CS 126 Lecture A2:
TOY Programming

Outline

- **Review and Introduction**
- Data representation
- Dynamic addressing
- Control flow
- TOY simulator
- Conclusions

What We Have Learned About TOY

- What's TOY, what's in it, how to use it.
  - von Neumann architecture
- Data representation
  - Binary and hexadecimal
- TOY instructions
  - Instruction set architecture
- Example TOY programs
  - Simple machine language programming

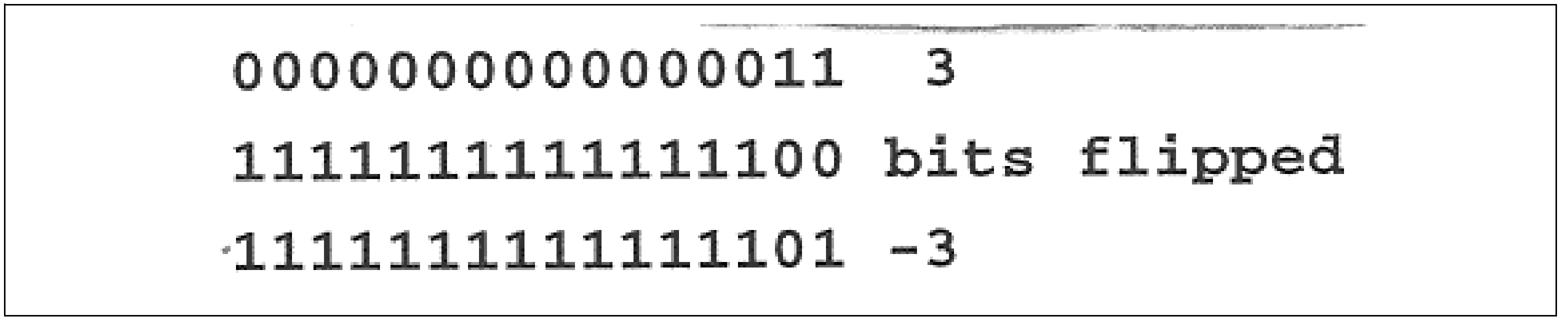
What We Haven’t Learned

- How to represent data types other than positive integers?
- How to represent complex data structures at machine level?
- How to make function calls at machine level?
- What’s the relationship among TOY, C programming, and “real” computers?

Outline

- ~~Review and Introduction~~
- **Data representation**
- Dynamic addressing
- Control flow
- TOY simulator
- Conclusions

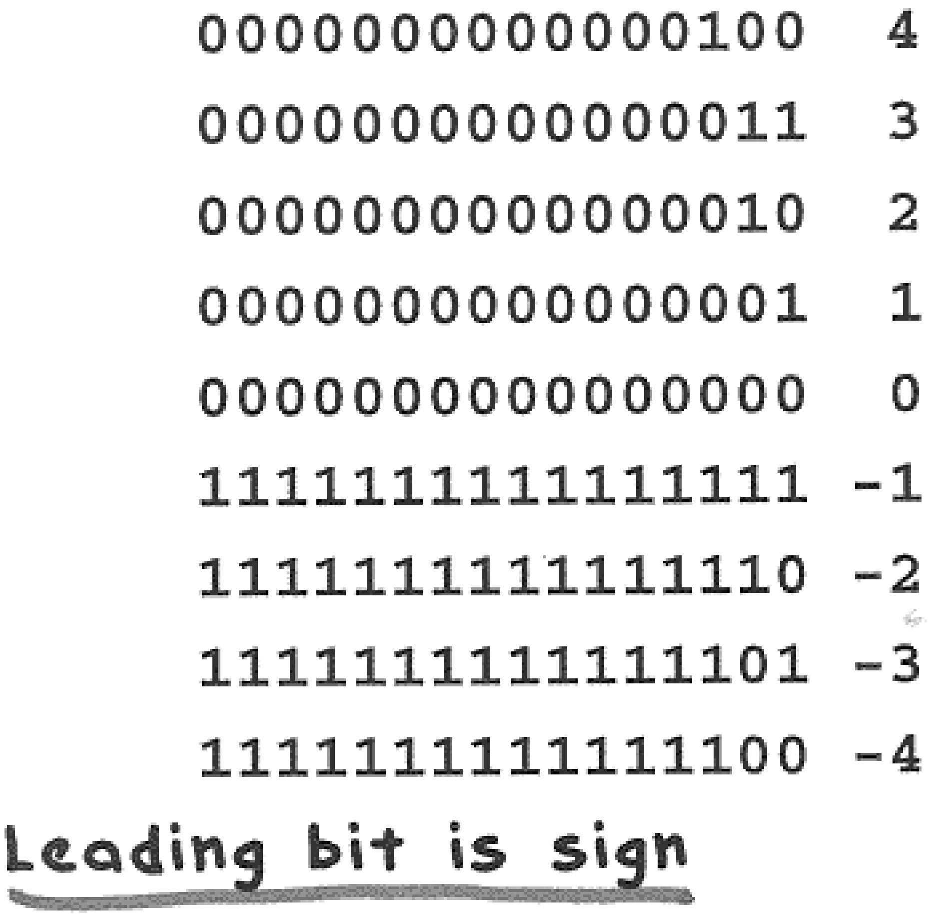
Represent Negative Numbers Using "Two's Complement"



```
0000000000000011  3
1111111111111100 bits flipped
1111111111111101 -3
```

- Represent **-N** with an **n**-bit 2's complement: $2^n - N$
- To calculate **-N**, start with **N**, flip bits, and add **1**

