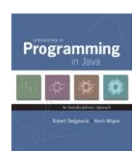
5. The TOY Machine

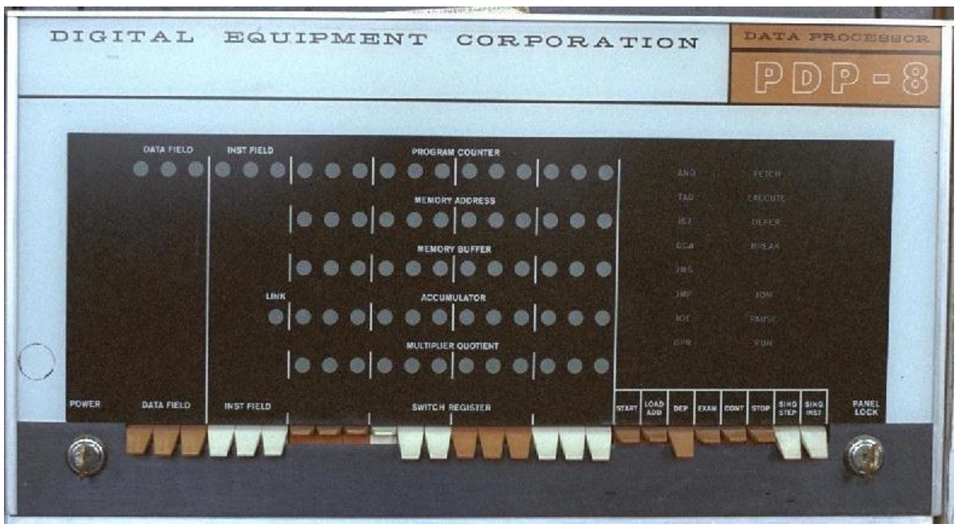


Booksite, Section 5

What is TOY?

An imaginary machine similar to:

- Ancient computers.
- Today's microprocessors.



Why Study TOY?

Machine language programming

- How do Java programs relate to computer?
- Key to understanding Java references.
- Some situations today where it is really necessary.

Games, Scientific Computing, GPU (Graphics Card) Programming

Computer architecture

- How does it work?
- How is a computer put together?

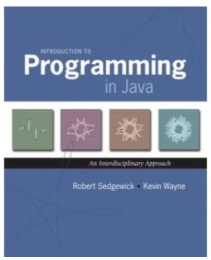
TOY machine: Optimized for simplicity, not cost or speed.

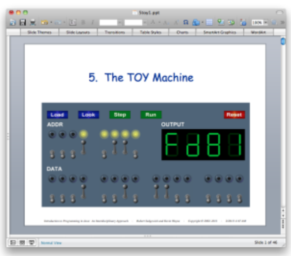
Penn Engineering

Data and Programs Are Just Bits

Each bit consists of two states:

- Switch is on or off; wire has high voltage or low voltage.





Everything in a computer is stored as bits.

– Data ***and programs***

– Text, pictures, sounds, movies, programs, …

Penn Engineering

Compilers

Compilers translate from high-level languages (like Java) to native machine instructions (bits).

```
int x = 8;
int y = 5;
int sum = x + y;
```

Compilation

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

`add.toy`

memory addresses

instructions (bits, written in hex)

assembly (human-readable mnemonics)

	Java	TOY
Variable names	✔	✗
Function names	✔	✗
Structure	✔	✗
Types	✔	✗
Error Checking	✔	✗
*, /, %, useful libraries	✔	✗
Bits	✗	✔

Penn Engineering

Binary People

http://www.thinkgeek.com/tshirts-apparel/unisex/frustrations/5aa9

Penn Engineering

Inside the TOY Box

Switches: Input data and programs.

Lights: View data.

Memory:

- Stores data and programs.
- 256 16-bit "words."
- Special word for stdin / stdout.

Program counter (PC):

- An extra 8-bit register.
- Next instruction to be executed.

Registers:

- Fastest form of storage.
- Scratch space during computation.
- 16 16-bit registers.
- Register 0 is always 0.

Arithmetic-Logic Unit (ALU): Manipulate data stored in registers.

Standard input/output: Interact with outside world.

Penn Engineering

TOY Machine "Core" Dump

A core dump is the contents of machine at a particular place and time.

