CSC 252: Computer Organization
Spring 2025: Lecture 11

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Today: Data Structures and Buffer Overflow

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
- Structures
  - Allocation
  - Access
  - Alignment
- Buffer Overflow

String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read
- Similar problems with other library functions
  - **`strcpy`**, **`strcat`**: Copy strings of arbitrary length
  - **`scanf`**, **`fscanf`**, **`sscanf`**, when given **`%s`** conversion specification

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

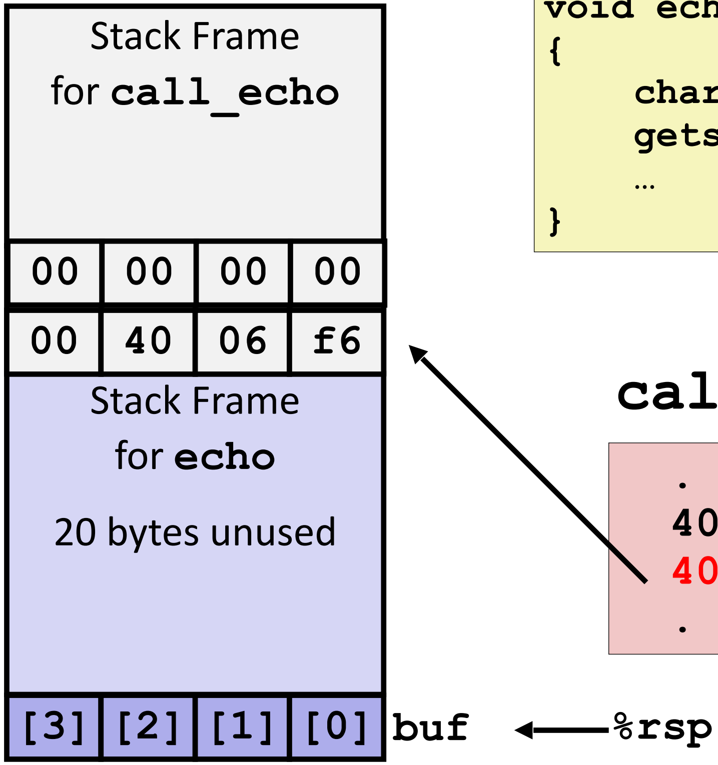
```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

Buffer Overflow Stack Example

Before call to gets



Stack Frame for call_echo

00	00	00	00
00	40	06	f6

Stack Frame for echo

20 bytes unused

[3]	[2]	[1]	[0]	buf

%rsp

```
void echo()
{
     char buf[4];
     gets(buf);
     …
}
```

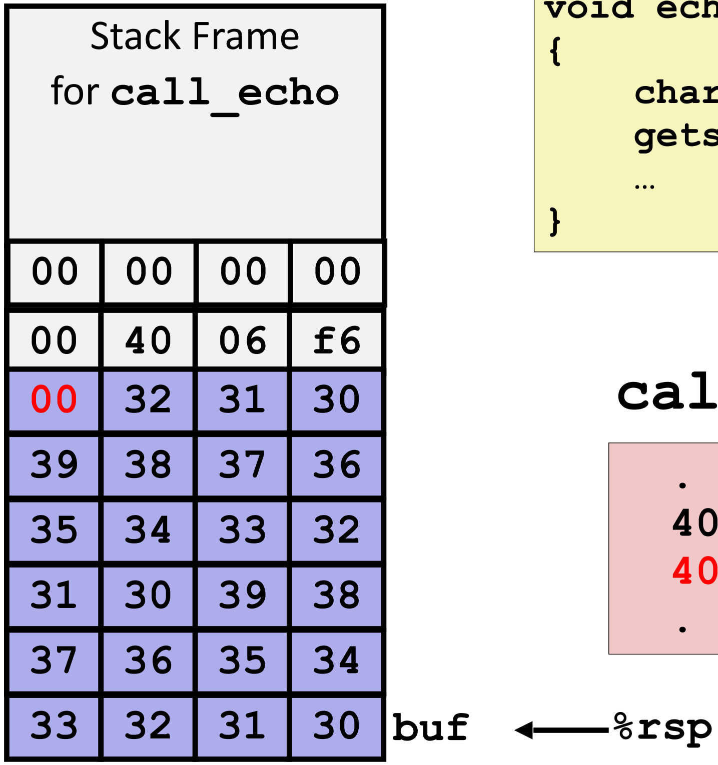
```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   …
```

call_echo:

```
  . . .
  4006f1:    callq  4006cf <echo>
  4006f6:    add    $0x8,%rsp
  . . .
```

Buffer Overflow Stack Example #1

After call to gets



Stack Frame
for `call_echo`

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
    char buf[4];
    gets(buf);
    …
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  …
```

call_echo:

