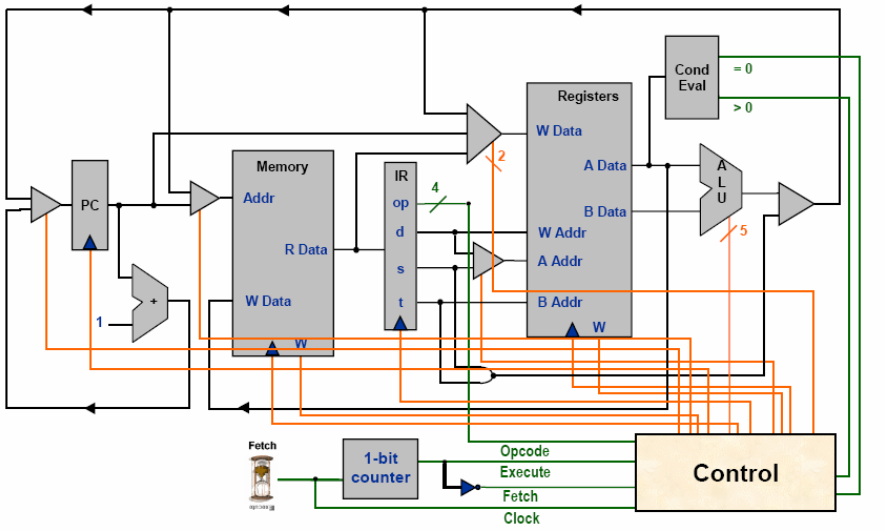
Build a TOY



Machine "Core" Dump

Machine contents 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

next instruction

data

program

variables

Main 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

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

Useful TOY "Idioms"

Jump absolute.

- Jump to a fixed memory address.
  - branch if zero with destination
  - register 0 is always 0

```
17: C014   pc ← 14
```

Register assignment.

- No instruction that transfers contents of one register into another.
- Pseudo-instruction that simulates assignment:
  - add with register 0 as one of two source registers

```
17: 1230   R[2] ← R[3]
```

No-op.

- Instruction that does nothing.
- Plays the role of whitespace in C programs.
  - numerous other possibilities!

```
17: 1000   no-op
```

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.

Applications.

- Load a small constant into a register.
- Load a 8-bit memory address into a register.
  - register stores "pointer" to a memory cell

```
a = 30;
```

Java code

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							

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
11: 8B0B   RB ← mem[0B]              b
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

Function Call: A Failed Attempt

Goal: x × y × z.

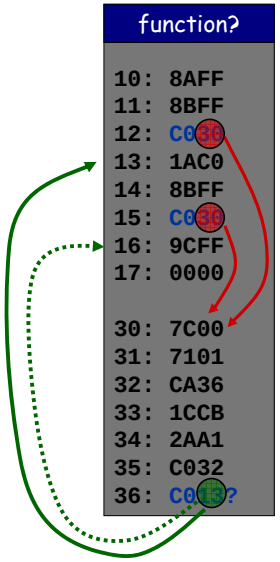
- Need two multiplications: x × y, (x × y) × z.
  - Solution 1: write multiply code 2 times.
  - Solution 2: write a TOY function.

A failed attempt:

- Write multiply loop at 30-36.
- Calling program agrees to store arguments in registers A and B.
- Function agrees to leave result in register C.
- Call function with jump absolute to 30.
- Return from function with jump absolute.

Reason for failure.

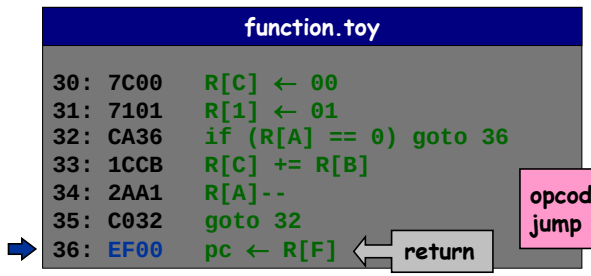
- Need to return to a VARIABLE memory address.