- Record of what program has done.
- Completely determines what machine will do.

Registers

R0	R1	R2	R3
0000	0000	0000	0000
R4	**R5**	**R6**	**R7**
0000	0000	0000	0000
R8	**R9**	**RA**	**RB**
0000	0000	0000	0000
RC	**RD**	**RE**	**RF**
0000	0000	0000	0000

pc

10

index of next instruction

data

program

variables

Memory

00:	0008	0005	0000	0000	0000	0000	0000	0000
08:	0000	0000	0000	0000	0000	0000	0000	0000
10:	8A00	8B01	1CAB	9C02	0000	0000	0000	0000
18:	0000	0000	0000	0000	0000	0000	0000	0000
20:	0000	0000	0000	0000	0000	0000	0000	0000
28:	0000	0000	0000	0000	0000	0000	0000	0000
.								
.								
E8:	0000	0000	0000	0000	0000	0000	0000	0000
F0:	0000	0000	0000	0000	0000	0000	0000	0000
F8:	0000	0000	0000	0000	0000	0000	0000	0000

Penn Engineering

A Sample Program

A sample program: Adds 0008 + 0005 = 000D.

RA	RB	RC
0000	0000	0000

Registers

pc
10

Program Counter

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

add.toy

TOY memory (program and data)

comments

A Sample Program

Program counter: The pc is initially `10`, so the machine interprets `8A00` as an instruction.

RA	RB	RC
0000	0000	0000

pc
10

Index of next instruction to execute.

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

add.toy

Penn Engineering

Load

Load [opcode 8]

- Loads the contents of some memory location into a register.
- `8A00` means "load the contents of memory cell `00` into register `A`."

RA	RB	RC
0000	0000	0000

pc
10

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
8_{16}				A_{16}				00_{16}							
opcode				dest d				addr							

Load

Load [opcode 8]

- Loads the contents of some memory location into a register.
- `8B00` means "load the contents of memory cell `00` into register `B`."

RA	RB	RC
0008	0000	0000

pc
11

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	1
8_{16}				B_{16}				01_{16}							
opcode				dest d				addr							

Add

Add [opcode 1]

- Add contents of two registers and store sum in a third
- `1CAB` means "add the contents of registers `A` and `B` and put the result in register `C`."

RA	RB	RC
0008	0005	0000

pc
12

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	1	0	1	0	1	1
1_{16}				C_{16}				A_{16}				B_{16}			
opcode				dest d				source s				source t			

Store

Store [opcode 9]

- Stores the contents of some register into a memory cell.
- `9C02` means "store the contents of register `C` into memory cell `02`."

RA	RB	RC
0008	0005	000D

pc
13

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	0	0	0	0	0	0	0	0	2	0
9_{16}				C_{16}				02_{16}							
opcode				dest d				addr							

Penn Engineering

Halt

Halt [opcode 0]

- Stop the machine.

RA	RB	RC
0008	0005	000D

pc
14

```
00: 0008    8
01: 0005    5
02: 0000    0

10: 8A00    RA ← mem[00]
11: 8B01    RB ← mem[01]
12: 1CAB    RC ← RA + RB
13: 9C02    mem[02] ← RC
14: 0000    halt
```

Program and Data

Program: Sequence of 16-bit integers, interpreted one way.

Data: Sequence of 16-bit integers, interpreted other way.

Program Counter (pc): Holds memory address of the next "instruction" and determines which integers get interpreted as instructions.

16 instruction types: Changes contents of registers, memory, and pc in specified, well-defined ways.

	Instructions
0:	halt
1:	add
2:	subtract
3:	and
4:	xor
5:	shift left
6:	shift right
7:	load address
8:	load
9:	store
A:	load indirect
B:	store indirect
C:	branch zero
D:	branch positive
E:	jump register
F:	jump and link

Penn Engineering

TOY Instruction Set Architecture

TOY instruction set architecture (ISA):

- Interface that specifies behavior of machine.
- 16 register, 256 words of main memory, 16-bit words.
- 16 instructions.