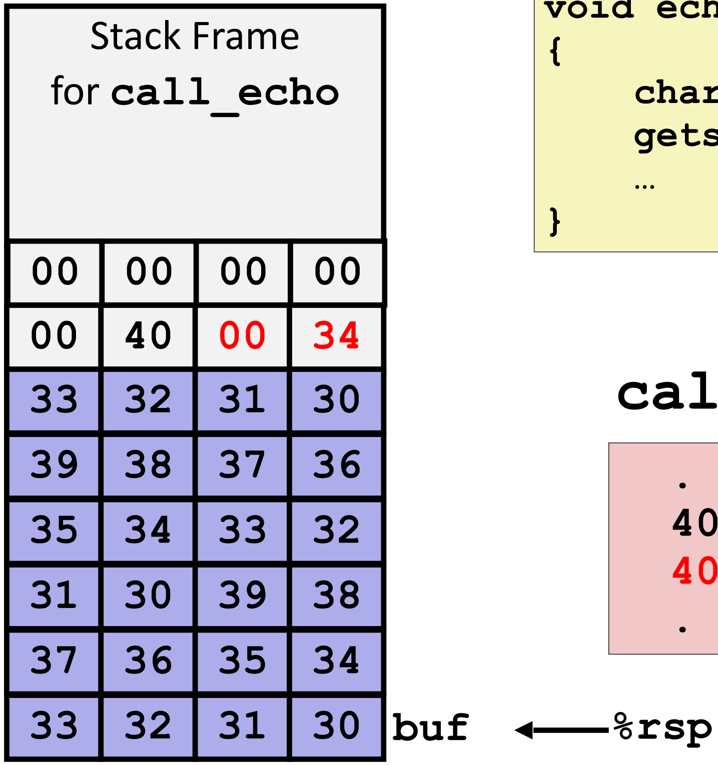
```
  . . .
  4006f1:   callq  4006cf <echo>
  4006f6:   add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets



Stack Frame for `call_echo`

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
     char buf[4];
     gets(buf);
     …
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   …
```

call_echo:

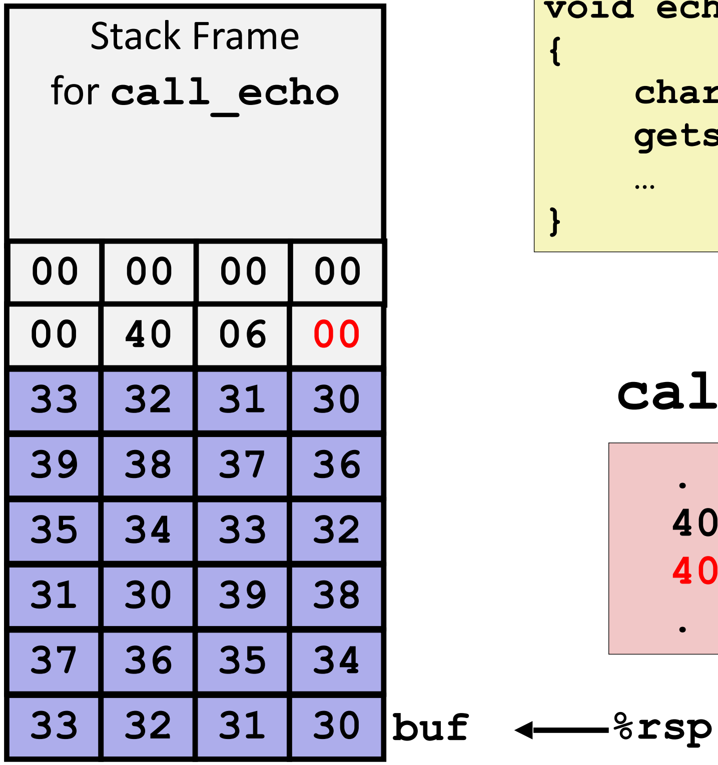
```
   . . .
   4006f1:    callq   4006cf <echo>
   4006f6:    add     $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

Overflowed buffer, and corrupt return address

Buffer Overflow Stack Example #3

After call to gets



Stack Frame
for call_echo

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

```
void echo()
{
    char buf[4];
    gets(buf);
    …
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  …
```

call_echo:

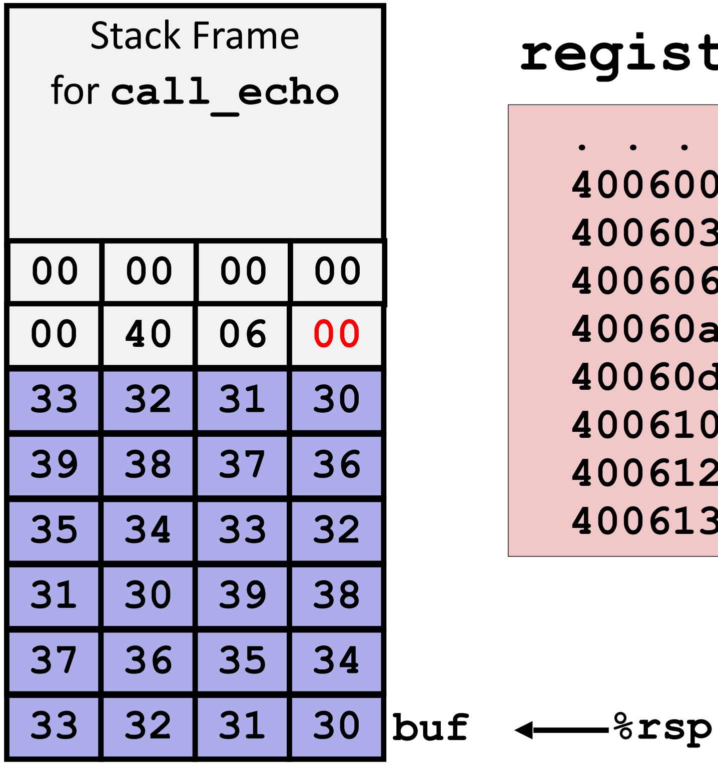
```
  . . .
  4006f1:    callq   4006cf <echo>
  4006f6:    add     $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

Overflowed buffer, corrupt return address, but program appears to still work!

