CSC 252: Computer Organization Spring 2021: Lecture 10

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 2 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2021/labs/assignment2.html
  - Due on **March 5**, 11:59 PM (extended two days)
  - You (may still) have 3 slip days

14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	Mar 1	2	3	4 **Today**	5 **Due**	6

Announcement

- Will release programming assignment 3 today.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.
- Problem set on arithmetics: https://www.cs.rochester.edu/courses/252/spring2021/handouts.html.
  - Not to be turned in.

Vulnerable Buffer Code

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

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

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:012345678901234567890123
012345678901234567890123
```

Vulnerable Buffer Code

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

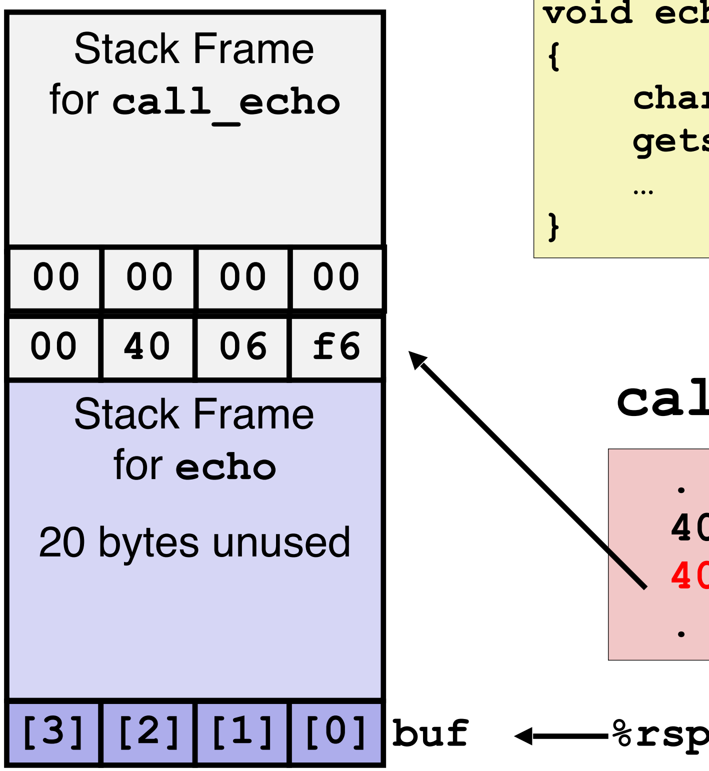
```
void call_echo() {
    echo();
}
```

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

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

Buffer Overflow Stack Example

Before call to gets




```
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	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
  . . .
```

```
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

`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
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123 4
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	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
  . . .
```

```
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	buf ←%rsp

`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”