Each instruction consists of 16 bits:

- Bits 12-15 encode one of 16 instruction types or opcodes.
- Bits 8-11 encode destination register d.
- Bits 0-7 encode:
  [Format 1] source registers s and t ← add, subtract, …
  [Format 2] 8-bit memory address or constant ← load, store, …

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

Interfacing with the TOY Machine

To enter a program or data:

- Set 8 memory address switches.
- Set 16 data switches.
- Press Load: data written into addressed word of memory.

To view the results of a program:

- Set 8 memory address switches.
- Press Look: contents of addressed word appears in lights.

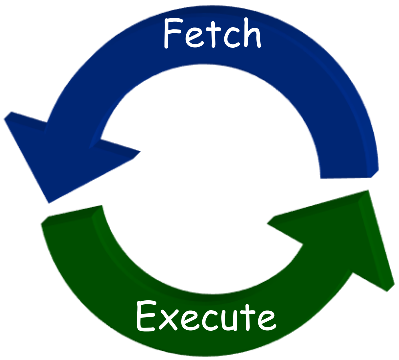
Interfacing with the TOY Machine

To execute the program:

- Set 8 memory address switches to address of first instruction.
- Press Look to set pc to first instruction.
- Press Run to repeat fetch-execute cycle until halt opcode.

Fetch-execute cycle:

- Fetch: get instruction from memory.
- Execute: update pc, move data to or from memory and registers, perform calculations.



Flow Control

Flow Control

- To harness the power of TOY, need loops and conditionals.
- Manipulate **`pc`** to control program flow.

```
if (boolean expression) {
     statement 1;
     statement 2;
}
```

```
while (boolean expression) {
     statement 1;
     statement 2;
}
```

Branch if zero [opcode C]

- Changes **`pc`** depending on whether value of some register is zero.
- Used to implement: **`for`**, **`while`**, **`if-else`**.

Branch if positive [opcode D]

- Changes **`pc`** depending on whether value of some register is positive.
- Used to implement: **`for`**, **`while`**, **`if-else`**.

Penn Engineering

An Example: Multiplication

Multiply: Given integers `a` and `b`, compute `c = a × b`.

TOY multiplication: No direct support in TOY hardware.

Brute-force multiplication algorithm:

- Initialize `c` to 0.
- Add `b` to `c`, `a` times.

```
int a = 3;
int b = 9;
int c = 0;

while (a != 0) {
   c = c + b;
   a = a - 1;
}
```

brute force multiply in Java

Issues ignored: Slow, overflow, negative numbers.

Penn Engineering

Multiply

```
0A: 0003    3   ← inputs
0B: 0009    9
0C: 0000    0   ← output

0D: 0000    0   ← constants
0E: 0001    1


10: 8A0A    RA ← mem[0A]            a = 3;
11: 8B0B    RB ← mem[0B]            b = 9;
12: 8C0D    RC ← mem[0D]            c = 0;

13: 810E    R1 ← mem[0E]            always 1

14: CA18    if (RA == 0) pc ← 18    while (a != 0) {
15: 1CCB    RC ← RC + RB               c = c + b;
16: 2AA1    RA ← RA - R1               a = a - 1;
17: C014    pc ← 14                 }

18: 9C0C    mem[0C] ← RC
19: 0000    halt
```

loop

multiply.toy

Step-By-Step Trace

			R1	RA	RB	RC
10:	8A0A	RA ← mem[0A]		0003		
11:	8B0B	RB ← mem[0B]			0009	
12:	8C0D	RC ← mem[0D]				0000
13:	810E	R1 ← mem[0E]	0001			
14:	CA18	if (RA == 0) pc ← 18				
15:	1CCB	RC ← RC + RB				0009
16:	2AA1	RA ← RA – R1		0002		
17:	C014	pc ← 14				
14:	CA18	if (RA == 0) pc ← 18				
15:	1CCB	RC ← RC + RB				0012
16:	2AA1	RA ← RA – R1		0001		
17:	C014	pc ← 14				
14:	CA18	if (RA == 0) pc ← 18				
15:	1CCB	RC ← RC + RB				001B
16:	2AA1	RA ← RA – R1		0000		
17:	C014	pc ← 14				
14:	CA18	if (RA == 0) pc ← 18				
18:	9C0C	mem[0C] ← RC				
19:	0000	halt				