Buffer Overflow Stack Example #4

After call to gets



Stack Frame
for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`register_tm_clones:`

```
 . . .
400600:    mov     %rsp,%rbp
400603:    mov     %rax,%rdx
400606:    shr     $0x3f,%rdx
40060a:    add     %rdx,%rax
40060d:    sar     %rax
400610:    jne     400614
400612:    pop     %rbp
400613:    retq
```

“Returns” to unrelated code
Could be code controlled by attackers!

What to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

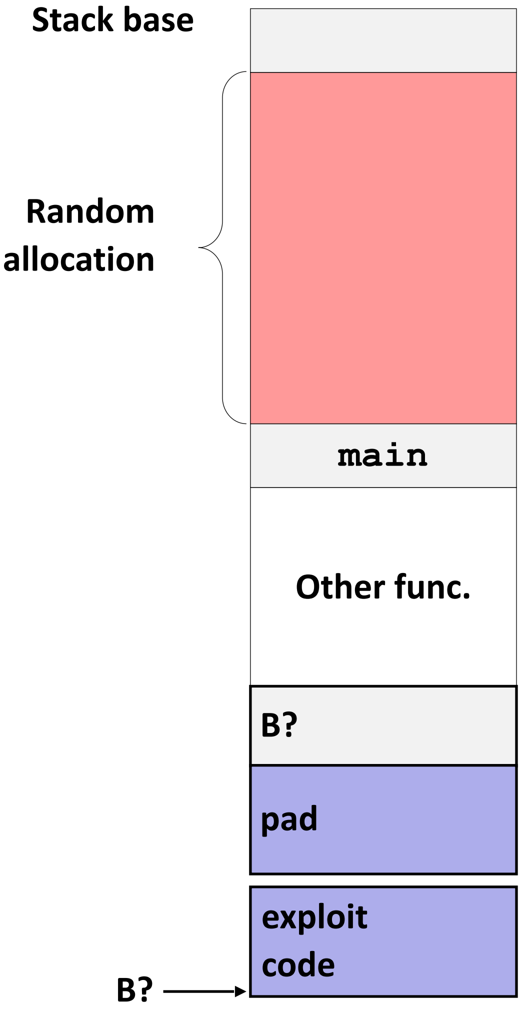
1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- For example, use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don’t use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where `n` is a suitable integer

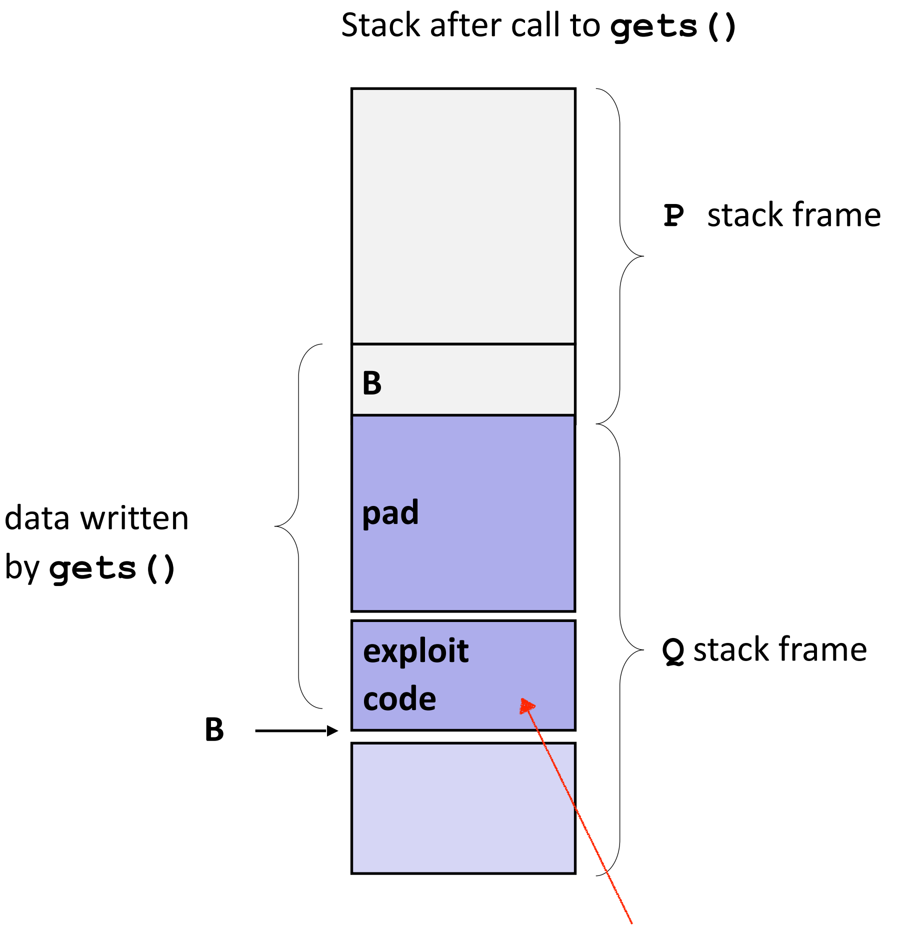
2. System-Level Protections can help

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code



2. System-Level Protections can help

- Nonexecutable code segments
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
  - X86-64 added explicit “execute” permission
  - Stack marked as non-executable



