8:0000
C9:0000  D9:0000
CA:0000  DA:0000
CB:0000  DB:0000
CC:0000  DC:0000
CD:0000  DD:0000
CE:0000  DE:0000
CF:0000  DF:0000
```

- An imaginary machine, similar to
  - * ancient early computers
  - * today's microprocessers
- Box with switches and lights, maybe TTY

Inside the Box



- ALU (arithmetic logic unit) -- executes instructions to manipulate date
- 8 registers -- the fastest form of storage, on-chip in modern computers, used as scratch space during computation
- PC (program counter) -- a register with special meaning, keeps track of the next instruction to be executed
- 256 16-bit words of memory -- stores both code and data

Binary Numbers

- Machine consists of two-state ("ON-OFF") switches and lights
- Use binary encoding to represent values

Ex: integers

.6375 = 0001100011100111

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

$2^{12} + 2^{11}$ $\quad 2^{7} + 2^{6} + 2^{5}$ $\quad 2^{2} + 2^{1} + 2^{0}$

.6375 = 4096+2048 +128+64+32 +4 +2 +1

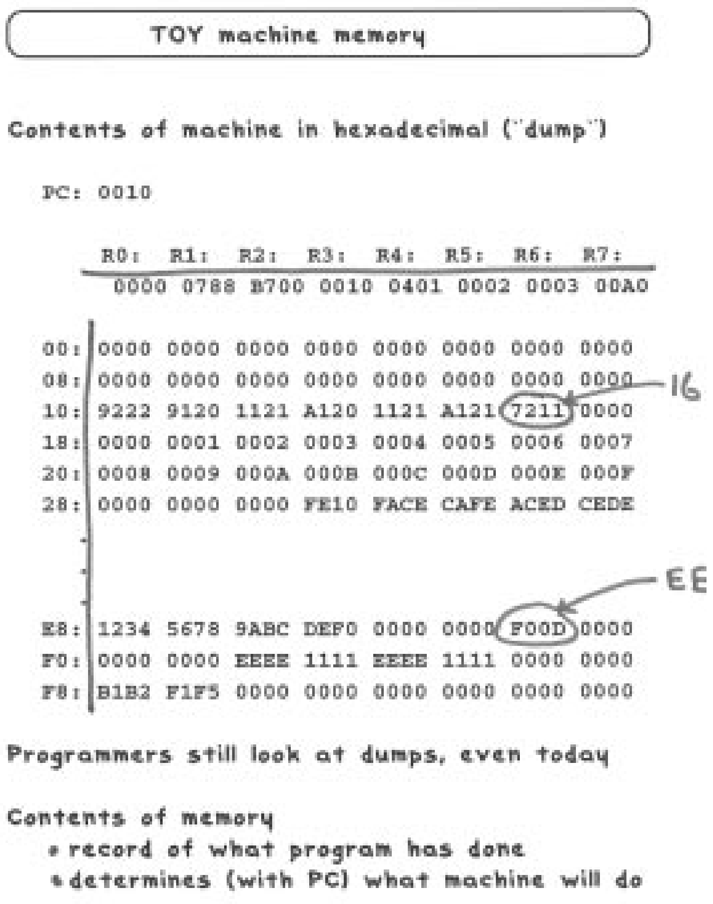
Hexadecimal Numbers

- Hexadecimal (base-16) notation provides shorthand binary code four bits at a time

0000	0001	0010	0011	0100	0101	0110	0111
0	1	2	3	4	5	6	7
1000	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
8	9	A	B	C	D	E	F

Ex:

- 6375 = $\underbrace{0001}_{1}\underbrace{1000}_{8}\underbrace{1110}_{E}\underbrace{0111}_{7}$
- 6375 = $1*16^3 + 8*16^2 + 14*16^1 + 7*16^0$
- = 4096 + 2048 + 224 + 7



TOY machine memory

Contents of machine in hexadecimal ("dump")