multiply.toy

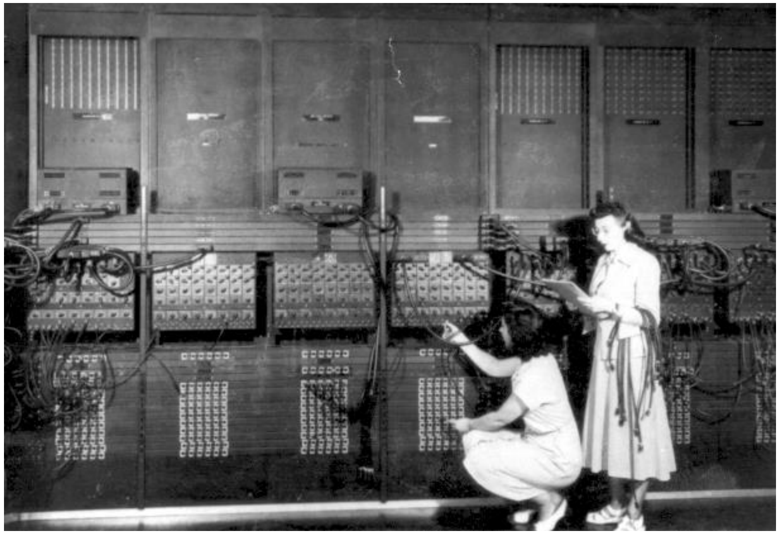
Penn Engineering

A Little History

Electronic Numerical Integrator and Calculator (ENIAC):

- First widely known general purpose electronic computer.
- Conditional jumps, programmable.
- Programming: change switches and cable connections.
- Data: enter numbers using punch cards.

John Mauchly (left) and J. Presper Eckert (right)
http://cs.swau.edu/~durkin/articles/history_computing.html



ENIAC, Ester Gerston (left), Gloria Gordon (right)
US Army photo: http://ftp.arl.mil/ftp/historic-computers

Penn Engineering

Standard Output

- Writing to memory location `FF` sends one word to TOY stdout.
- Ex. `9AFF` writes the integer in register `A` to stdout.

```
00: 0000   0
01: 0001   1

10: 8A00   RA ← mem[00]             a = 0
11: 8B01   RB ← mem[01]             b = 1
                                    do {
12: 9AFF   write RA to stdout          print a
13: 1AAB   RA ← RA + RB                a = a + b
14: 2BAB   RB ← RA - RB                b = a - b
15: DA12   if (RA > 0) goto 12      } while (a > 0)
16: 0000   halt
```

`fibonacci.toy`

```
0000
0001
0001
0002
0003
0005
0008
000D
0015
0022
0037
0059
0090
00E9
0179
0262
03DB
063D
0A18
1055
1A6D
2AC2
452F
6FF1
```

Standard Input

- Loading from memory address `FF` loads one word from stdin.
- Ex. `8AFF` reads an integer from stdin and puts it in register `A`.

Ex: read in a sequence of integers and print their sum.

- In Java, stop reading when EOF.
- In TOY, stop reading when user enters `0000`.

```
while (!StdIn.isEmpty()) {
   a = StdIn.readInt();
   sum = sum + a;
}
StdOut.println(sum);
```

```
00: 0000    0

10: 8C00    RC <- mem[00]
11: 8AFF    read RA from stdin
12: CA15    if (RA == 0) pc ← 15
13: 1CCA    RC ← RC + RA
14: C011    pc ← 11
15: 9CFF    write RC
16: 0000    halt
```

```
00AE
0046
0003
0000
00F7
```

Penn Engineering

An Efficient Multiplication Algorithm

Inefficient multiply:

- Brute force multiplication algorithm loops a times.
- In worst case, 65,535 additions!