Any attempt to execute this code will fail

3. Stack Canaries can help

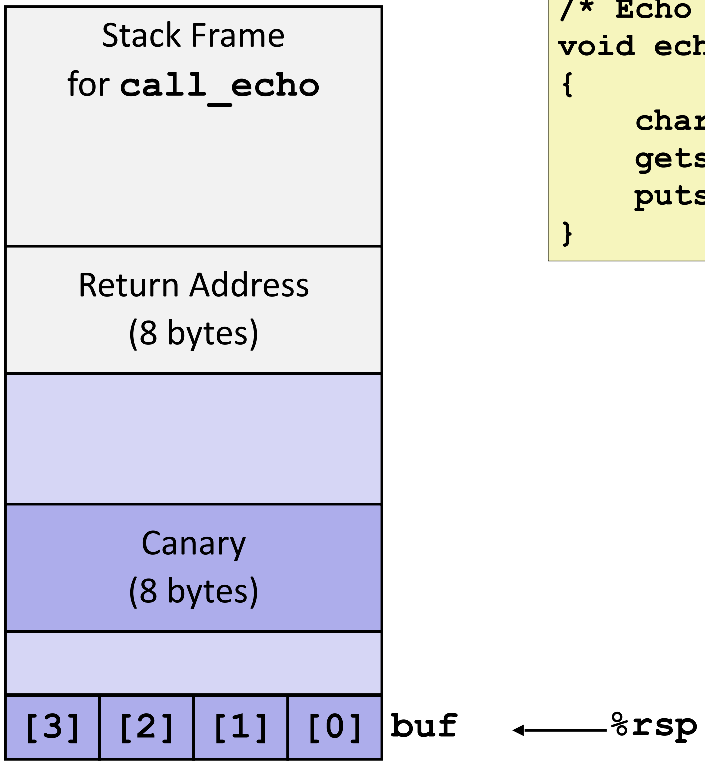
- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - `-fstack-protector`
  - Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

Setting Up Canary

Before call to gets

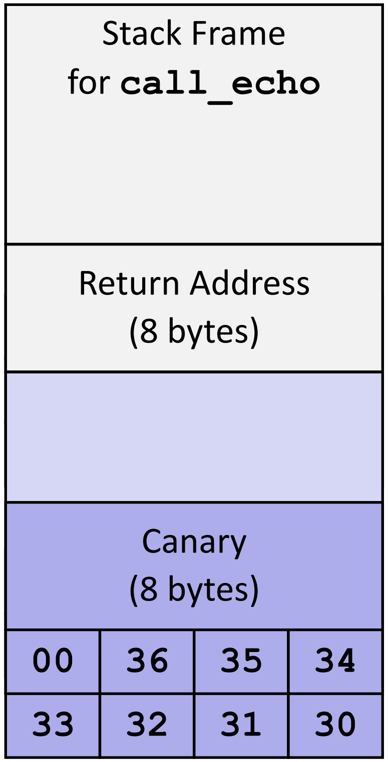


```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq      %fs:40, %rax   # Get canary
    movq      %rax, 8(%rsp) # Place on stack
    xorl      %rax, %rax    # Erase canary
    . . .
```

Checking Canary

After call to gets



Stack Frame
for `call_echo`

Return Address
(8 bytes)

Canary
(8 bytes)