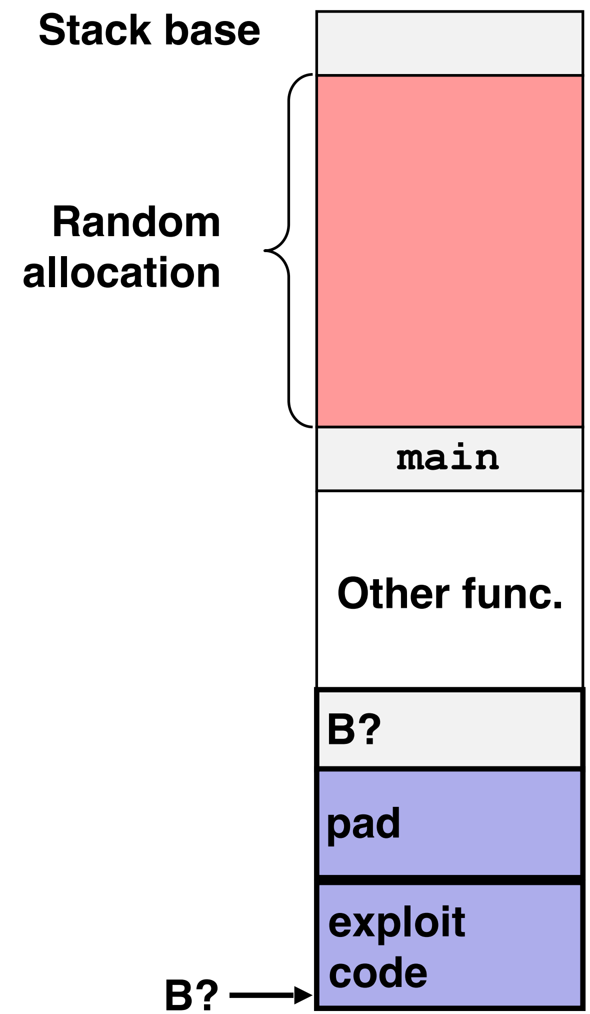
1. Avoid Overflow Vulnerabilities in Code (!)

```
/* 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

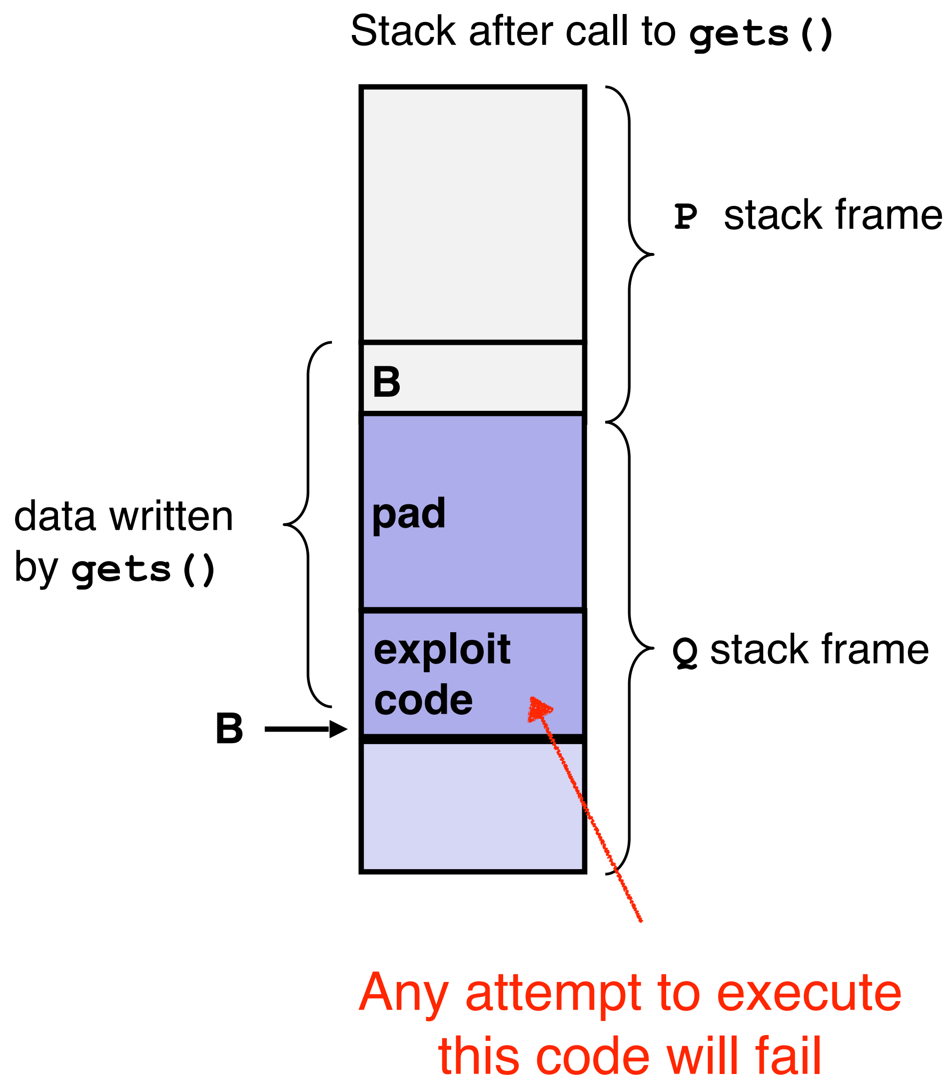
2. System-Level Protections can help

- 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



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)

3. Stack Canaries can help

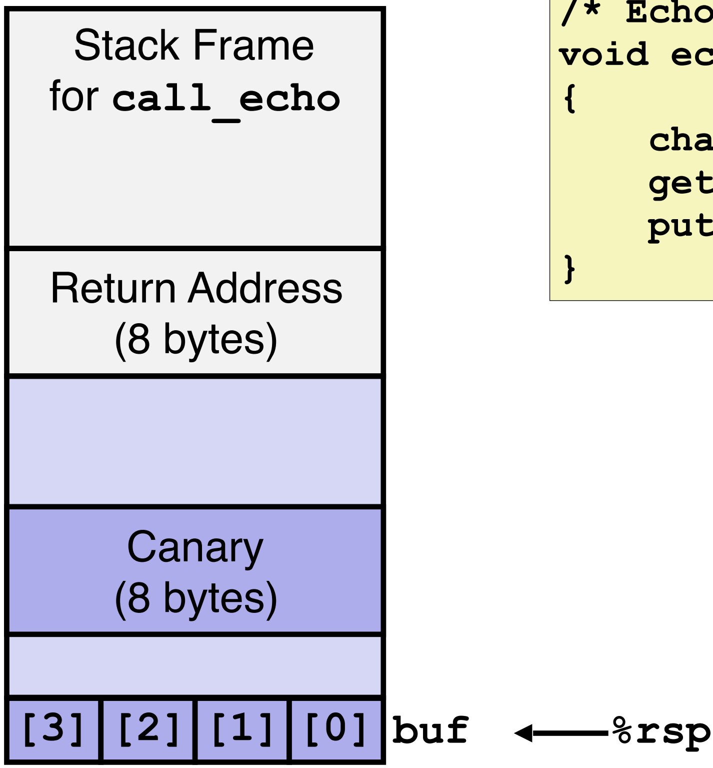
- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - `-fstack-protector`
  - Now the default (disabled earlier)