```
PC: 0010

      R0:  R1:  R2:  R3:  R4:  R5:  R6:  R7:
      0000 0788 B700 0010 0401 0002 0003 00A0

00: 0000 0000 0000 0000 0000 0000 0000 0000
08: 0000 0000 0000 0000 0000 0000 0000 0000  16
10: 9222 9120 1121 A120 1121 A121 7211 0000
18: 0000 0001 0002 0003 0004 0005 0006 0007
20: 0008 0009 000A 000B 000C 000D 000E 000F
28: 0000 0000 0000 FE10 FACE CAFE ACED CEDE
 .
 .
 .                                           EE
E8: 1234 5678 9ABC DEF0 0000 0000 F00D 0000
F0: 0000 0000 EEEE 1111 EEEE 1111 0000 0000
F8: B1B2 F1F5 0000 0000 0000 0000 0000 0000
```

- Programmers still look at dumps, even today
- Contents of memory
  - record of what program has done
  - determines (with PC) what machine will do

Program and Data

Program: sequence of instructions

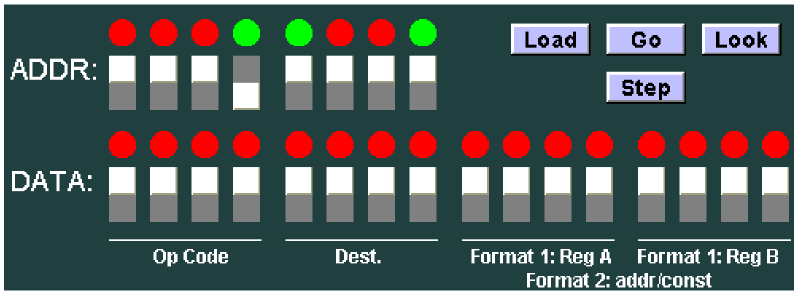
Instruction:

16-bit word (interpreted one way)

Data:

16-bit word (interpreted other ways)

How to Use the TOY Machine



To run a program

* load the program and data

(set switches, press LOAD for each wor