00	36	35	34
33	32	31	30

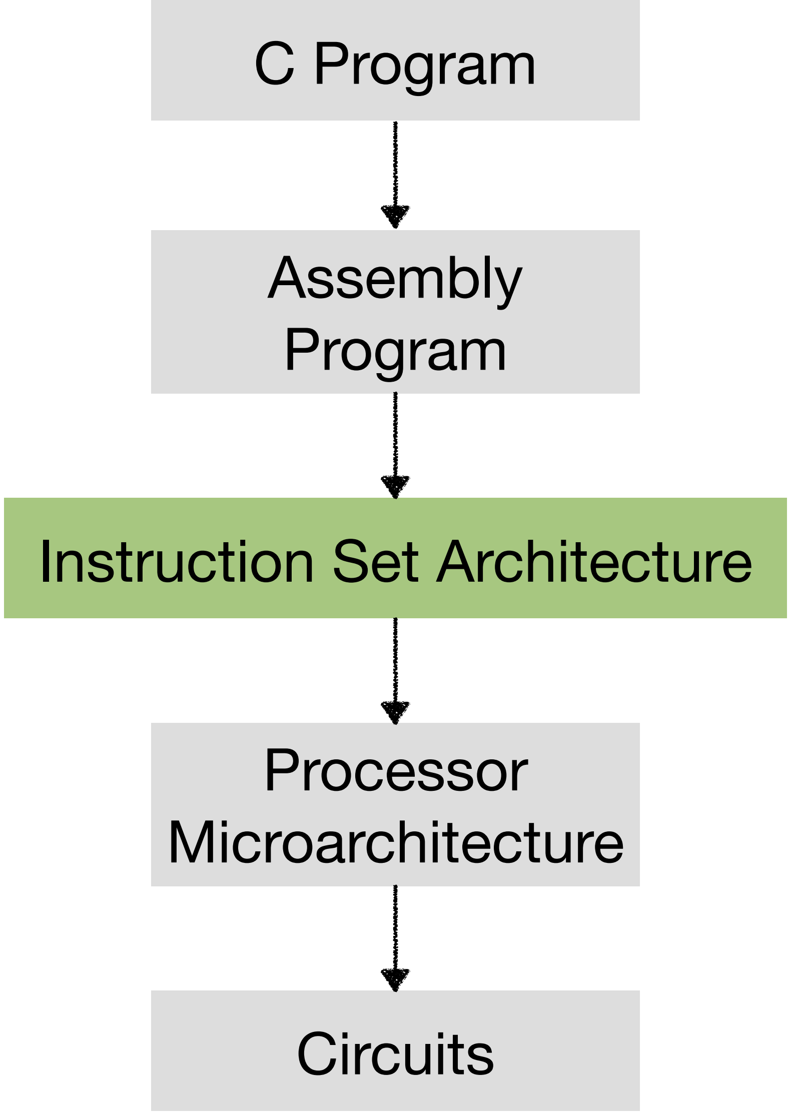
buf ← %rsp

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: ***0123456***

```
echo:
     . . .
     movq      8(%rsp), %rax      # Retrieve from stack
     xorq      %fs:40, %rax       # Compare to canary
     je        .L6                # If same, OK
     call      __stack_chk_fail   # FAIL
.L6: . . .
```

So far in 252…



C Program

Assembly Program

Instruction Set Architecture

Processor Microarchitecture

Circuits

```
int, float
if, else
+, -, >>
```

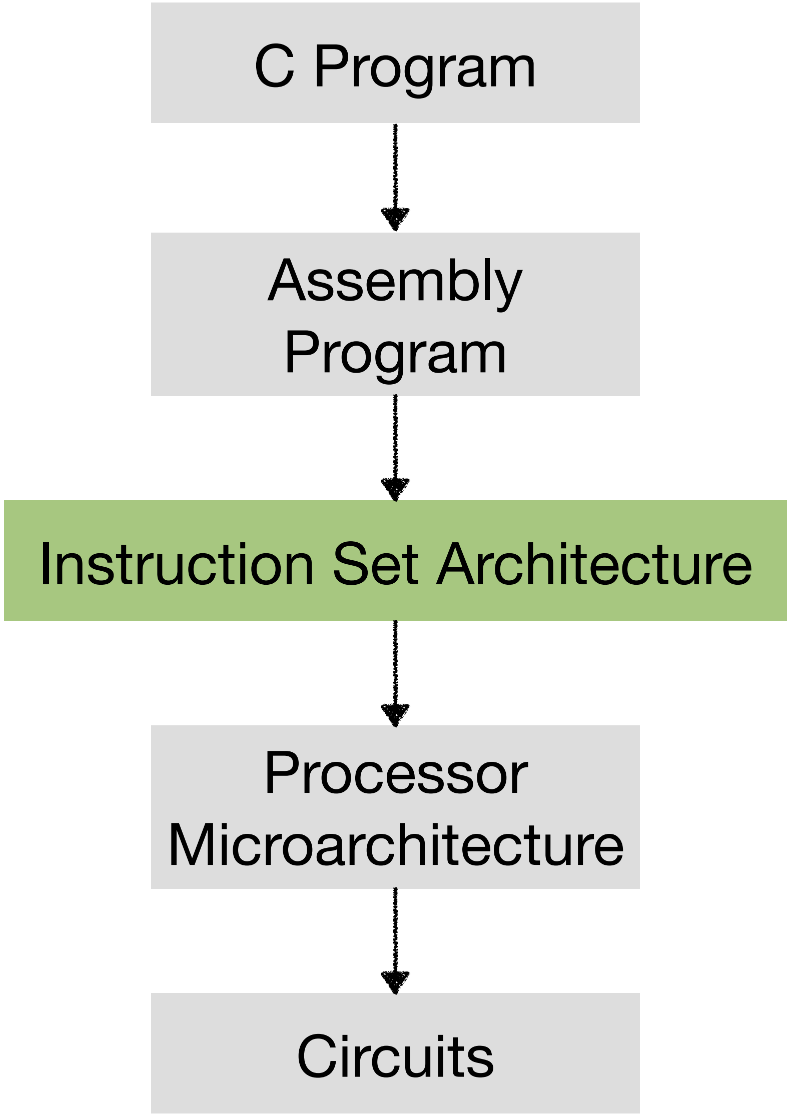
```
movq    %rsi, %rax
imulq   %rdx, %rax
jmp     .done
```

```
ret, call
movq, addq
jmp, jne
```