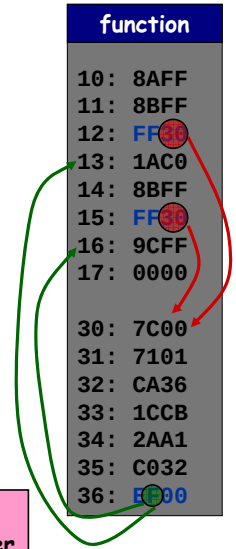


Multiplication Function

Calling convention.

- Jump to line 30.
- Store a and b in registers A and B.
- Return address in register F.
- Put result c = a × b in register C.
- Register 1 is scratch.
- Overwrites registers A and B.





Multiplication Function Call

Client program to compute x × y × z.

- Read x, y, z from standard input.
- Note: PC is incremented before instruction is executed.
  - value stored in register F is correct return address

function.toy (cont)

```
10: 8AFF   read R[A]              x
11: 8BFF   read R[B]              y
12: FF30   R[F] ← pc; goto 30     x * y
13: 1AC0   R[A] ← R[C]            (x * y)
14: 8BFF   read R[B]              z
15: FF30   R[F] ← pc; goto 30     (x * y) * z
16: 9CFF   write R[C]
17: 0000   halt
```

opcode F jump and link

R[F] ← 13

R[F] ← 16

Function Call: One Solution

Contract between calling program and function:

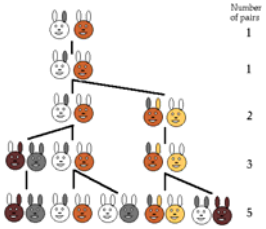
- Calling program stores function parameters in specific registers.
- Calling program stores return address in a specific register.
  - jump-and-link
- Calling program sets PC to address of function.
- Function stores return value in specific register.
- Function sets PC to return address when finished.
  - jump register

What if you want a function to call another function?

- Use a different register for return address.
- More general: store return addresses on a stack.

Fibonacci Numbers

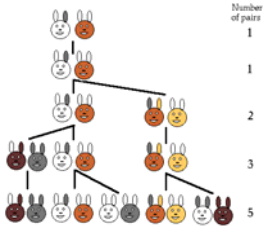
Fibonacci rabbits: Beginning with a single pair of rabbits, if every month each productive pair bears a new pair (which become productive when one month old), how many rabbits after n months?



L. P. Fibonacci
(1170 - 1250)

Fibonacci Numbers

Fibonacci rabbits: Beginning with a single pair of rabbits, if every month each productive pair bears a new pair (which become productive when one month old), how many rabbits after n months?



L. P. Fibonacci
(1170 - 1250)

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

Standard Output

Standard output.

- Writing to memory location FF sends one word to TOY stdout.
- 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
                                  while(a > 0) {
12: 9AFF   print RA                  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    }
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

Standard input.

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

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

- In TOY, stop reading when user enters 0000.

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

```
00: 0000   0

10: 8C00   RC <- mem[00]
11: 8AFF   read RA
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
```


Standard Input and Output: Implications

Standard input and output enable you to:

- Put information from real world into machine.
- Get information out of machine.
- Process more information than fits in memory.
- Interact with the computer while it is running.

Information can be instructions!

- Booting a computer.
- Sending programs over the Internet.
- Sending viruses over the Internet.

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.

Load indirect. (opcode A) — a variable index

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

Store indirect. (opcode B) — a variable index

- 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++)
   System.out.println(a[N-i-1]);
```

Reverse.java

TOY Implementation of Reverse

TOY implementation of reverse.

- Read in a sequence of integers and store in memory 30, 31, 32, ...
- Stop reading if 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                 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

TOY Implementation of Reverse

TOY implementation of reverse.

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