* set switches to address of first instruction
* press GO

How to Use the TOY Machine

GO button
* loads PC from address switches
* initiates FETCH-INCREMENT-EXECUTE cycle
* machine runs until halt instruction hit

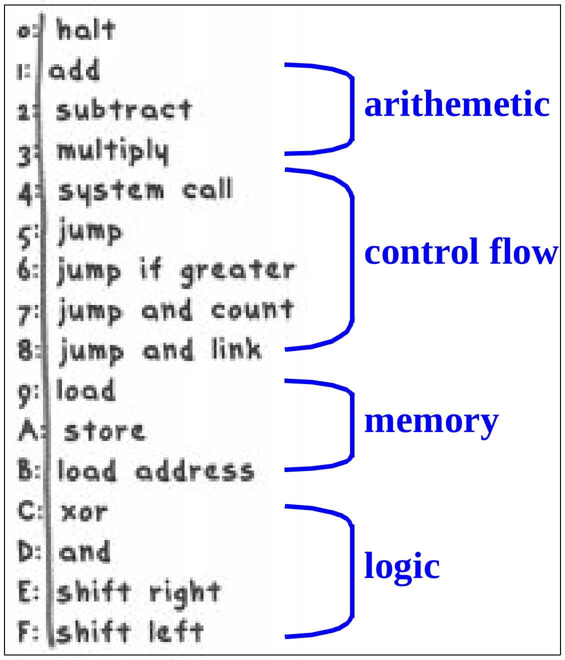
FETCH (get instruction from memory into CPU)
INCREMENT program counter (PC)
EXECUTE (may require data from or to memory)

"Output:"
- read contents of memory word in lights
- system call can write output
  to an output device (tty)

Outline

- ~~Introduction~~
- ~~Toy machine~~
- **Machine language instructions**
- Example machine language programs
- Conclusions

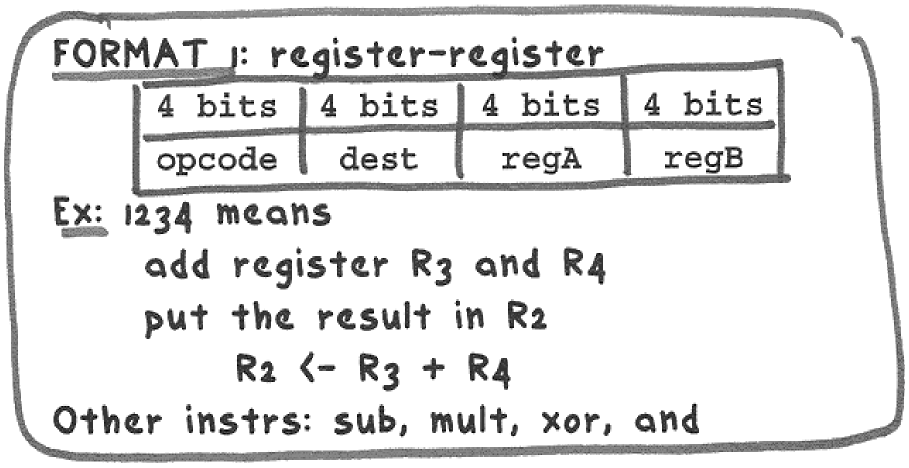
TOY Instructions



0:	halt	
1:	add	arithemetic
2:	subtract	
3:	multiply	
4:	system call	control flow
5:	jump	
6:	jump if greater	
7:	jump and count	
8:	jump and link	
9:	load	memory
A:	store	
B:	load address	
C:	xor	logic
D:	and	
E:	shift right	
F:	shift left	

- Encode each of these instructions using 16 bits
- Need to divide up the 16 bits to denote components of each type of instructions