"Grade-school" multiplication:

- Always 16 additions to multiply 16-bit integers.

```
Decimal                          Binary

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

Binary Multiplication

Grade school binary multiplication algorithm to compute $c = a \times b$.

- 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	a
			*	1	1	0	1	b
				1	0	1	1	a << 0
			0	0	0	0		
		1	0	1	1			a << 2
	1	0	1	1				a << 3
1	0	0	0	1	1	1	1	c

Implement with built-in TOY shift instructions.

b_i = i^{th} bit of b

```
int c = 0;
for (int i = 15; i >= 0; i--)
   if (((b >> i) & 1) == 1)
      c = c + (a << i);
```

Shift Left

Shift Left [opcode 5]

- Move bits to the left, padding with zeros as needed.
  - 1234_{16} << 7_{16} = $1A00_{16}$

discard

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

<< 7

pad with 0's

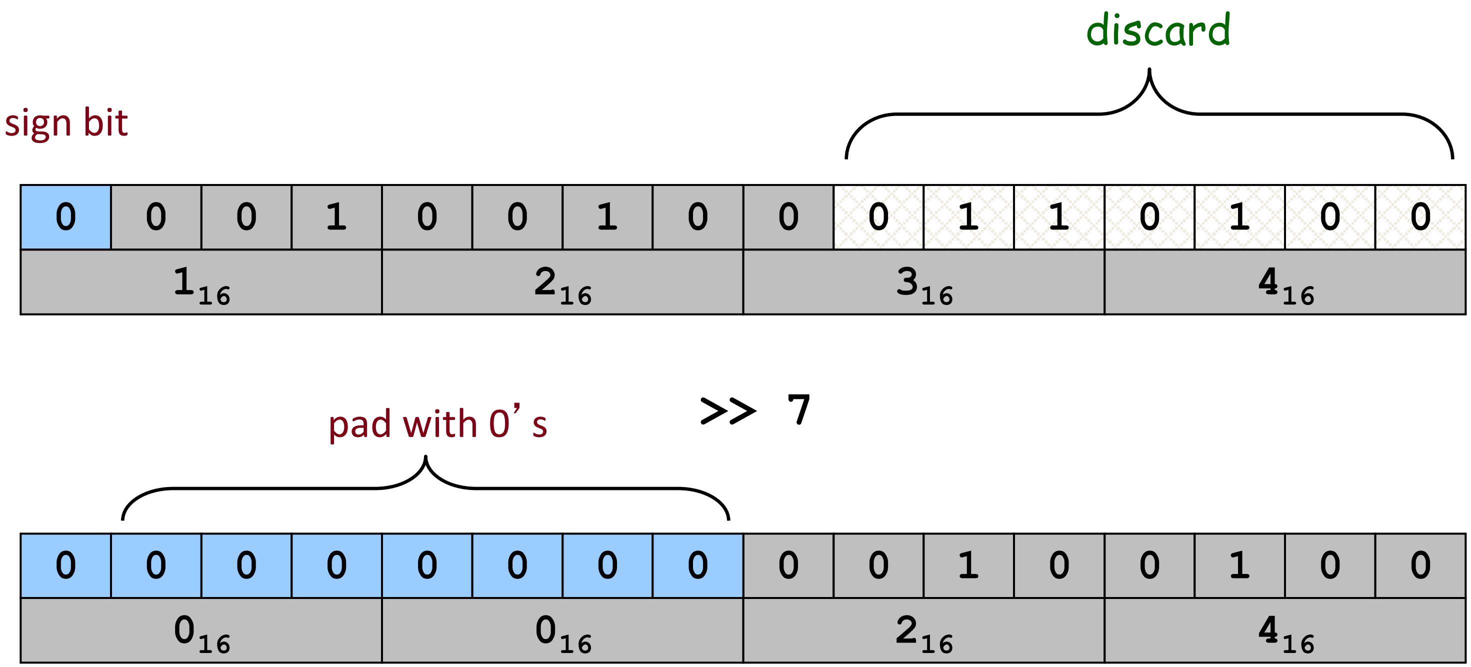
0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0
1_{16}				A_{16}				0_{16}				0_{16}			

Penn Engineering

Shift Right

Shift Right [opcode 6]

- Move bits to the right, padding with sign bit as needed.
  - 1234_{16} >> 7_{16} = 0024_{16}



Penn Engineering

Bitwise AND

Logical AND [opcode B]

- Logic operations are BITWISE.
  - 002416 & 000116 = 000016

x	y	&
0	0	0
0	1	0
1	0	0
1	1	1

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

&

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

=

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0_{16}				0_{16}				0_{16}				0_{16}			

Penn Engineering

Shifting and Masking

Shift and mask: get the 7th bit of 1234

- Compute 1234_{16} >> 7_{16} = 0024_{16}
- Compute 0024_{16} & 1_{16} = 0_{16}

<table>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="4">1_{16}</td><td colspan="4">2_{16}</td><td colspan="4">3_{16}</td><td colspan="4">4_{16}</td></tr>
</table>

>> 7

<table>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="4">0_{16}</td><td colspan="4">0_{16}</td><td colspan="4">2_{16}</td><td colspan="4">4_{16}</td></tr>
</table>

&

<table>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td colspan="4">0_{16}</td><td colspan="4">0_{16}</td><td colspan="4">0_{16}</td><td colspan="4">1_{16}</td></tr>
</table>

=

<table>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="4">0_{16}</td><td colspan="4">0_{16}</td><td colspan="4">0_{16}</td><td colspan="4">0_{16}</td></tr>
</table>

Binary Multiplication