```
19: CB20  if (RB == 0) goto 20      while (n > 0) {
1A: 16AB  R6 ← RA + RB                 address of a[n]
1B: 2661  R6 ← R6 - R1                 address of a[n-1]
1C: AC06  RC ← mem[R6]                 c = a[n-1];
1D: 9CFF  write RC                     System.out.println(c);
1E: 2BB1  RB ← RB - R1                 n--;
1F: C019  goto 19                   }
20: 0000  halt
```

print in reverse order

TOY assembly

- Data directives
- A DW n: initialize a variable A as n
- B DUP n: reserve n words (n is DEC)
- Support two types of literals, decimal and hexadecimal (0x)
- Label begins with a letter
- Comment begins with ;
- Case insensitive
- Program starts with the first instruction it meets
- Some tricks to handle the starting address 0x10

opcode	mnemonic	syntax
0	hlt	hlt
1	add	add rd, rs, rt
2	sub	sub rd, rs, rt
3	and	and rd, rs, rt
4	xor	xor rd, rs, rt
5	shl	shl rd, rs, rt
6	shr	shr rd, rs, rt
7	lda	lda rd, addr
8	ld	ld rd, addr
9	st	st rd, addr
A	ldi	ldi rd, rt
B	sti	sti rd, rt
C	bz	bz rd, addr
D	bp	bp rd, addr
E	jr	jr rd
F	jl	jl rd, addr

Assembler

Assembler's task:

- Convert mnemonic operation codes to their machine language equivalents
- Convert symbolic operands to their equivalent machine addresses
- Build machine instructions in proper format
- Convert data constants into internal machine representations (data formats)
- Write object program and the assembly listing

Forward Reference

Definition
- A reference to a label that is defined later in the program

Solution
- Two passes
  - First pass: scan the source program for label definition, address accumulation, and address assignment
  - Second pass: perform most of the actual instruction translation
- One-pass assemblers are used when
  - it is necessary or desirable to avoid a second pass over the source program
  - the external storage for the intermediate file between two passes is slow or is inconvenient to use
  - Main problem: forward references to both data and instructions
  - One simple way to eliminate this problem: require that all areas be defined before they are referenced. It is possible, although inconvenient, to do so for data items. Forward jump to instruction items cannot be easily eliminated.

Assembly version of REVERSE

```
int A[32];              A       DUP     32              10: C020

                                lda     R1, 1           20: 7101
                                lda     RA, A           21: 7A00
i=0;                            lda     RC, 0           22: 7C00
Do {
  RD=stdin;             read    ld      RD, 0xFF        23: 8DFF
  if (RD==0) break;             bz      RD, exit        24: CD29
                                add     R2, RA, RC      25: 12AC
  A[i]=RD;                      sti     RD, R2          26: BD02
  i=i+1;                        add     RC, RC, R1      27: 1CC1
} while (1);                    bz      R0, read        28: C023

printr();               exit    jl      RF, printr      29: FF2B
                                hlt                     2A: 0000
```


Assembly version of REVERSE

```
printr()                ; print reverse
{                       ; array address (RA)
  do {                  ; number of elements (RC)
    i=i-1;      printr  sub     RC, RC, R1      2B: 2CC1
                        add     R2, RA, RC      2C: 12AC
                        ldi     RD, R2          2D: AD02
    print A[i];         st      RD, 0xFF        2E: 9DFF
  } while (i>=0);       bp      RC, printr      2F: DC2B
                        bz      RC, printr      30: CC2B
  return;       return  jr      RF              31: EF00
}
```

toyasm < reverse.asm > reverse.toy

Horner's Method

Goal: 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, multiply by a; compute x^2, multiply by b, … (cumbersome, inefficient)

Efficient algorithm (Horner's method):
- Rewrite $ax^3 + bx^2 + cx + d$ as $((a x + b) x + c) x + d$.
- Does polynomial evaluation for arbitrary x.
- Many applications (e.g., convert from decimal to hex).
- One raison d'être for early machines.

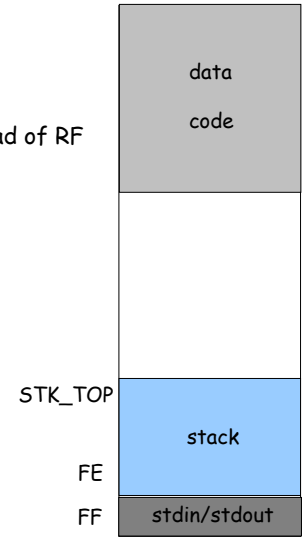
Converts from decimal to hex: $2397_{10} = 95D_{16}$.

stack

```
STK_TOP         DW      0xFF

; these procedures will use R8, R9
; return address is assumed to be in RE, instead of RF
; it is the only exception