- Instruction formats - different ways of dividing up the 16 bits

Instruction Format 1



FORMAT 1: register-register

4 bits	4 bits	4 bits	4 bits
opcode	dest	regA	regB

Ex: 1234 means
add register R3 and R4
put the result in R2
R2 <- R3 + R4
Other instrs: sub, mult, xor, and

Instruction Format 2

FORMAT 2: register-memory, register-immediate

4 bits	4 bits	8 bits
opcode	dest	addr/const

Ex: 9234 means "load memory loc 34 (hex) into R2"

R2 <- mem[34]

Ex: A234 means "store R2 into memory loc 34"

mem[34] <- R2

Ex: B234 means "load the value 0034 into R2"

R2 <- 0034

Other instrs: shifts, halt, system call, jumps

Logical Instructions

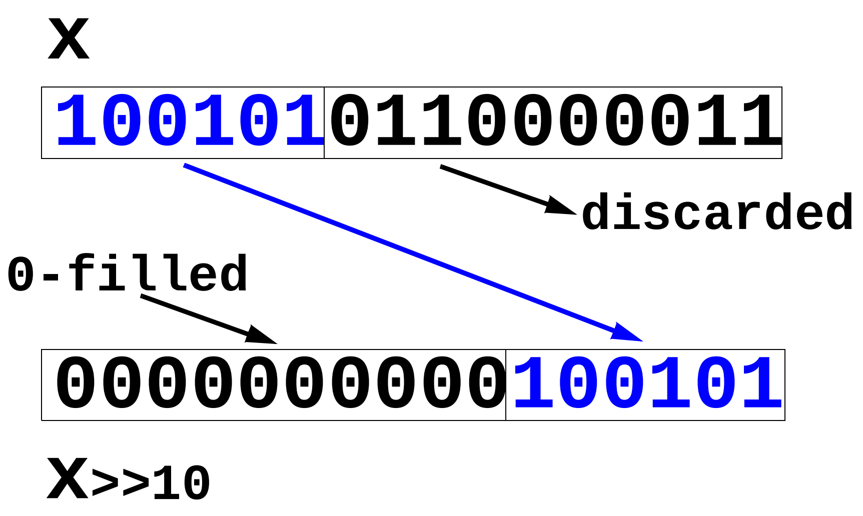
opcode

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

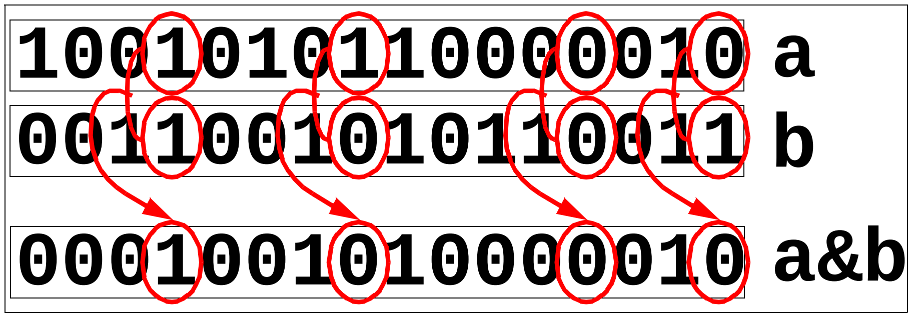
xor, and: bit-by-bit operations

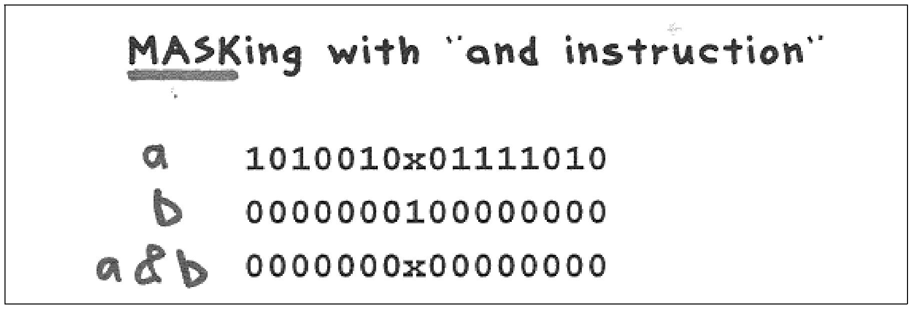
shift: move bits

Right-Shift



Bit-by-Bit-And

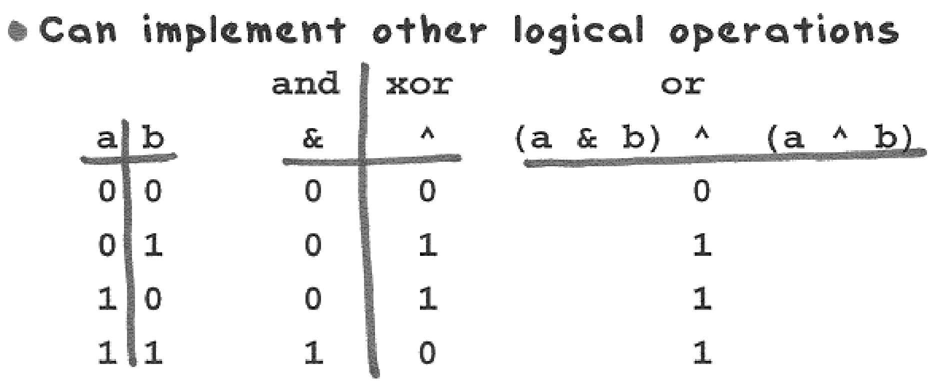




MASKing with "and instruction"