```
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

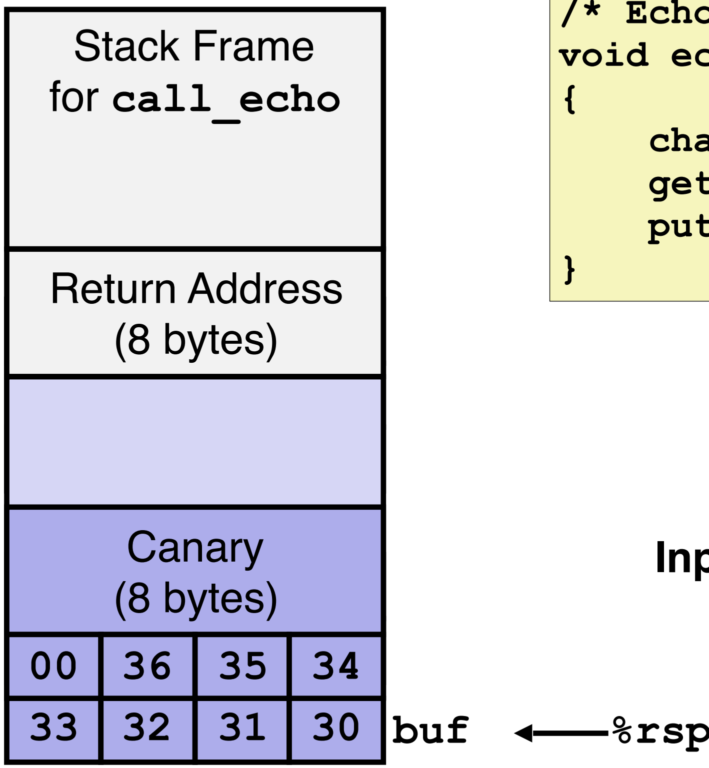



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
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