Examples

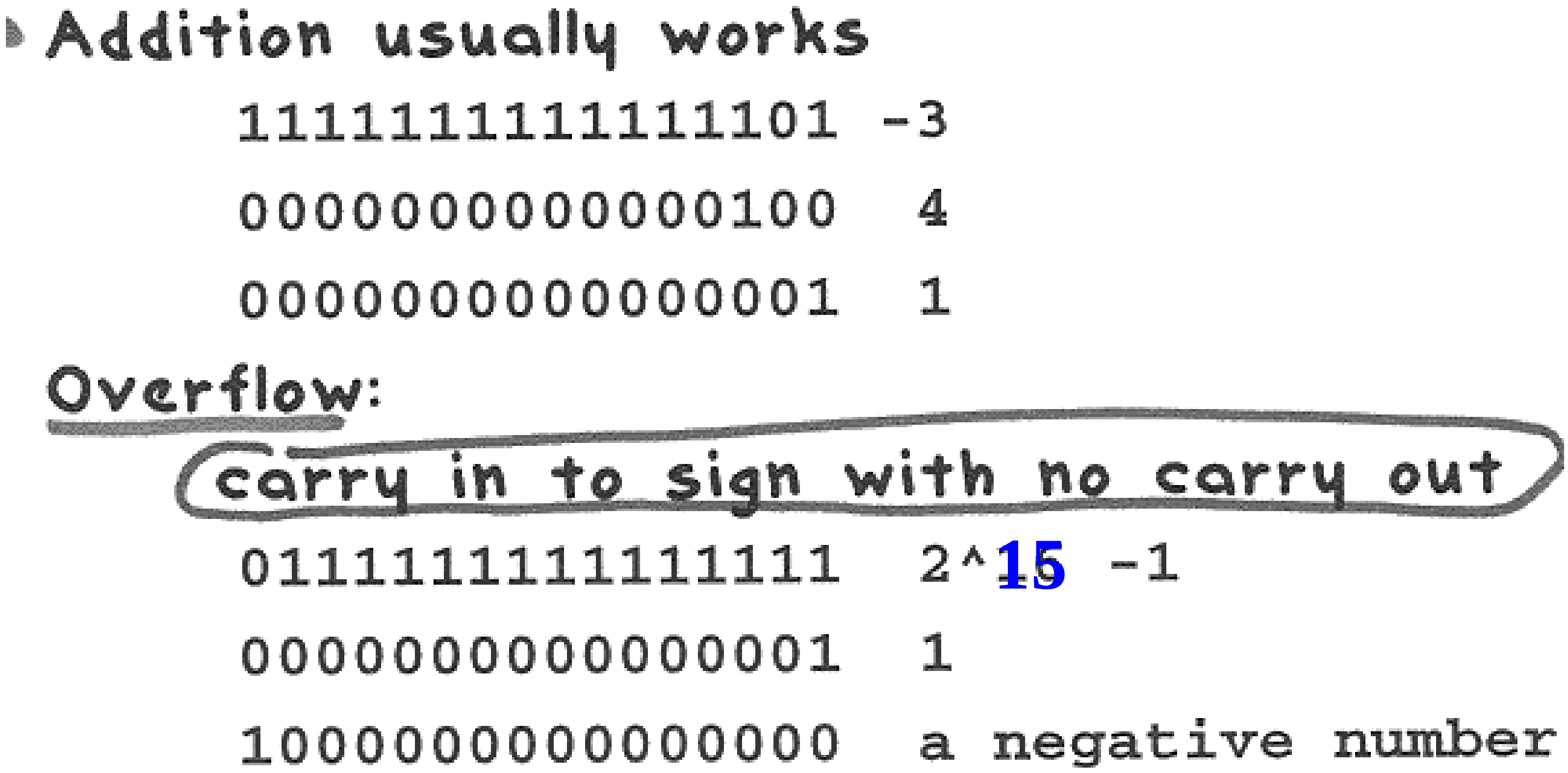


0000000000000100	4
0000000000000011	3
0000000000000010	2
0000000000000001	1
0000000000000000	0
1111111111111111	-1
1111111111111110	-2
1111111111111101	-3
1111111111111100	-4

Leading bit is sign

Arithmetic

- Addition is carried out as if all numbers were positive



- Addition usually works

```
1111111111111101 -3
0000000000000100  4
0000000000000001  1
```

Overflow:

carry in to sign with no carry out

```
0111111111111111  2^15 -1
0000000000000001  1
1000000000000000  a negative number
```

- Subtraction **-N** is done with addition of **N**

Nice and Not-So-Nice Properties

- Nice properties
  - 0 is 0
  - -0 and +0 are the same
- Not-so-nice property
  - Can represent one more negative number than positive numbers
  - With n bits, can represent:

    $2^{n-1} - 1$ positive numbers ($2^{n-1} - 1$ is maximum)

    0

    2^{n-1} negative numbers (-2^{n-1} is minimum)
  - **A2-3 of course reader is wrong! (Replace 16s with 15s)**
- Alternatives other than 2's complement exist

Other Primitive Data Types

big integers: could use “multiple precision”

multiple words per integer

required for multiply, divide?

real numbers: could use “floating point”

like scientific notation

- “double” type, “long long” type (for most compilers)

character strings: could use ASCII code

8 bits/character (packed/unpacked)

Outline

- ~~Review and Introduction~~
- ~~Data representation~~
- **Dynamic addressing**
- Control flow
- TOY simulator
- Conclusions

The Need for Dynamic Addressing

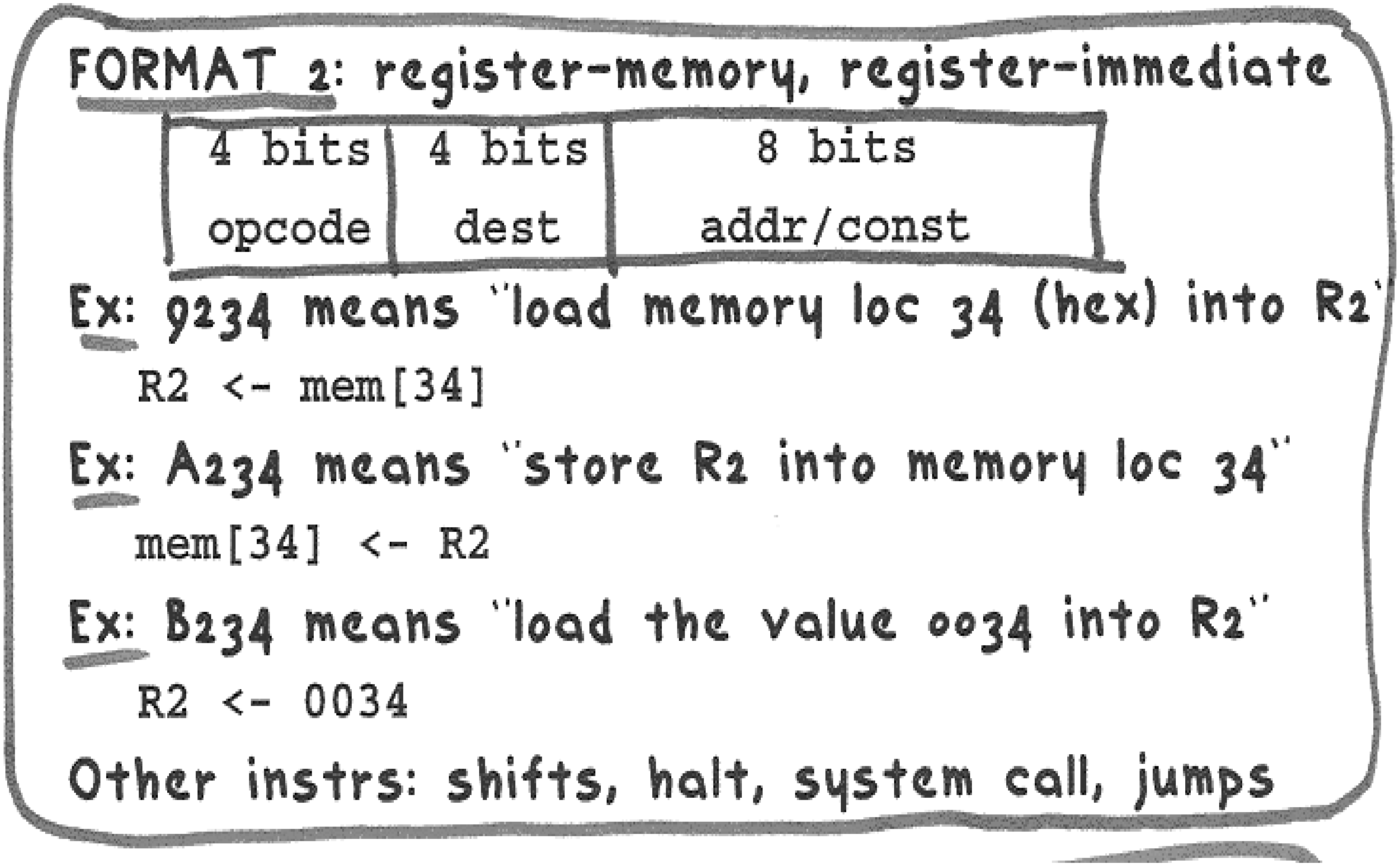
```
int a;          ①
```

```
int a[100];     ②
```

```
int *p;                                    ③
p = (int *) malloc(sizeof *p);
```