```
a     1010010x01111010
b     0000000100000000
a&b   0000000x00000000
```

Other Logical Operations

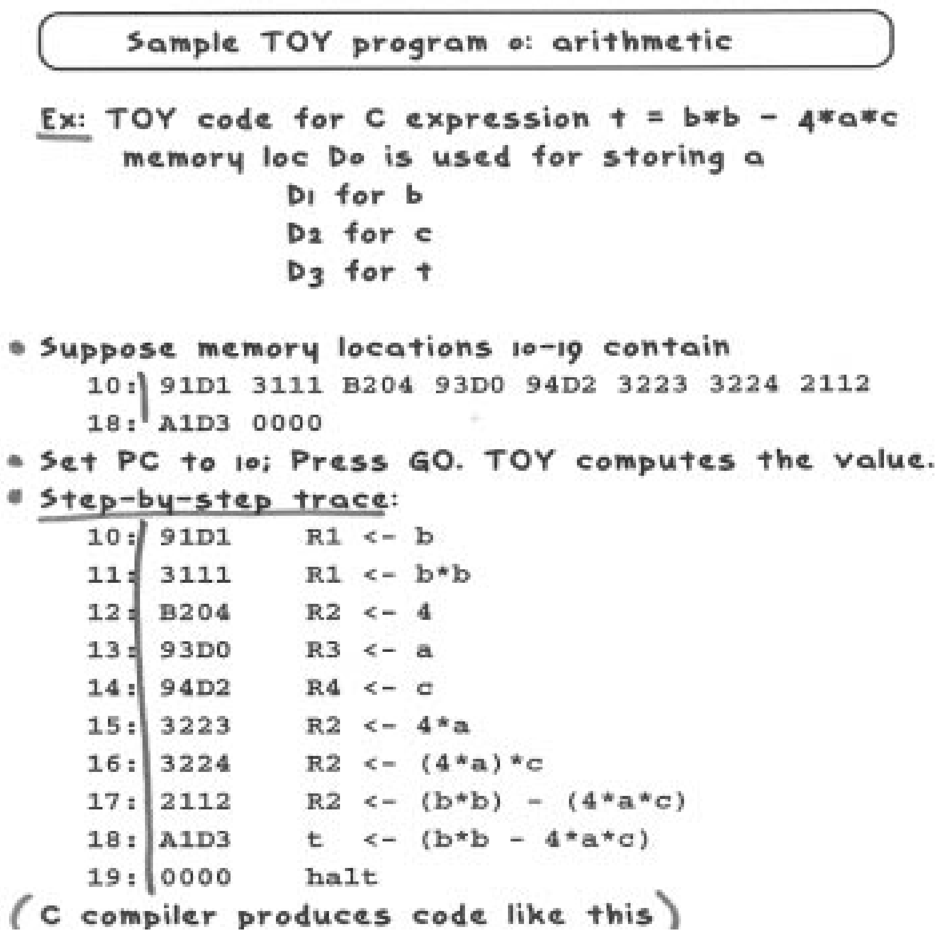


- Can implement other logical operations

		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

Outline

- ~~Introduction~~
- ~~Toy machine~~
- ~~Machine language instructions~~
- **Example machine language programs**
- Conclusions



Sample TOY program o: arithmetic

Ex: TOY code for C expression t = b*b - 4*a*c
memory loc Do is used for storing a
D1 for b
D2 for c
D3 for t

- Suppose memory locations 10-19 contain

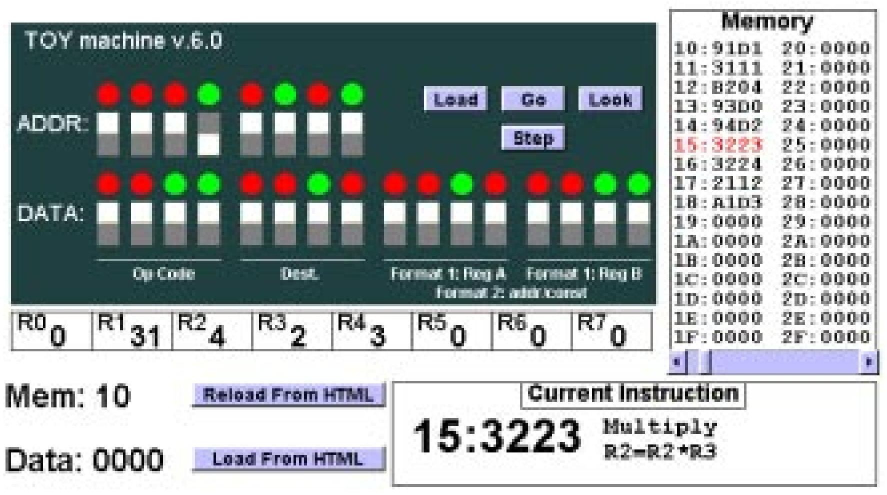
```
10: 91D1 3111 B204 93D0 94D2 3223 3224 2112
18: A1D3 0000
```

- Set PC to 10; Press GO. TOY computes the value.
- Step-by-step trace:

```
10: 91D1    R1 <- b
11: 3111    R1 <- b*b
12: B204    R2 <- 4
13: 93D0    R3 <- a
14: 94D2    R4 <- c
15: 3223    R2 <- 4*a
16: 3224    R2 <- (4*a)*c
17: 2112    R2 <- (b*b) - (4*a*c)
18: A1D3    t  <- (b*b - 4*a*c)
19: 0000    halt
```

(C compiler produces code like this)

TOY Demo



Memory

```
10:91D1  20:0000
11:3111  21:0000
12:B204  22:0000
13:93D0  23:0000
14:94D2  24:0000
15:3223  25:0000
16:3224  26:0000
17:2112  27:0000
18:A1D3  28:0000
19:0000  29:0000
1A:0000  2A:0000
1B:0000  2B:0000
1C:0000  2C:0000
1D:0000  2D:0000
1E:0000  2E:0000
1F:0000  2F:0000
```

R0	R1	R2	R3	R4	R5	R6	R7
0	31	4	2	3	0	0	0

Mem: 10

Data: 0000

Current Instruction

15:3223 Multiply R2=R2*R3

Sample TOY program 1: more arithmetic

Ex: Suppose memory locations 10-1F contain

```
10: B001 B200 B101 1221 1110 1221 1110 1221
18: 1110 1221 1110 1221 1110 1221 0000 0000
```

- Set PC to 10. Press GO. What happens?
- Step-by-step trace:

```
10:| B001     R0 <- 0001
11:| B200     R2 <- 0000                 0000
12:| B101     R1 <- 0001          0001
13:| 1221     R2 <- R2 + R1              0001
14:| 1110     R1 <- R1 + R0       0002
15:| 1221     R2 <- R2 + R1              0003
16:| 1110     R1 <- R1 + R0       0003
17:| 1221     R2 <- R2 + R1              0006
18:| 1110     R1 <- R1 + R0       0004
19:| 1221     R2 <- R2 + R1              000A
1A:| 1110     R1 <- R1 + R0       0005
1B:| 1221     R2 <- R2 + R1              000F
1C:| 1110     R1 <- R1 + R0       0006
1D:| 1221     R2 <- R2 + R1              0015
1E:| 0000     halt
```

Computes 1 + 2 + 3 + 4 + 5 + 6 = 21

Sample TOY program 2: loop

- Suppose memory locations 10-17 contain

```
10: B106 B200 B001 1221 2110 6113 0000 0000
```

- Set PC to 10. Press GO. What happens?
- Step-by-step trace:

```
                                    N
10:| B106     R1 <- 0006          0006 sum
11:| B200     R2 <- 0000                 0000
12:| B001     R0 <- 0001
13:| 1221     R2 <- R2 + R1              0006
14:| 2110     R1 <- R1 - R0       0005
15:| 6113     jump if (R1 > 0)
13:| 1221     R2 <- R2 + R1              000B
14:| 2110     R1 <- R1 - R0       0004
15:| 6113     jump if (R1 > 0)
13:| 1221     R2 <- R2 + R1              000F
14:| 2110     R1 <- R1 - R0       0003
15:| 6113     jump if (R1 > 0)
13:| 1221     R2 <- R2 + R1              0012
14:| 2110     R1 <- R1 - R0       0002
15:| 6113     jump if (R1 > 0)
13:| 1221     R2 <- R2 + R1              0014
14:| 2110     R1 <- R1 - R0       0001
15:| 6113     jump if (R1 > 0)
13:| 1221     R2 <- R2 + R1              0015
14:| 2110     R1 <- R1 - R0       0000
15:| 6113     jump if (R1 > 0)
16:| 0000     halt
```

Computes
$N + (N-1) + \dots + 3 + 2 + 1 = N(N+1)/2$
for *any* value N loaded into R_1

Avg

Horner's Method

Problem:

- evaluate 2x^3 + 3x^2 + 9x + 7 at x = 10
  assume "data" stored in locations 30--34