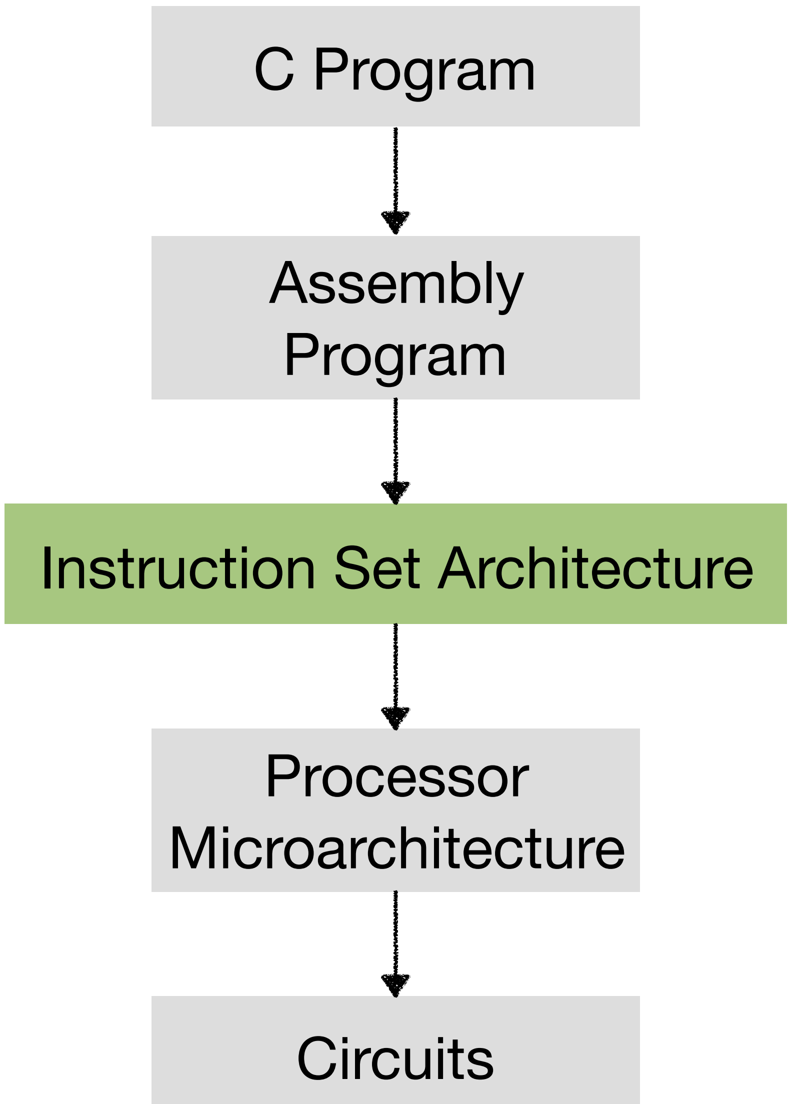



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

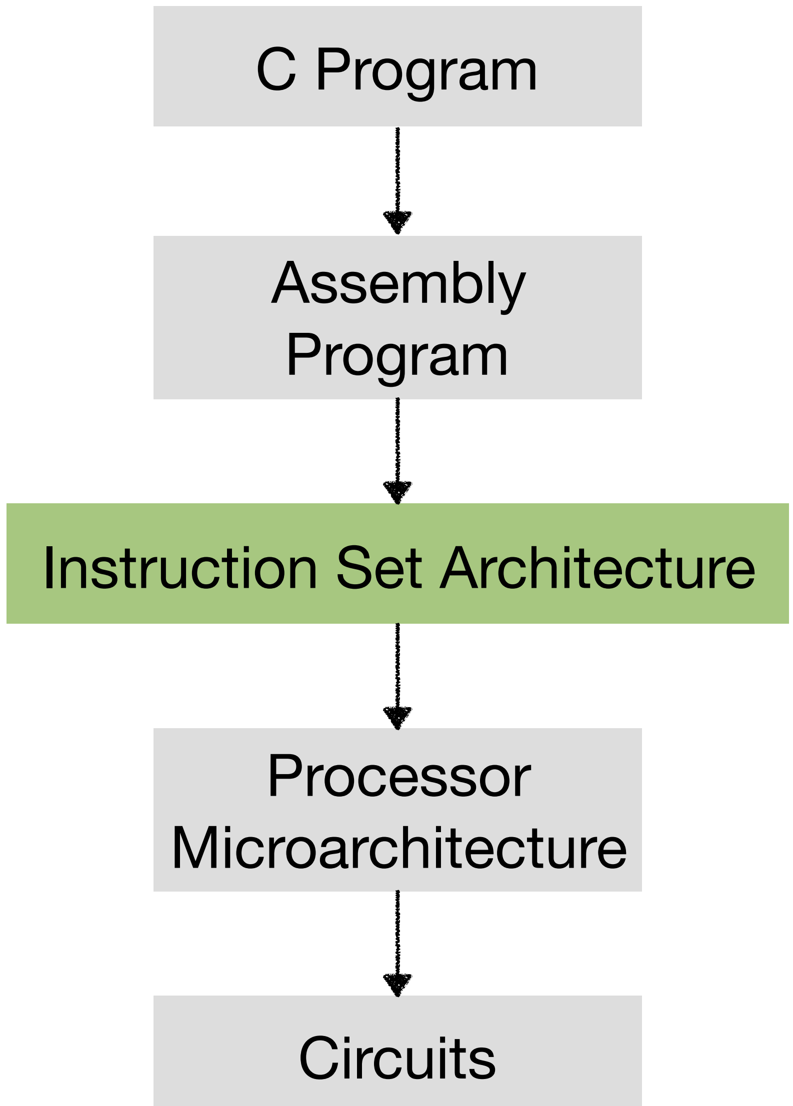
Input: ***0123456***

```
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…

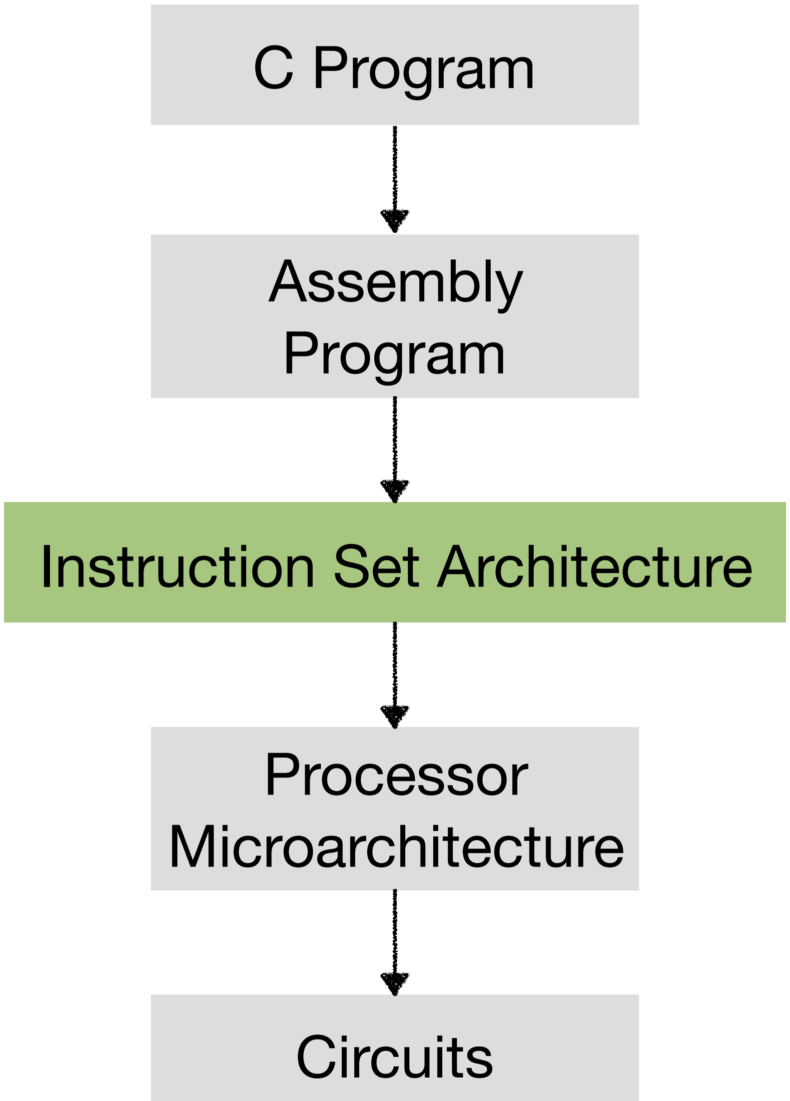

C Program