- All we have so far: “hard-wired” addresses inside instructions (`R1<-MEM[D0]`)
- Many cases where guessing address at **compile-time** is impossible
  - case 1: possible for compiler to hard-wire address of a
  - case 2: difficult for compiler to hard-wire address of a[i]
  - case 3: impossible for compiler to guess address at p
- Solution:
  - Compute address at **run time**
  - Put address in a register
  - Augment instruction format to use address register

Review: 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”

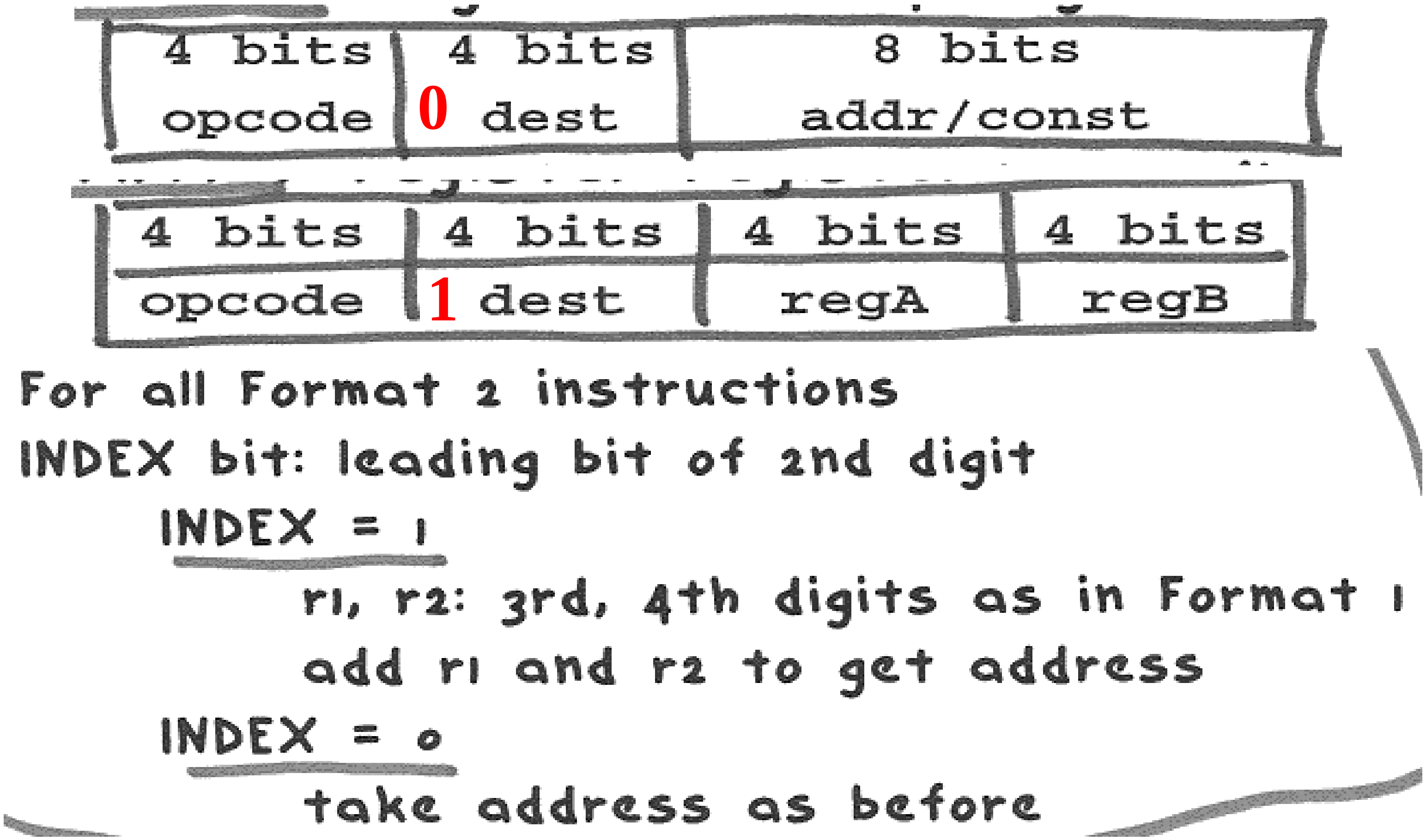
mem[34] <- R2

Ex: B234 means “load the value 0034 into R2”

R2 <- 0034

Other instrs: shifts, halt, system call, jumps

Indexed Addressing



4 bits	4 bits	8 bits
opcode	0 dest	addr/const

4 bits	4 bits	4 bits	4 bits
opcode	1 dest	regA	regB

For all Format 2 instructions

INDEX bit: leading bit of 2nd digit

- INDEX = 1
  - r1, r2: 3rd, 4th digits as in Format 1
  - add r1 and r2 to get address
- INDEX = 0
  - take address as before

- Example: **A923** means **MEM[R[2]+R[3]]<-R[1] (9 is binary 1001)**

Why “Stealing” One Bit is OK

4 bits	4 bits	8 bits
opcode	0 dest	addr/const

4 bits	4 bits	4 bits	4 bits
opcode	1 dest	regA	regB

- We only have 8 registers
- Only three bits are necessary
- But 4 bits allocated to dest register field
- So we can “steal” 1 bit