; push RF into stack
push    lda     R8, 1
        ld      R9, STK_TOP
        sub     R9, R9, R8
        st      R9, STK_TOP
        sti     RF, R9
        jr      RE
```



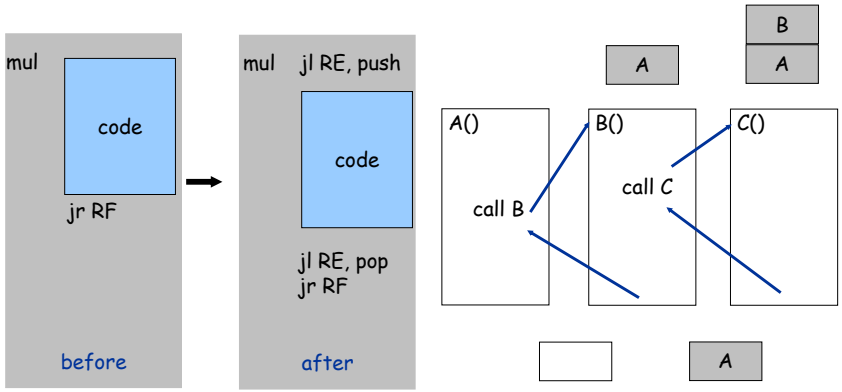
stack

```
; pop and return [top] to RF
pop     lda     R8, 0xFF
        ld      R9, STK_TOP
        sub     R8, R8, R9
        bz      R8, popexit
        ldi     RF, R9
        lda     R8, 1
        add     R9, R9, R8
        st      R9, STK_TOP
popexit jr      RE

; the size of the stack, the result is in R9
stksize lda     R8, 0xFF
        ld      R9, STK_TOP
        sub     R9, R8, R9
        jr      RE
```


Procedure prototype

With stack, now, the procedure prototype is changed. It allows us to have a deeper call graph by using stack.



Linking

We will write a procedure horner, which will call multiply. Since multiply will be used by many applications, could we make multiply a library?

Toyasm has an option to generate an object file so that it can be later linked with other object files.

That is why we need linking. Write a subroutine mul3 which multiplies three numbers in RA, RB, RC together and place the result in RD.
Three files:

- stack.obj: implementation of stack, needed for procedure
- mul.obj: implementation of multiplication.
- multest.obj: main program and procedure of mul3

toylink multest.obj mul.obj stack.obj > multest.toy

object file (multest.obj)

```
A       DW      3
B       DW      4
C       DW      5

; calculate A*B*C
main    ld      RA, A
        ld      RB, B
        ld      RC, C
        jl      RF, mul3
        st      RD, 0xFF
        hlt

; RD=RA*RB*RC
; return address is in RF
mul3    jl      RE, push

        lda     RD, 0
        add     RD, RC, R0
        jl      RF, mul
        add     RA, RC, R0
        add     RB, RD, R0
        jl      RF, mul
        add     RD, RC, R0

        jl      RE, pop
        jr      RF
```


object file (mul.obj)

```
SIXTEEN         DW      16

; multiply RC=RA*RB
; return address is in RF
; it will modify R2, R3 and R4 as well
mul     jl      RE, push

        lda     RC, 0
        lda     R1, 1
        ld      R2, SIXTEEN
m_loop  sub     R2, R2, R1
        shl     R3, RA, R2
        shr     R4, RB, R2
        and     R4, R4, R1
        bz      R4, m_end
        add     RC, RC, R3
m_end   bp      R2, m_loop