Assembly Program
Instruction Set Architecture
Processor Microarchitecture
Circuits

So far in 252…




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

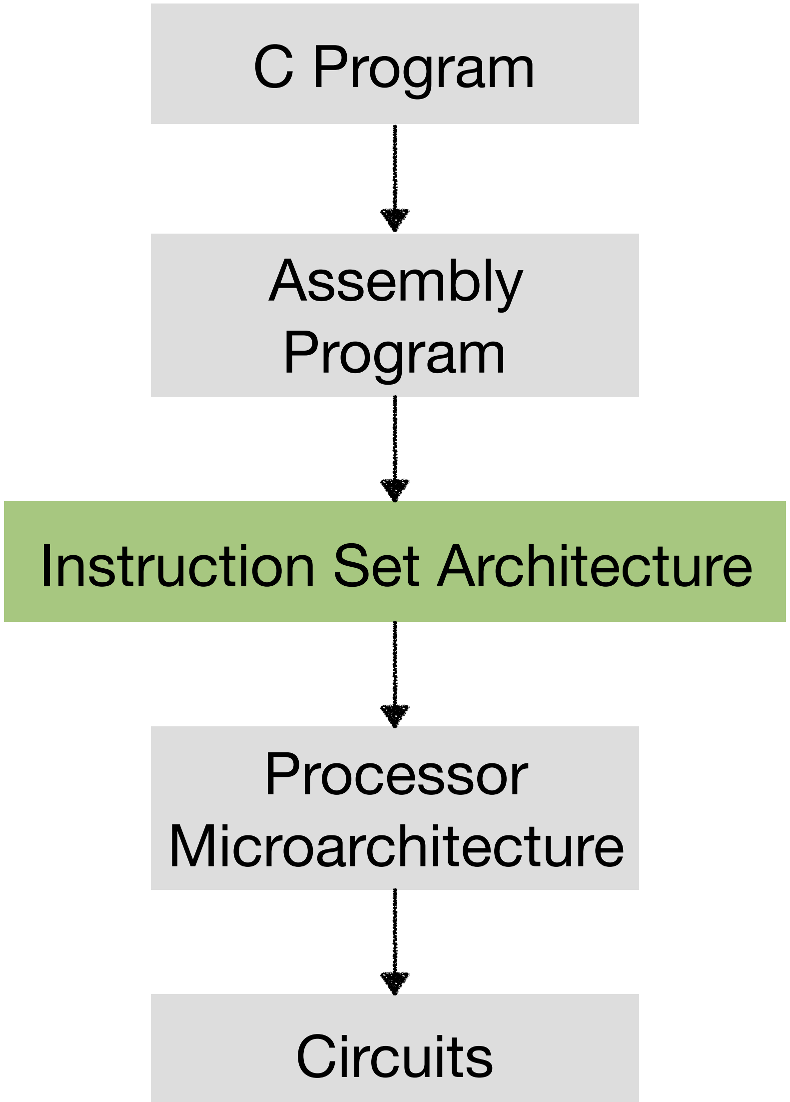
So far in 252…




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