```
0A: 0003     3     inputs
0B: 0009     9
0C: 0000     0     output
0D: 0000     0
0E: 0001     1     constants
0F: 0010    16

10: 8A0A    RA ← mem[0A]        a
11: 8B0B    RB ← mem[0B]        b
12: 8C0D    RC ← mem[0D]        c = 0
13: 810E    R1 ← mem[0E]        always 1
14: 820F    R2 ← mem[0F]        i = 16          16 bit words

                     do {
loop
15: 2221    R2 ← R2 -  R1          i--
16: 53A2    R3 ← RA << R2          a << i
17: 64B2    R4 ← RB >> R2          b >> i
branch
18: 3441    R4 ← R4 &  R1          b_i = i^th bit of b
19: C41B    if (R4 == 0) goto 1B       if b_i is 1
1A: 1CC3    RC ← RC + R3              add a << i to sum
1B: D215    if (R2 > 0) goto 15     } while (i > 0);

1C: 9C0C    mem[0C] ← RC
```

multiply-fast.toy

Bitwise XOR

Bitwise XOR [opcode 4]

- Logic operations are BITWISE.
  - 1234_{16} ^ $FAD2_{16}$ = $E8E6_{16}$

x	y	^
0	0	0
0	1	1
1	0	1
1	1	0

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}			

Shift Right (Sign Extension)

Shift Right [opcode 6]

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

discard

sign bit

1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	0
F_{16}				F_{16}				C_{16}				A_{16}			

$>> 2$

pad with 1s

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

Load Address (a.k.a. Load Constant)

Load Address [opcode 7]

- Loads an 8-bit integer into a register.
- `7A30` means load the value 30 into register `A`.

```
a = 0x30;
```

Java code

Applications.

register stores "pointer" to a memory cell

- Load a small constant into a register.
- Load a 8-bit memory address into a register.

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

Arrays in TOY

TOY main memory is a giant array

- Can access memory cell 30 using load and store.
- 8C30 means load mem[30] into register C.
- Goal: access memory cell i where i is a variable.

...	...
30	0000
31	0001
32	0001
33	0002
34	0003
35	0005
36	0008
37	000D
...	...

TOY memory

Load Indirect [opcode A]

- AC06 means load mem[R6] into register C.

variable index

Store Indirect [opcode B]

- BC06 means store contents of register C into mem[R6].

```
for (int i = 0; i < N; i++)
    a[i] = StdIn.readInt();

for (int i = 0; i < N; i++)
    StdOut.println(a[N-i-1]);
```

TOY Implementation of Reverse

TOY implementation of reverse:

- Read in a sequence of integers and store in memory 30, 31, 32, … until reading 0000.
- Print sequence in reverse order.

```
10: 7101   R1 ← 0001                  constant 1
11: 7A30   RA ← 0030                  a[]
12: 7B00   RB ← 0000                  n

                                      while(true) {
13: 8CFF   read RC                        c = StdIn.readInt();
14: CC19   if (RC == 0) goto 19           if (c == 0) break;
15: 16AB   R6 ← RA + RB                   memory address of a[n]
16: BC06   mem[R6] ← RC                   a[n] = c;
17: 1BB1   RB ← RB + R1                   n++;
18: C013   goto 13                    }
```

read in the data

Penn Engineering

TOY Implementation of Reverse

TOY implementation of reverse:

- Read in a sequence of integers and store in memory 30, 31, 32, …
  until reading 0000.
- Print sequence in reverse order.