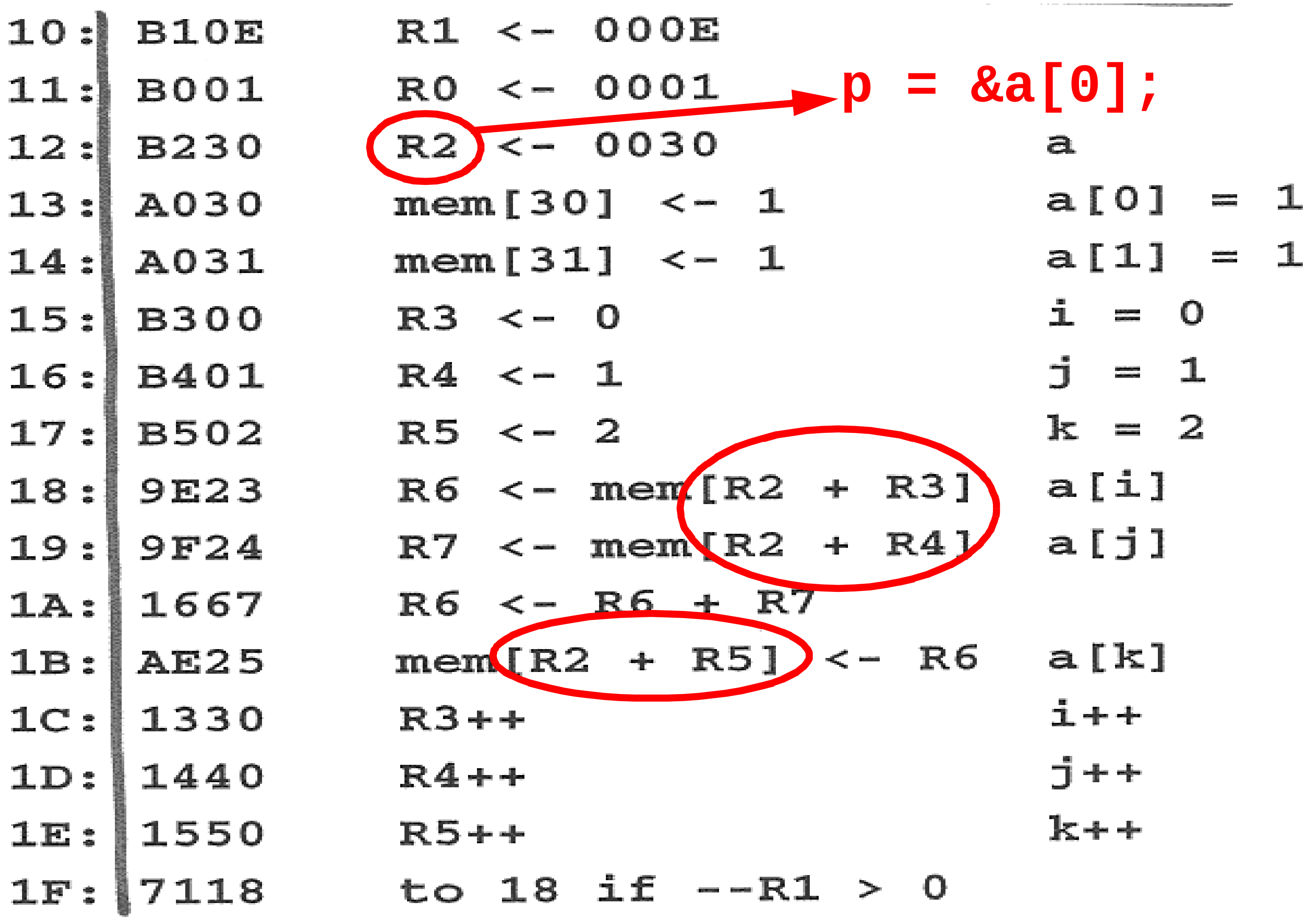
C Program for Fibonacci Array

```
#include <stdio.h>

main()
{
 int a[16];
 int n, i, j, k;

 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);

 for (i = 0; i < 16; i++) {
  printf("%d ", a[i]);
 }
 printf("\n");
}
```

- We will see how to implement the line in red using indexed addressing in TOY

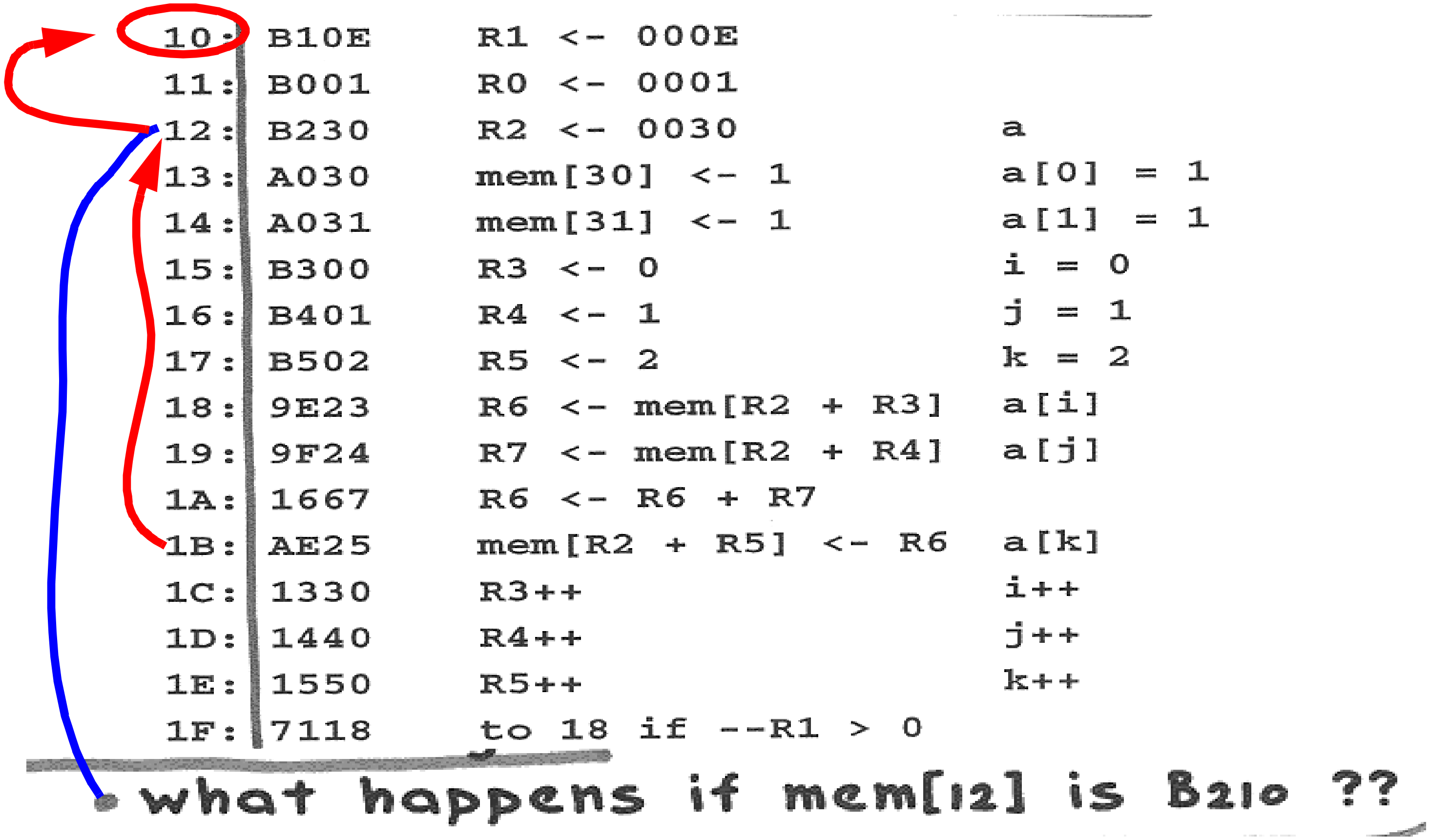
TOY Version of Fibonacci Program



```
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
```

p = &a[0];

Food for Thought



```
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
```

• what happens if mem[12] is B210 ??

- Self-modifying programs
- Special purpose computer **->** general purpose computer **->** stored program computer **->** self-modifying stored program computer
- Are some machines intrinsically more powerful than others?? Stay tuned.