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

So far in 252…




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

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

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

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

So far in 252…

C Program

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

Assembly Program

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

Instruction Set Architecture

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

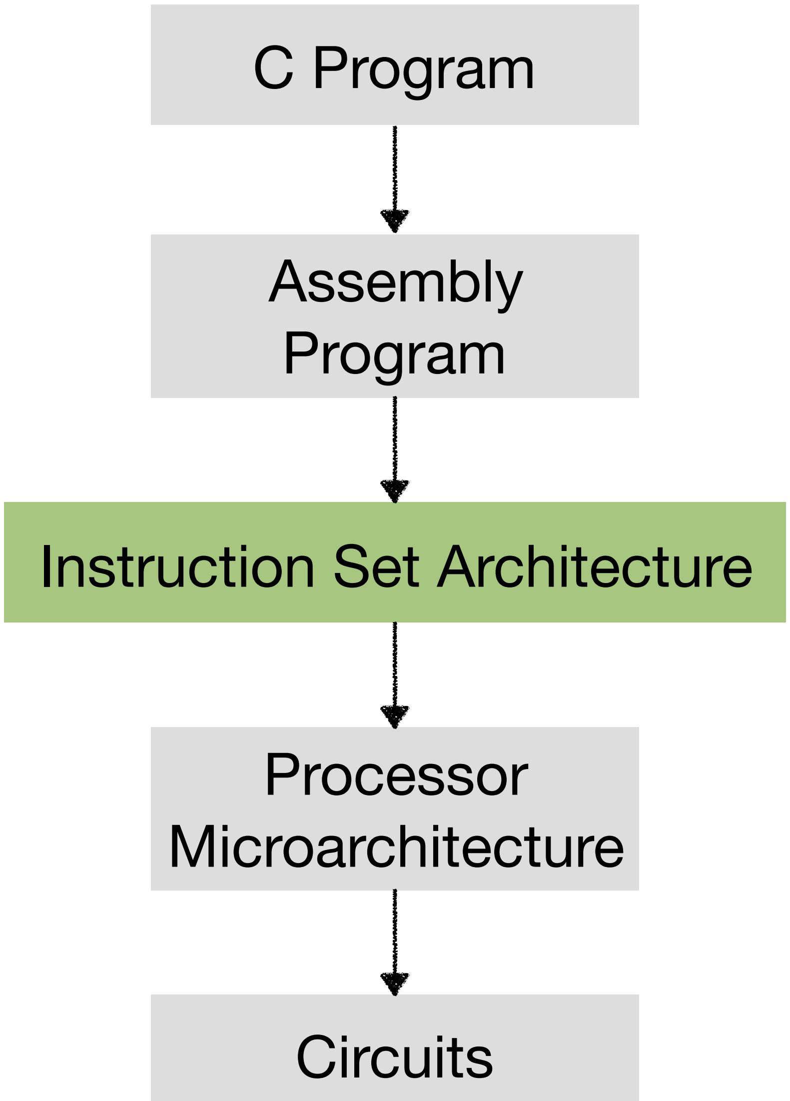