```
        x    a    b    c    d
30: 000A 0002 0003 0009 0007 0000 0000 0000
```

First try:

- compute x^3, mult. by a; compute x^2, ...
  (cumbersome, inefficient)

- **Efficient algorithm** (Horner's method):
  rewrite ax^3+bx^2+cx+d as $\boxed{((ax+b)x+c)x+d}$

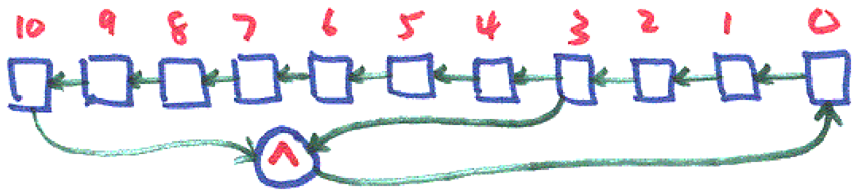
Sample TOY Program 3: Horner's Method

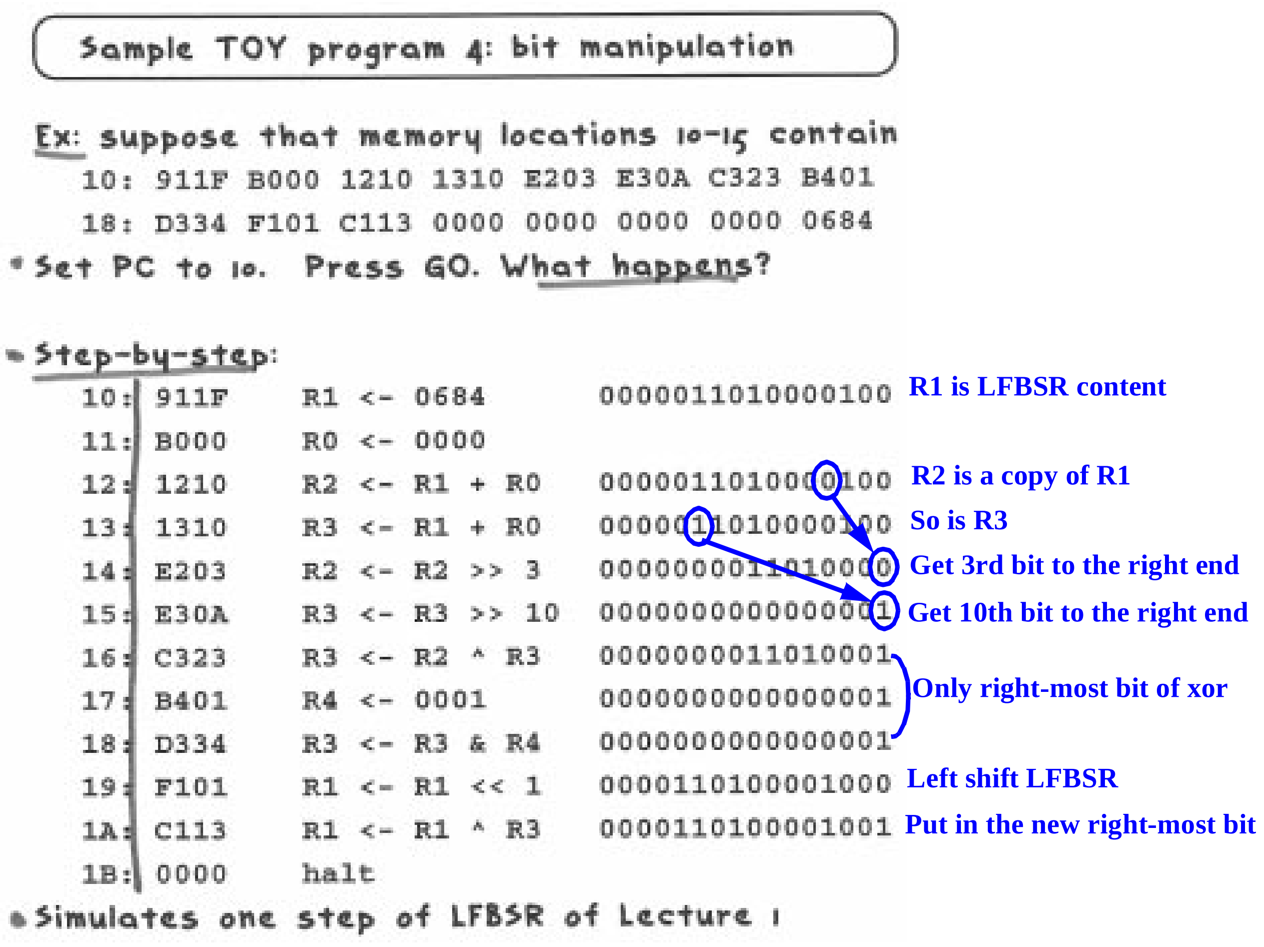
- **Efficient algorithm** (Horner's method):
  rewrite ax^3+bx^2+cx+d as $\boxed{((ax+b)x+c)x+d}$