Outline

- ~~Review and Introduction~~
- ~~Data representation~~
- ~~Dynamic addressing~~
- **Control flow**
- TOY simulator
- Conclusions

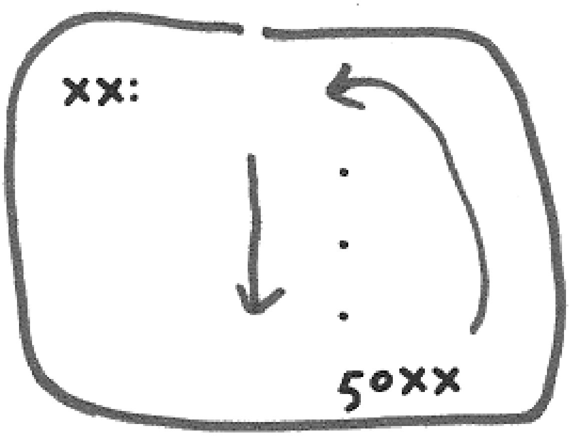
Branches and Looping

Press GO, computer either

* executes some instructions and halts
* gets caught in a loop

"infinite loop"

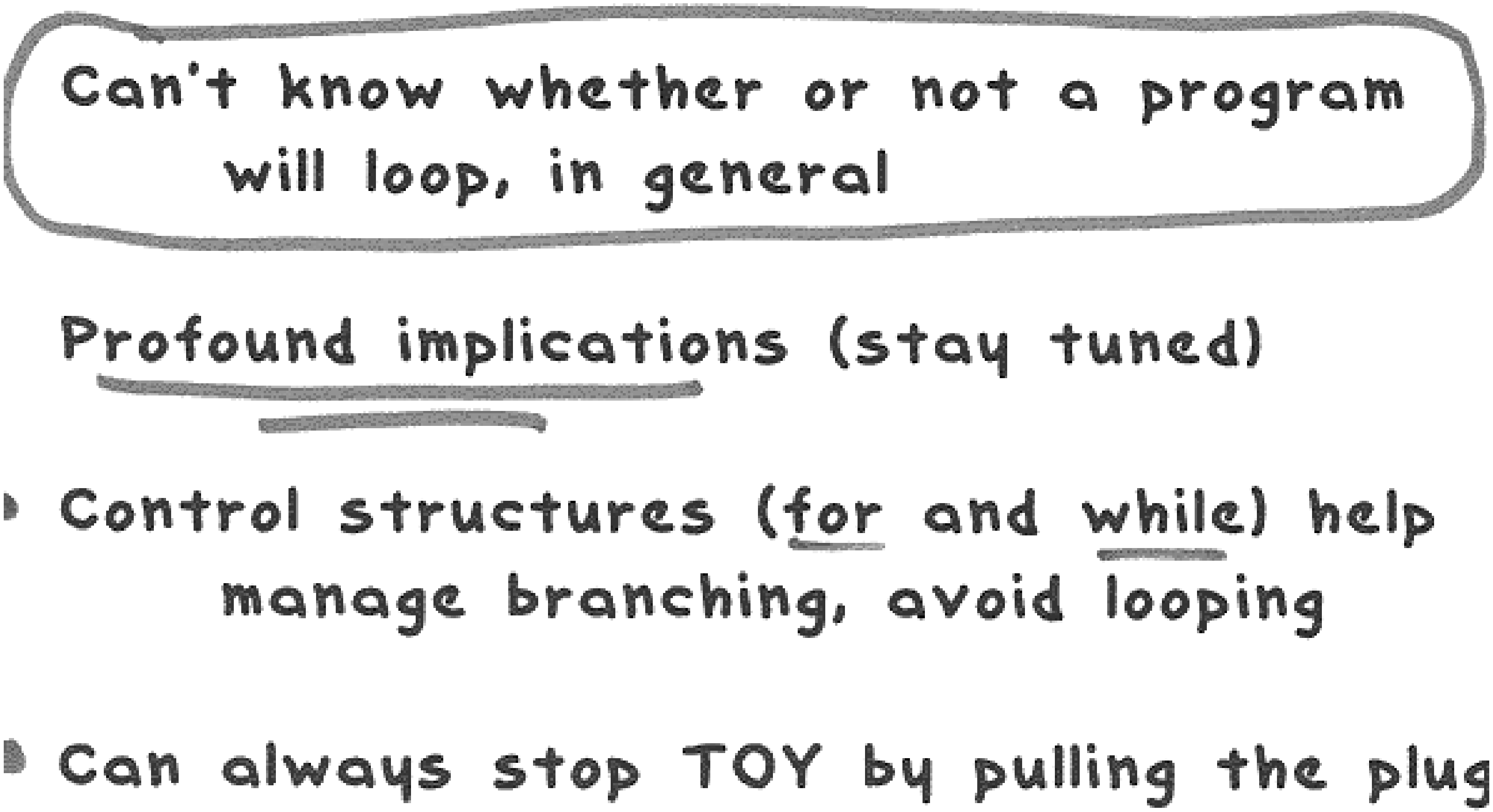
puzzles and/or panics programmers
(it will happen to you!)



- Often more complicated

The Halting Problem

- Why doesn't the compiler detect infinite loops and tell me?



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

Profound implications (stay tuned)

- Control structures (for and while) help manage branching, avoid looping
- Can always stop TOY by pulling the plug

Function Calls

- Functions can be written and used by different people

Issues:

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

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

To implement functions **(one possibility)**

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

Example Function

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

a^b

- 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
```

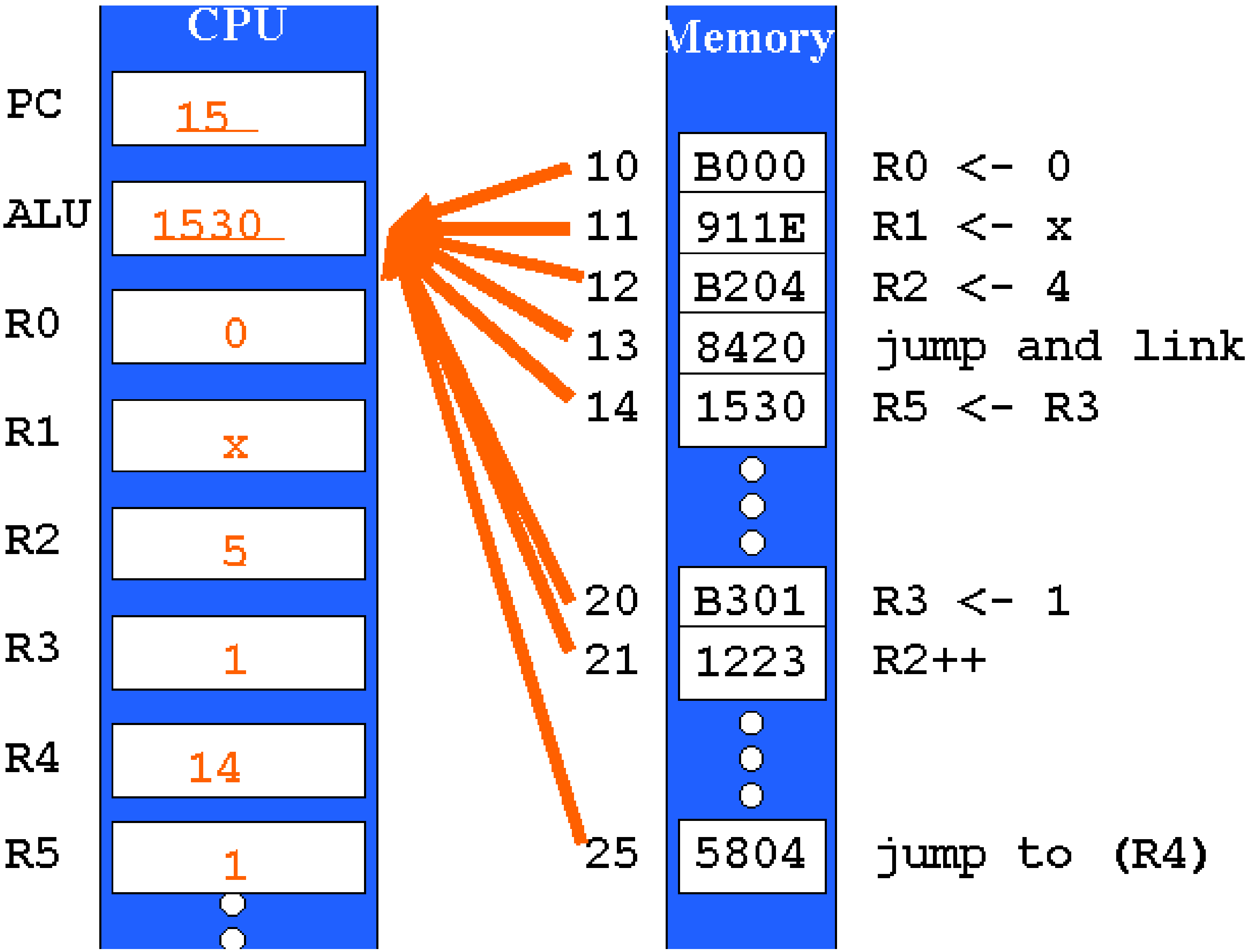
Takes care of b==0

Example Caller

<u>Ex:</u> 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
```