Logic gates

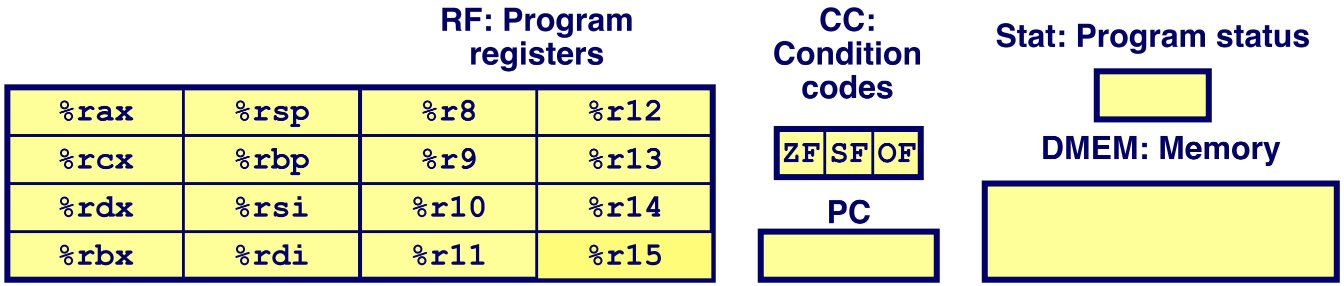
Transistors

So far in 252…



- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.
- Microarchitecture view:
  - What hardware needs to be built to run assembly programs?
  - How to run programs as fast (energy-efficient) as possible?

(Simplified) x86 Processor State



RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	%r15

CC: Condition codes

ZF	SF	OF

PC

Stat: Program status

DMEM: Memory

- Processor state is what's visible to assembly programs. Also known as architecture state.
- **Program Registers:** 16 registers.
- **Condition Codes:** Single-bit flags set by arithmetic or logical instructions (ZF, SF, OF)
- **Program Counter:** Indicates address of next instruction
- **Program Status:** Indicates either normal operation or error condition
- Memory
    - Byte-addressable storage array
    - Words stored in little-endian byte order

Why Have Instructions?

- Why do we need an ISA? Can we directly program the hardware?
- Simplifies interface
  - Software knows what is available
  - Hardware knows what needs to be implemented
- Abstraction protects software and hardware
  - Software can run on new machines
  - Hardware can run old software
- Alternatives: Application-Specific Integrated Circuits (ASIC)
  - No instructions, (largely) not programmable, fixed-functioned, so no instruction fetch, decoding, etc.
  - So could be implemented extremely efficiently.
  - Examples: video/audio codec, (conventional) image signal processors, (conventional) IP packet router

Today: Instruction Encoding

- How to translate assembly instructions to binary
  - Essentially how an assembler works
- Using the Y86-64 ISA: Simplified version of x86-64

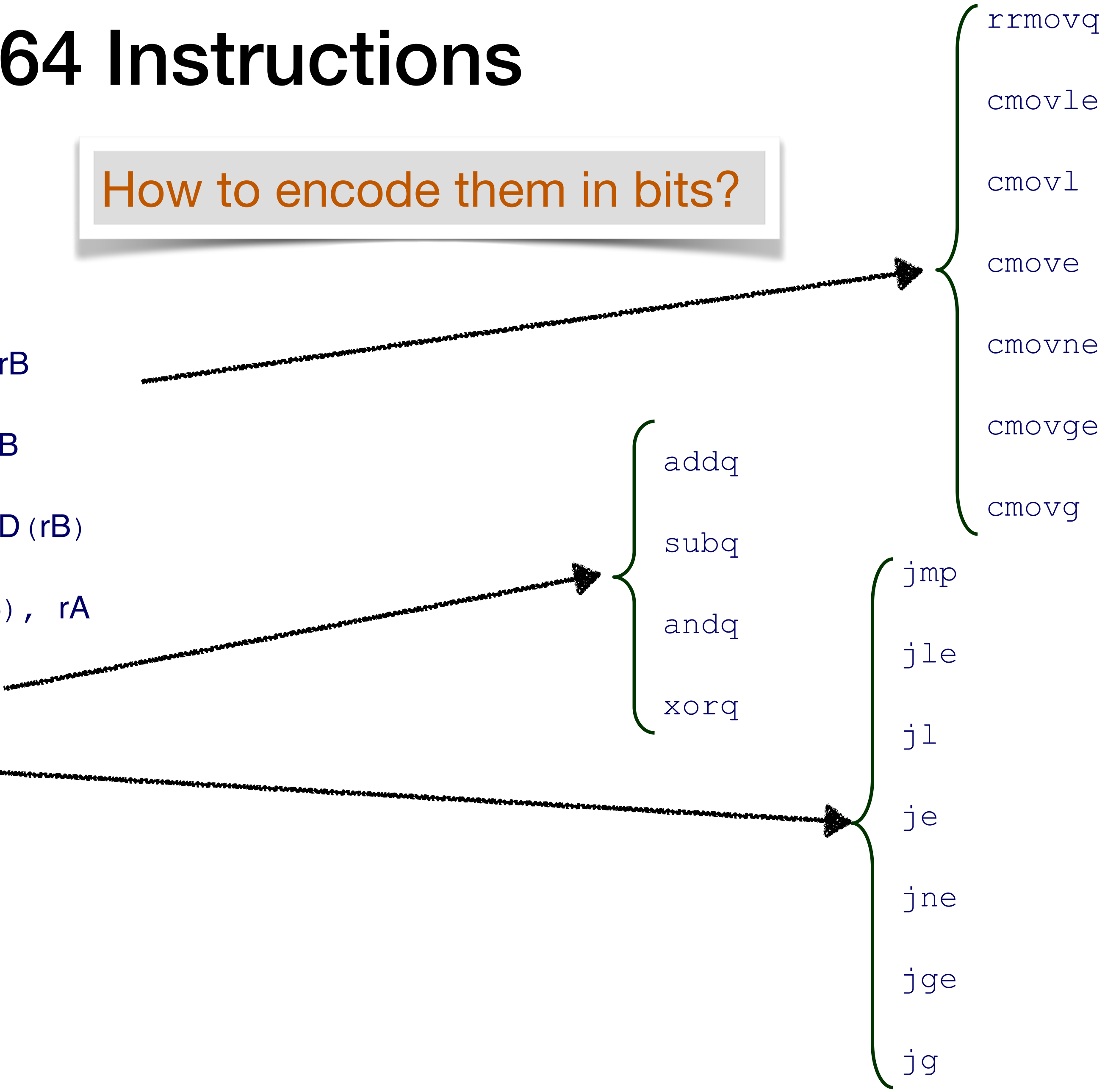
How are Instructions Encoded in Binary?