        jl      RE, pop
        jr      RF
```

```
// size 29
// export 4
// SIXTEEN 0x00
// mul 0x10
// m_loop 0x14
// m_end 0x1A
// literal 2 17 18
// lines 14
00: 0010
10: FE00
11: 7C00
12: 7101
13: 8200
14: 2221
15: 53A2
16: 64B2
17: 3441
18: C41A
19: 1CC3
1A: D214
1B: FE00
1C: EF00
// import 2
// push 1 0x10
// pop 1 0x1B
```

export table

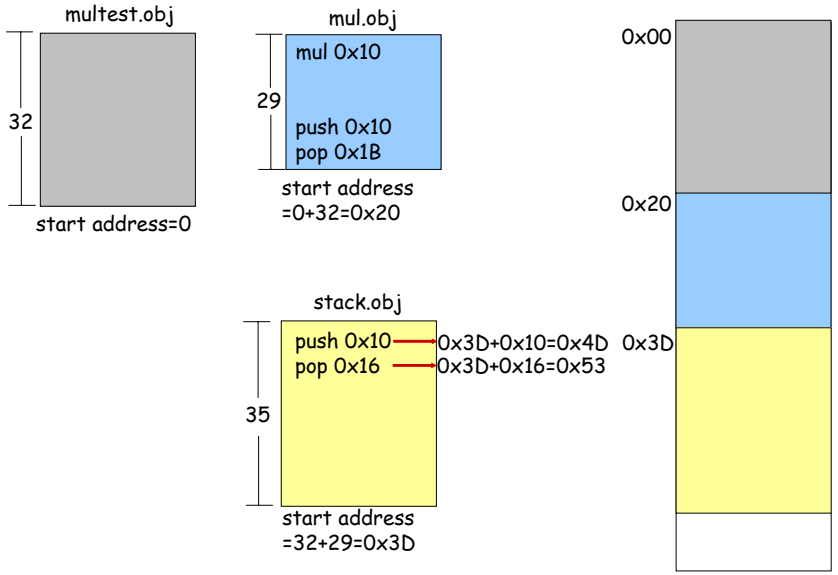
These are literals. No need to relocate

need to fill in address of push once we know it

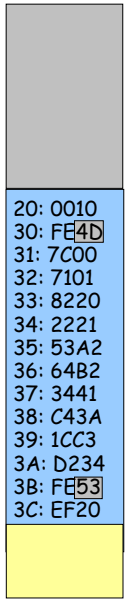
need to fill in address of pop once we know it

import table

Linking



Resolve external symbols




```
// size 29
// export 4
// SIXTEEN 0x00
// mul 0x10
// m_loop 0x14
// m_end 0x1A
// literal 2 17 18
// lines 14
00: 0010
10: FE00
11: 7C00
12: 7101
13: 8200
14: 2221
15: 53A2
16: 64B2
17: 3441
18: C41A
19: 1CC3
1A: D214
1B: FE00
1C: EF00
// import 2
// push 1 0x10
// pop 1 0x1B
```

export table

These are literals. No need to relocate

need to fill in address of push once we know it

need to fill in address of pop once we know it

import table

Unsafe Code at any Speed

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

- Self modifying program.
- Exploit buffer overrun 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                  }
```

Crazy 8s Input

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


Unsafe Code at any Speed

```
00: 0001        08: 0001
01: 0001        09: 0001
02: 0001        0A: 0001
03: 0001        0B: 0001
04: 0001        0C: 0001
05: 0001        0D: 0001
06: 0001        0E: 0001
07: 0001        0F: 0001

10: 8888
11: 8810
12: 98FF
```

Crazy 8s Input

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

data becomes code!

```
13: C011  goto 11
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                  }
```


What Can Happen When We Lose Control (in C or C++)?

Buffer overrun.

- Array `buffer[]` has size 100.
- User might enter 200 characters.
- Might lose control of machine behavior.
- Majority of viruses and worms caused by similar errors.

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

unsafe C program

Robert Morris Internet Worm.

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

Microsoft Windows JPEG bug. [September, 2004]

- Step 1. User views malicious JPEG in Internet Explorer or Outlook.
- Step 2. Machine is Owned.
- Data becomes code by exploiting buffer overrun in GDI+ library.

Assignment #2

Assigned: 10/23/2006
Due: 11/6/2006

You can write in either TOY assembly or TOY machine code
Part 1 (50%): write a procedure BCD to convert a hexadecimal number into a BCD (Binary-Coded Decimal). The input number is placed in RA. The result should be placed in RB. The return address is in RF. (Hint: you need to implement division)

Part 2 (30%): write a procedure CNT0 to count 0's in an array. The address of the array is placed at RA. The size of the array is specified by RC. The result should be placed in RB. The return address is in RF.

Part 3 (20%): write a program to read a series of numbers specified by the user from stdin until the input is 0x0000. Count the number of 0-bits in the input array and display this number using BCD in stdout.