Function Call Demo



CPU

PC	15
ALU	1530
R0	0
R1	x
R2	5
R3	1
R4	14
R5	1

Memory

10	B000	R0 <- 0
11	911E	R1 <- x
12	B204	R2 <- 4
13	8420	jump and link
14	1530	R5 <- R3
20	B301	R3 <- 1
21	1223	R2++
25	5804	jump to (R4)

The Use of Registers vs. Memory for Function Calls

Precious resource: registers

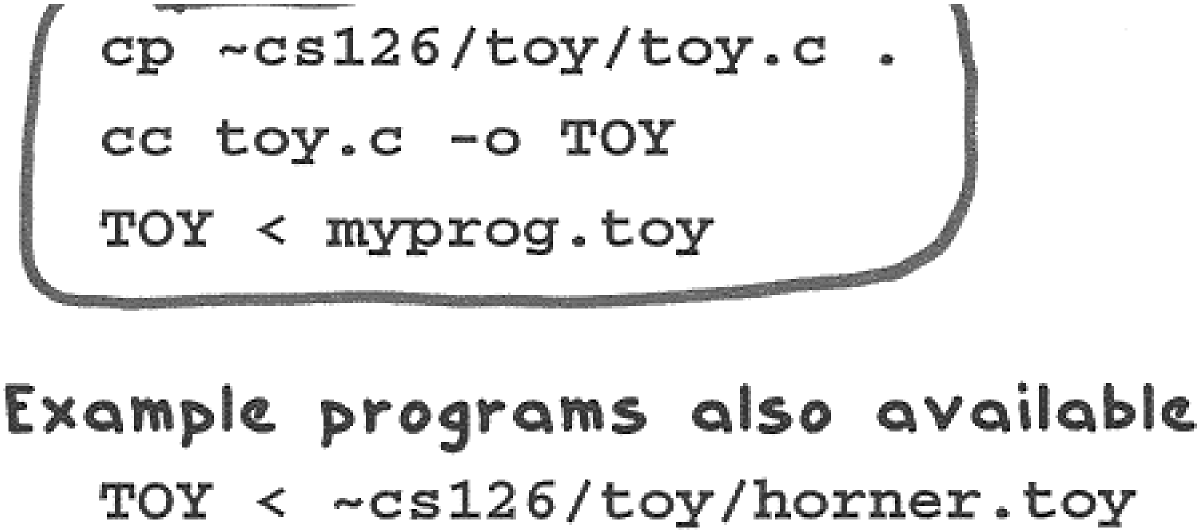
- Call a function from within a function?
  (use a stack) recursion

- Stack is implemented using main memory
- Review:
  - Call: push environment (registers and PC)
  - Call: push function parameters
  - Inside a function: look for parameters on the stack
  - Return: restores environment by popping stack
- Registers can still be used as optimizations

Outline

- ~~Review and Introduction~~
- ~~Data representation~~
- ~~Dynamic addressing~~
- ~~Control flow~~
- **TOY simulator**
- Conclusions

Availability



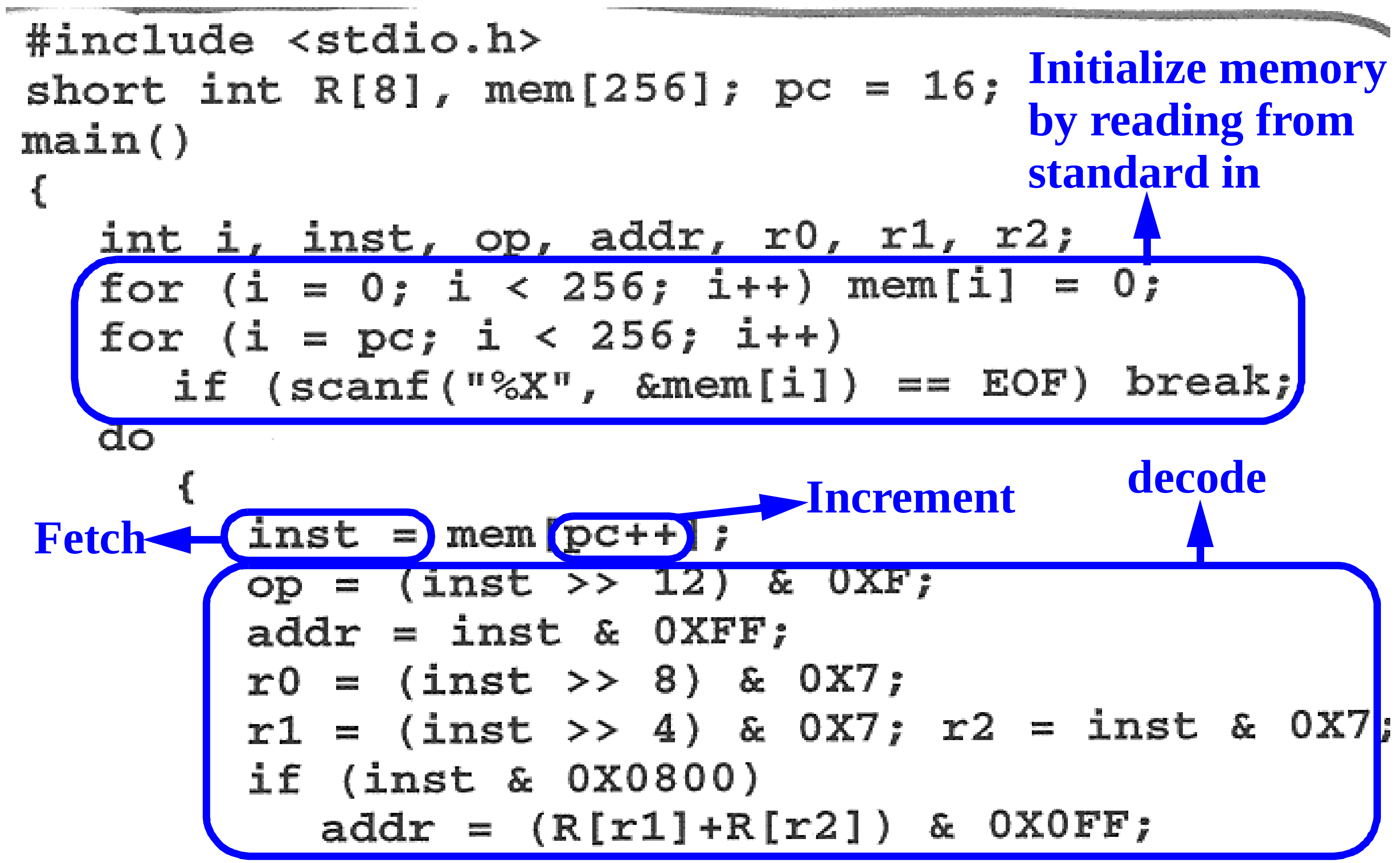
```
cp ~cs126/toy/toy.c .
cc toy.c -o TOY
TOY < myprog.toy
```