```
10: 7101   R1 ← 0001                  constant 1
11: 7A30   RA ← 0030                  a[]
12: 7B00   RB ← 0000                  n

                                      while(true) {
13: 8CFF   read RC                        c = StdIn.readInt();
14: CC19   if (RC == 0) goto 19           if (c == 0) break;
15: 16AB   R6 ← RA + RB                   memory address of a[n]
16: BC06   mem[R6] ← RC                   a[n] = c;
17: 1BB1   RB ← RB + R1                   n++;
18: C013   goto 13                    }
```

print in reverse order

Unsafe Code at any Speed

Q. What happens if we make array start at `00` instead of `30`?

A. Self modifying program; can overflow buffer and run arbitrary code!

```
10: 7101   R1 ← 0001                 constant 1
11: 7A00   RA ← 0000                 a[]
12: 7B00   RB ← 0000                 n

                                     while(true) {
13: 8CFF   read RC                       c = StdIn.readInt();
14: CC19   if (RC == 0) goto 19          if (c == 0) break;
15: 16AB   R6 ← RA + RB                  address of a[n]
16: BC06   mem[R6] ← RC                  a[n] = c;
17: 1BB1   RB ← RB + R1                  n++;
18: C013   goto 13                   }
```

```
% more crazy8.txt
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
8888 8810
98FF C011
```

What If We Lose Control (in C or C++)?

Buffer overflow:

- Array buffer[] has size 100.
- User might enter 200 characters.
- Might lose control of machine behavior.

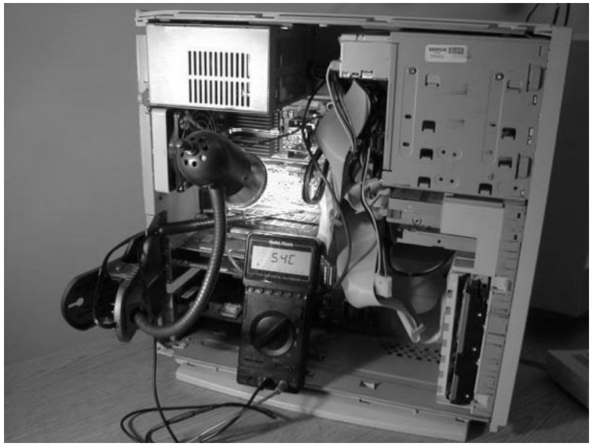
```
#include <stdio.h>
int main(void) {
   char buffer[100];
   scanf("%s", buffer);
   printf("%s\n", buffer);
   return 0;
}
```

unsafe C program

Consequences: Viruses and worms.

Java enforces security:

- Type safety.
- Array bounds checking.
- Not foolproof.



shine 50W bulb at DRAM
[Appel-Govindavajhala '03]

Penn Engineering

Buffer Overflow Attacks

Stuxnet Worm [July 2010]

- Step 1. Natanz centrifuge fuel-refining plant employee plugs in USB flash drive.
- Step 2. Data becomes code by exploiting Window buffer overflow; machine is 0wned.
- Step 3. Uranium enrichment in Iran stalled.

More buffer overflow attacks: Duqu, Flame, Morris worm, Code Red, SQL Slammer, iPhone unlocking, Xbox softmod, JPEG of death, …

Lesson:

- Not easy to write error-free software.
- Embrace Java security features.
- Keep your OS patched.

Penn Engineering

Dumping

Q. Work all day to develop operating system in `mem[10]` to `mem[FF]`.
How to save it?

A. Write dump.toy and run it to dump contents of memory onto tape.

```
00: 7001    R1 ← 0001
01: 7210    R2 ← 0010                  i = 10
02: 73FF    R3 ← 00FF
                                       do {
03: AA02    RA ← mem[R2]                   a = mem[i]
04: 9AFF    write RA                       print a
05: 1221    R2 ← R2 + R1                   i++
06: 2432    R4 ← R3 - R2
07: D403    if (R4 > 0) goto 03        } while (i < 255)
08: 0000    halt
```

dump.toy

Penn Engineering

Booting

Q. How do you get it back?

A. Run boot.toy and to read `mem[10]` to `mem[FF]` from tape.