- Remember that instructions are stored in memory **as bits** (just like data)
- Each instruction is fetched (according to the address specified in the PC), decoded, and executed by the CPU
- The ISA defines the format of an instruction (syntax) and its meaning (semantics)
- Idea: encode the two major fields, opcode and operand, separately in bits.
    - The OPCODE field says what the instruction does (e.g. ADD)
    - The OPERAND field(s) say where to find inputs and outputs

Y86-64 Instructions

How to encode them in bits?

```
halt
nop
cmovXX rA, rB
irmovq V, rB
rmmovq rA, D(rB)
mrmovq D(rB), rA
OPq rA, rB
jXX Dest
call Dest
ret
pushq rA
popq rA
```

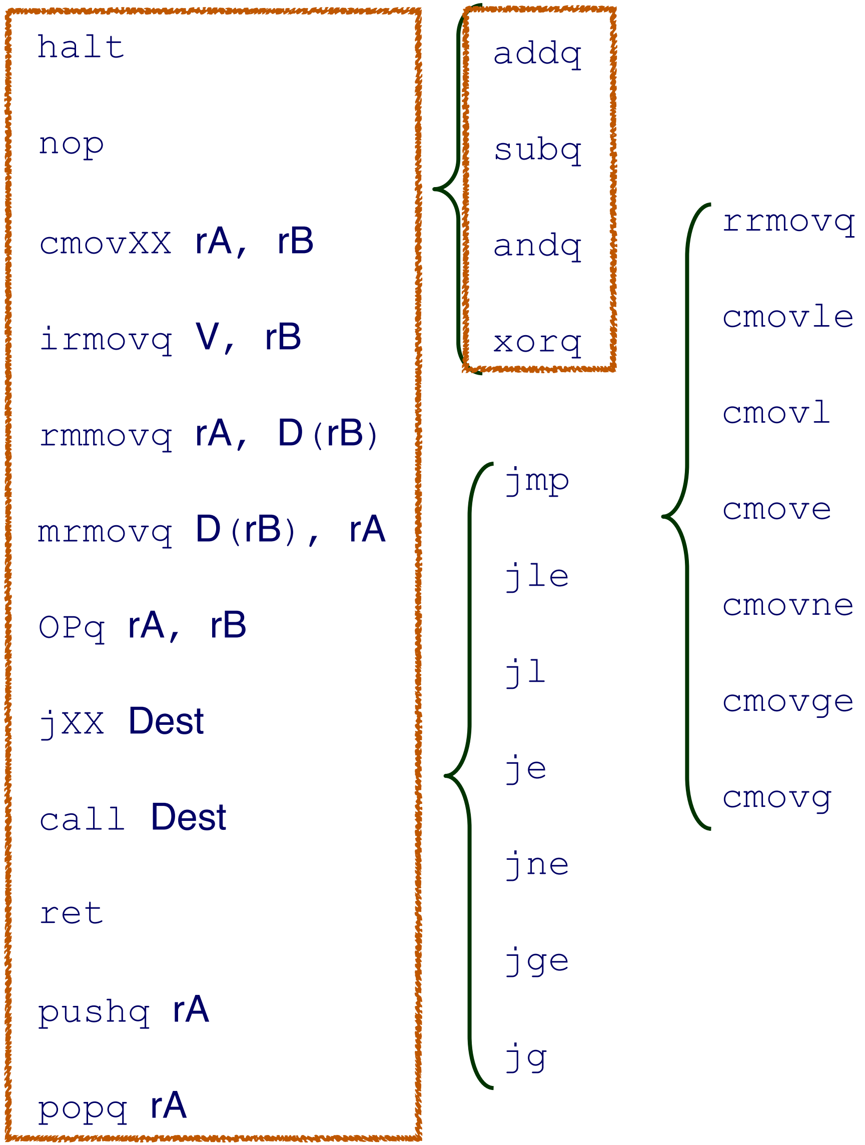


```
rrmovq
cmovle
cmovl
cmove
cmovne
cmovge
cmovg
```

```
addq
subq
andq
xorq
```

```
jmp
jle
jl
je
jne
jge
jg
```