Example programs also available

```
TOY < ~cs126/toy/horner.toy
```

- Better yet, download java version from announcement page
- Edit “toy.html”, reopen it in browser

TOY Simulator (Part 1:fetch, incr, decode)



```
#include <stdio.h>
short int R[8], mem[256]; pc = 16;
main()
{
   int i, inst, op, addr, r0, r1, r2;
   for (i = 0; i < 256; i++) mem[i] = 0;
   for (i = pc; i < 256; i++)
      if (scanf("%X", &mem[i]) == EOF) break;
   do
      {
        inst = mem[pc++];
        op = (inst >> 12) & 0XF;
        addr = inst & 0XFF;
        r0 = (inst >> 8) & 0X7;
        r1 = (inst >> 4) & 0X7; r2 = inst & 0X7;
        if (inst & 0X0800)
           addr = (R[r1]+R[r2]) & 0X0FF;
```

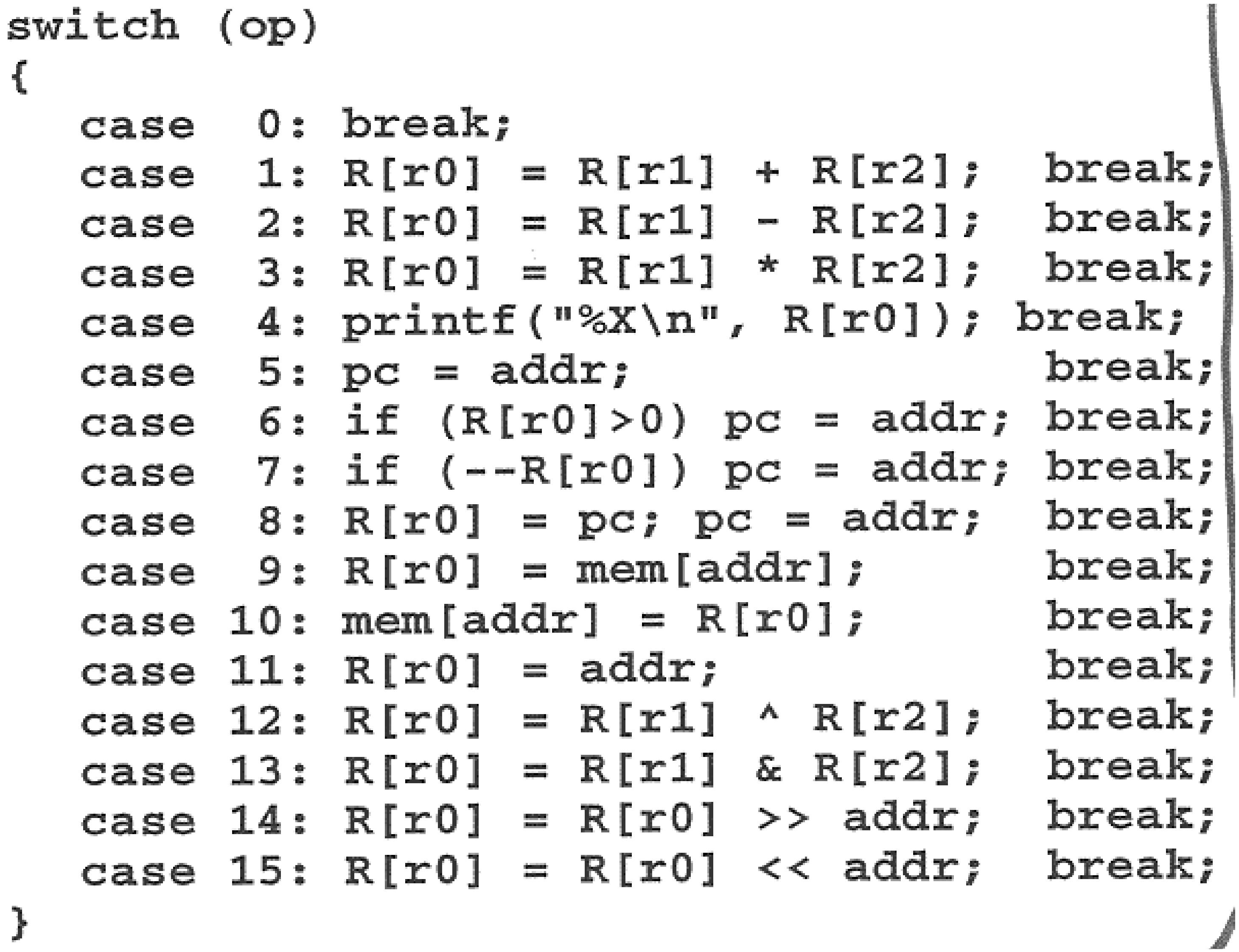
Initialize memory by reading from standard in