```
10: 9430  R4 <- M[30]     000A x
11: 9531  R5 <- M[31]     0002 a
12: 3554  R5 <- R5 * R4   0014 a*x
13: 9632  R6 <- M[32]     0003 b
14: 1556  R5 <- R5 + R6   0017 a*x+b
15: 3554  R5 <- R5 * R4   00DC (a*x+b)*x
16: 9633  R6 <- M[33]     0009 c
17: 1556  R5 <- R5 + R6   00E5 (a*x+b)*x + c
18: 3554  R5 <- R5 * R4   0956 ((a*x+b)*x+c)*x
19: 9634  R6 <- M[34]     0007 d
1A: 1556  R5 <- R5 + R6   095D ((a*x+b)*x+c*x)+d
1B: 4502  write R5 to tty
```

LFBSR

Linear feedback shift register (LFBSR)





Sample TOY program 4: bit manipulation

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  R1 is LFBSR content
11: B000   R0 <- 0000
12: 1210   R2 <- R1 + R0  0000011010000100  R2 is a copy of R1
13: 1310   R3 <- R1 + R0  0000011010000100  So is R3
14: E203   R2 <- R2 >> 3  0000000011010000  Get 3rd bit to the right end
15: E30A   R3 <- R3 >> 10 0000000000000001  Get 10th bit to the right end
16: C323   R3 <- R2 ^ R3  0000000011010001
17: B401   R4 <- 0001     0000000000000001  Only right-most bit of xor
18: D334   R3 <- R3 & R4  0000000000000001
19: F101   R1 <- R1 << 1  0000110100001000  Left shift LFBSR
1A: C113   R1 <- R1 ^ R3  0000110100001001  Put in the new right-most bit
1B: 0000   halt
```

* Simulates one step of LFBSR of Lecture 1

Outline

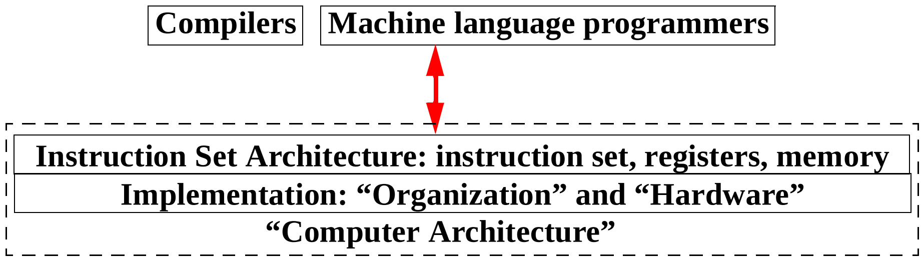
- ~~Introduction~~
- ~~Toy machine~~
- ~~Machine language instructions~~
- ~~Example machine language programs~~
- Conclusions

Basic Characteristics of TOY Machine

TOY is a "general purpose" computer

- "von Neumann" machine
    - instructions and data in same memory
    - can change program (control) w/o rewiring
        immediate applications
        profound implications
- sufficient power to perform any computation
    - limited only by amount of memory (and time)
      [stay tuned]
- similar to real machines

“Computer Architecture”


Compilers
Machine language programmers
Instruction Set Architecture: instruction set, registers, memory
Implementation: “Organization” and “Hardware”
“Computer Architecture”

- Interface--“instruction set architecture” (ISA)
  - visible to machine language programmers
  - boundary between software and hardware
- Implementation
  - “Organization”: interaction of high-level components
  - “Hardware”: low level specifics such as detailed logic design
- Abstractions
  - Can change hardware without changing organization
  - Can change implementation without changing ISA