```
00: 7001    R1 ← 0001
01: 7210    R2 ← 0010                 i = 10
02: 73FF    R3 ← 00FF
                                      do {
03: 8AFF    read RA                      read a
04: BA02    mem[R2] ← RA                 mem[i] = a
05: 1221    R2 ← R2 + R1                 i++
06: 2432    R4 ← R3 - R2
07: D403    if (R4 > 0) goto 0        } while (i < 255)
08: 0000    halt
```

boot.toy

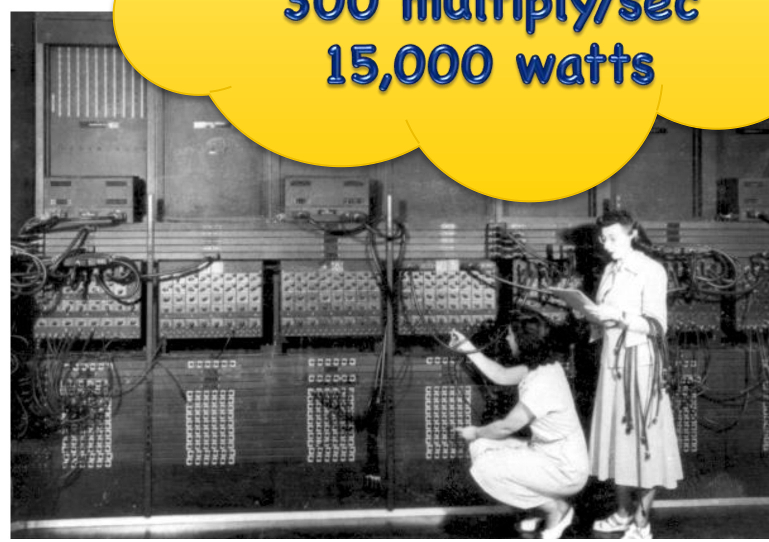
A Little History

Electronic Numerical Integrator and Calculator (ENIAC):

- First widely known general purpose el
- Conditional jumps, programmable.
- Programming: change switches a
- Data: enter numbers using punch

30 tons
30 x 50 x 8.5 ft
17,468 vacuum tubes
300 multiply/sec
15,000 watts

John Mauchly (left) and J. Presper Eckert (right)
http://cs.swau.edu/~durkin/articles/history_computing.html



ENIAC, Ester Gerston (left), Gloria Gordon (right)
US Army photo: http://ftp.arl.mil/ftp/historic-computers

Penn Engineering

Basic Characteristics of TOY Machine

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, 1944]

- Data and program encoded in binary.
- Data and program stored in same memory.
- Can change program without rewiring.

Outgrowth of Alan Turing's work

All modern computers are general-purpose computers and have same (von Neumann) architecture.

John von Neumann

Maurice Wilkes (left)
EDSAC (right)

Penn Engineering

TOY Reference Card

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Format 1	opcode				dest d				source s				source t			
Format 2	opcode				dest d				addr							

#	Operation	Fmt	Pseudocode
0:	halt	1	exit(0)
1:	add	1	R[d] ← R[s] + R[t]
2:	subtract	1	R[d] ← R[s] - R[t]
3:	and	1	R[d] ← R[s] & R[t]
4:	xor	1	R[d] ← R[s] ^ R[t]
5:	shift left	1	R[d] ← R[s] << R[t]
6:	shift right	1	R[d] ← R[s] >> R[t]
7:	load addr	2	R[d] ← addr
8:	load	2	R[d] ← mem[addr]
9:	store	2	mem[addr] ← R[d]
A:	load indirect	1	R[d] ← mem[R[t]]
B:	store indirect	1	mem[R[t]] ← R[d]
C:	branch zero	2	if (R[d] == 0) pc ← addr
D:	branch positive	2	if (R[d] > 0) pc ← addr
E:	jump register	2	pc ← R[d]
F:	jump and link	2	R[d] ← pc; pc ← addr

Register 0 always reads 0.
Loads from mem[FF] from stdin.
Stores to mem[FF] to stdout.

16-bit registers.
16-bit memory.
8-bit program counter.