Encoding Operands



```
halt
nop
cmovXX rA, rB
irmovq V, rB
rmmovq rA, D(rB)
mrmovq D(rB), rA
OPq rA, rB
jXX Dest
call Dest
ret
pushq rA
popq rA
```

```
addq
subq
andq
xorq
```

```
jmp
jle
jl
je
jne
jge
jg
```

```
rrmovq
cmovle
cmovl
cmove
cmovne
cmovge
cmovg
```

- 27 Instructions, so need 5 bits for encoding the opcode.
- Or: group similar instructions, use one opcode for them, and then use more bits to indicate specific instructions within a group.
- E.g., 12 categories, so 4 bits
- There are four instructions within the `OPq` category, so additional 2 bits. Similarly, 3 more bits for `jXX` and `cmovXX`, respectively.
- Which one is better???

Encoding Operands

Byte	0	1	2	3	4	5	6	7	8	9
halt	0 0									
nop	1 0									
cmovXX rA, rB	2 fn									
irmovq V, rB	3 0									
rmmovq rA, D(rB)	4 0									
mrmovq D(rB), rA	5 0									
OPq rA, rB	6 fn									
jXX Dest	7 fn									
call Dest	8 0									
ret	9 0									
pushq rA	A 0									
popq rA	B 0									

- Design decision chosen by the textbook authors (don’t have to be this way!)
  - Use 4 bits to encode the instruction category
  - Another 4 bits to encode the specific instructions within a category
  - So 1 bytes for encoding opcode.
  - Is this better than the alternative of using 5 bits without classifying instructions?
  - Trade-offs.

Encoding Registers

Each register has 4-bit ID

- Same encoding as in x86-64
- Register ID 15 (`0xF`) indicates "no register"

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

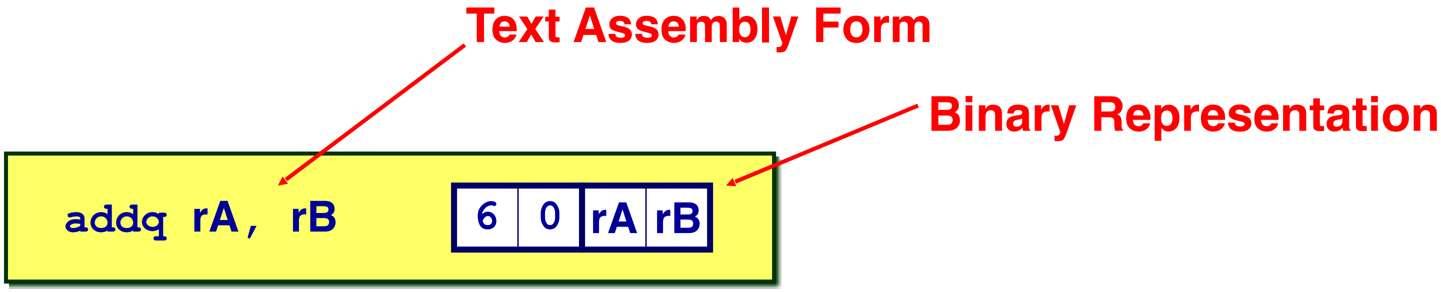
%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

Encoding Registers

Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`cmovXX` rA, rB	2 fn	rA rB								
`irmovq` V, rB	3 0	F rB								
`rmmovq` rA, D(rB)	4 0	rA rB								
`mrmovq` D(rB), rA	5 0	rA rB								
`OPq` rA, rB	6 fn	rA rB								
`jXX` Dest	7 fn									
`call` Dest	8 0									
`ret`	9 0									
`pushq` rA	A 0	rA F								
`popq` rA	B 0	rA F								

Instruction Example

Addition Instruction



Text Assembly Form

Binary Representation

`addq rA, rB`

6	0	rA	rB

- Add value in register rA to that in register rB
  - Store result in register rB
- Set condition codes based on result
- e.g., `addq %rax,%rsi` Encoding: `60 06`
- Two-byte encoding
  - First indicates instruction type
  - Second gives source and destination registers

Arithmetic and Logical Operations

Add

```
addq rA, rB
```

6	0	rA	rB

Subtract (rA from rB)

```
subq rA, rB
```

6	1	rA	rB

And

```
andq rA, rB
```

6	2	rA	rB

Exclusive-Or

```
xorq rA, rB
```

6	3	rA	rB

- Referred to generically as “`OPq`”
- Encodings differ only by “function code”
    - Low-order 4 bytes in first instruction word
- Set condition codes as side effect