Fetch

Increment

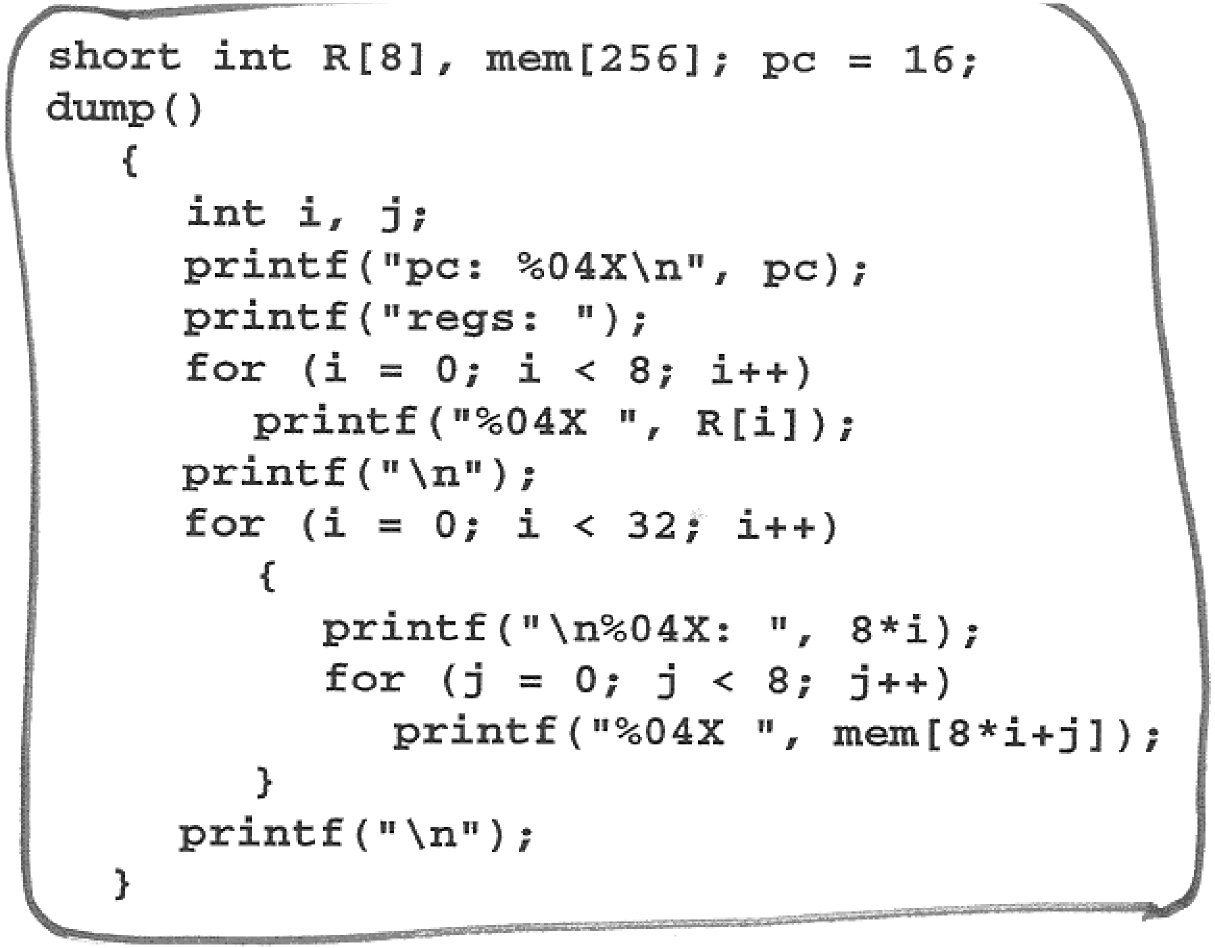
decode

TOY Simulator (Part 2: execute)



```
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("%X\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;
}
```

TOY Dump



```
short int R[8], mem[256]; pc = 16;
dump()
  {
     int i, j;
     printf("pc: %04X\n", pc);
     printf("regs: ");
     for (i = 0; i < 8; i++)
        printf("%04X ", R[i]);
     printf("\n");
     for (i = 0; i < 32; i++)
        {
           printf("\n%04X: ", 8*i);
           for (j = 0; j < 8; j++)
              printf("%04X ", mem[8*i+j]);
        }
     printf("\n");
  }
```

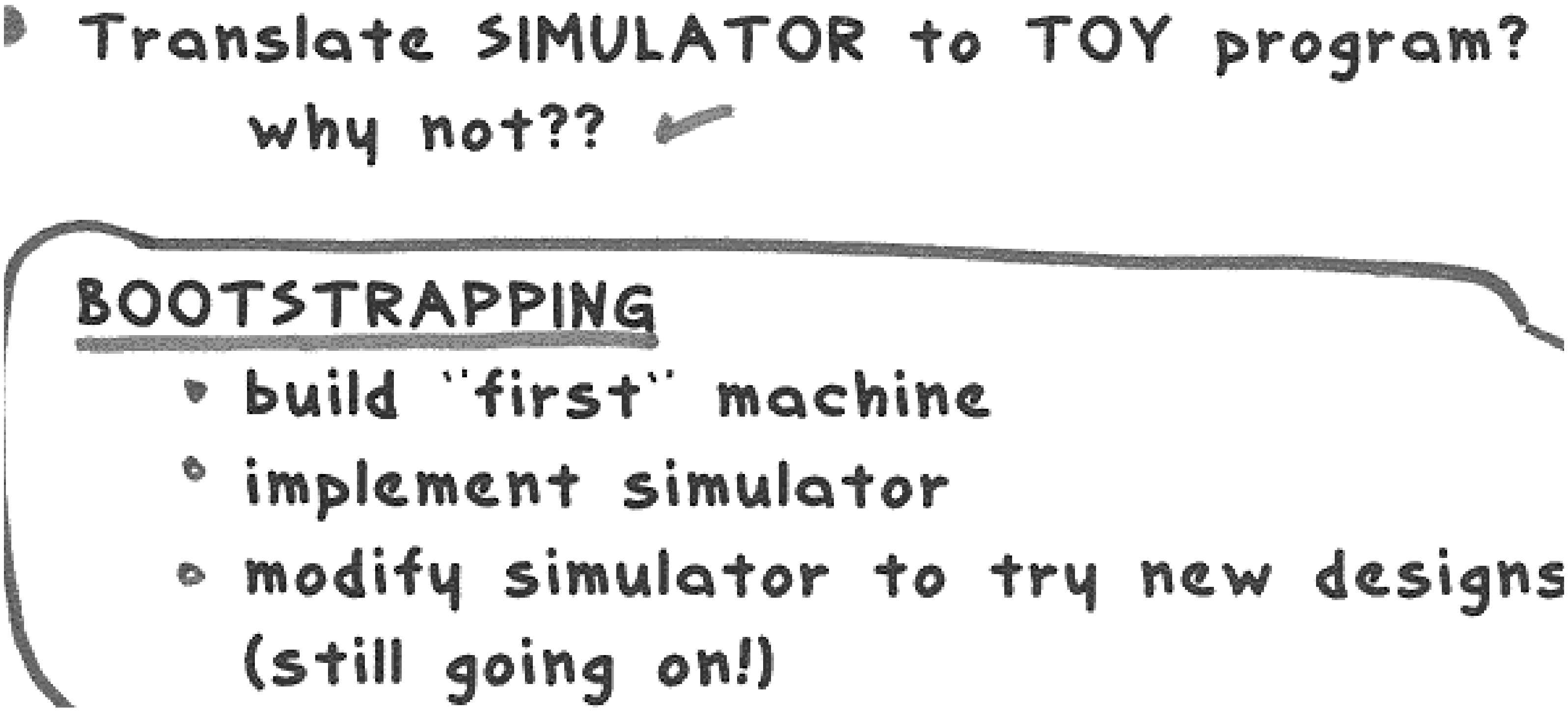
- Dump is in hex, 8 words/line, 4 digits/word

```
0000:| 0000 0004 0000 0000 0000 0000 0000 0000
0008:| 910A 110A 0002 0000 0000 0000 0000 0000
```

Outline

- ~~Review and Introduction~~
- ~~Data representation~~
- ~~Dynamic addressing~~
- ~~Control flow~~
- ~~TOY simulator~~
- **Conclusions**
  - Relationships among machine language programming, C programming, TOY machine, and “other” machines

Engineering and Theoretical Implications of Simulator



- Translate SIMULATOR to TOY program?

  why not??

BOOTSTRAPPING

- build "first" machine
- implement simulator
- modify simulator to try new designs (still going on!)

- Theoretically, any von Neumann machine can simulate any other von Neumann machine--all of them have the same "power"!! (More later)

What We Have Learned

- Two’s complement
  - How to represent negative numbers
  - How to perform addition and subtraction
  - Understand overflow
- How to use indexed addressing to access data structures
- Function calls
  - Passing parameters in registers
  - Save and restore PC