Processor Microarchitecture

Logic gates

Circuits

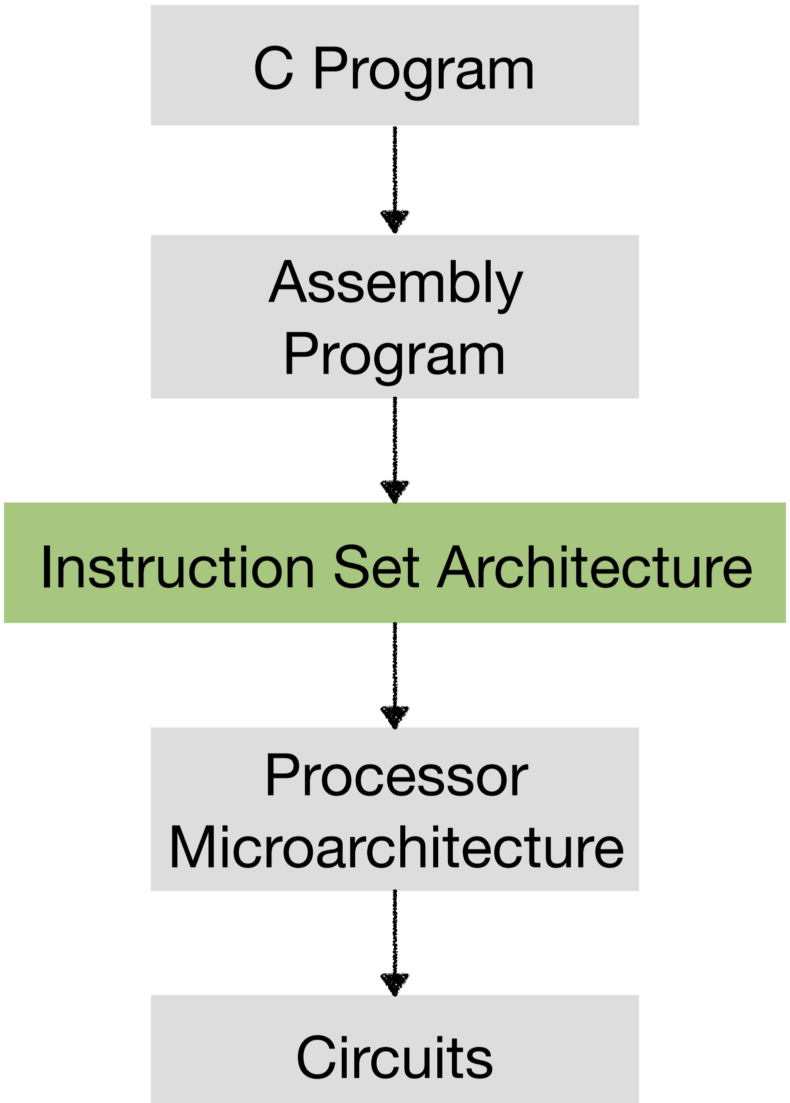
Transistors

So far in 252…



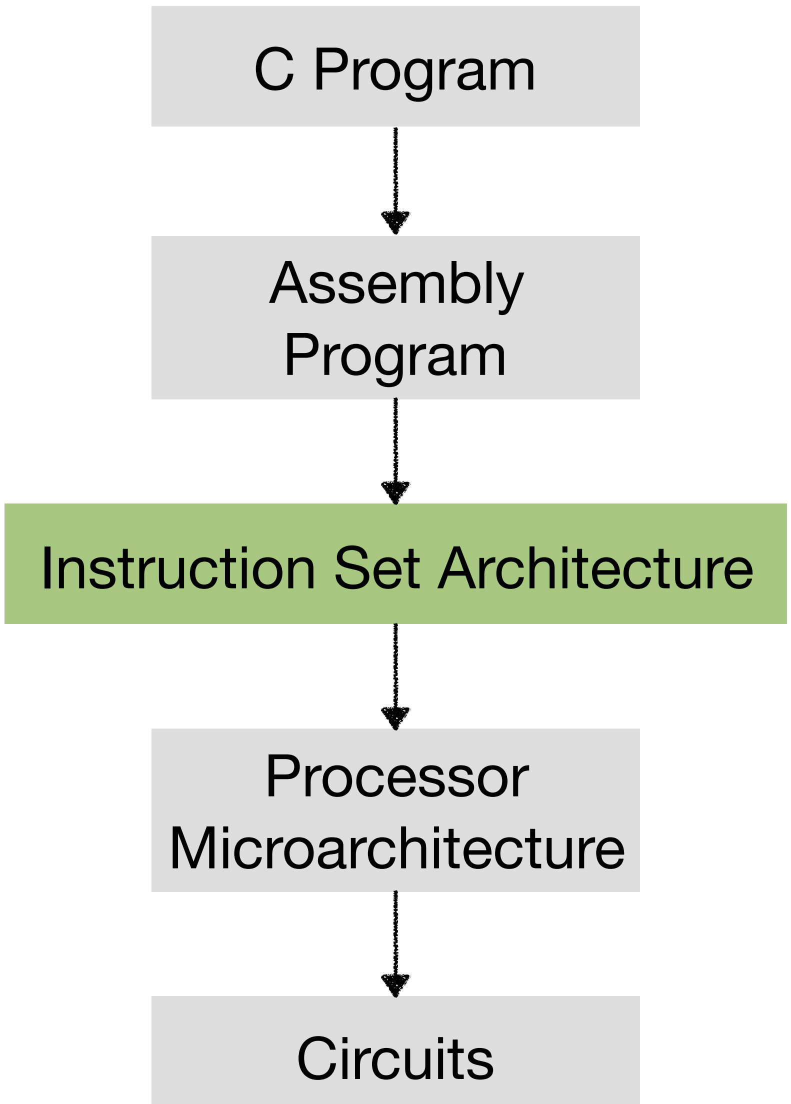
- ISA is the interface between assembly programs and microarchitecture

So far in 252…



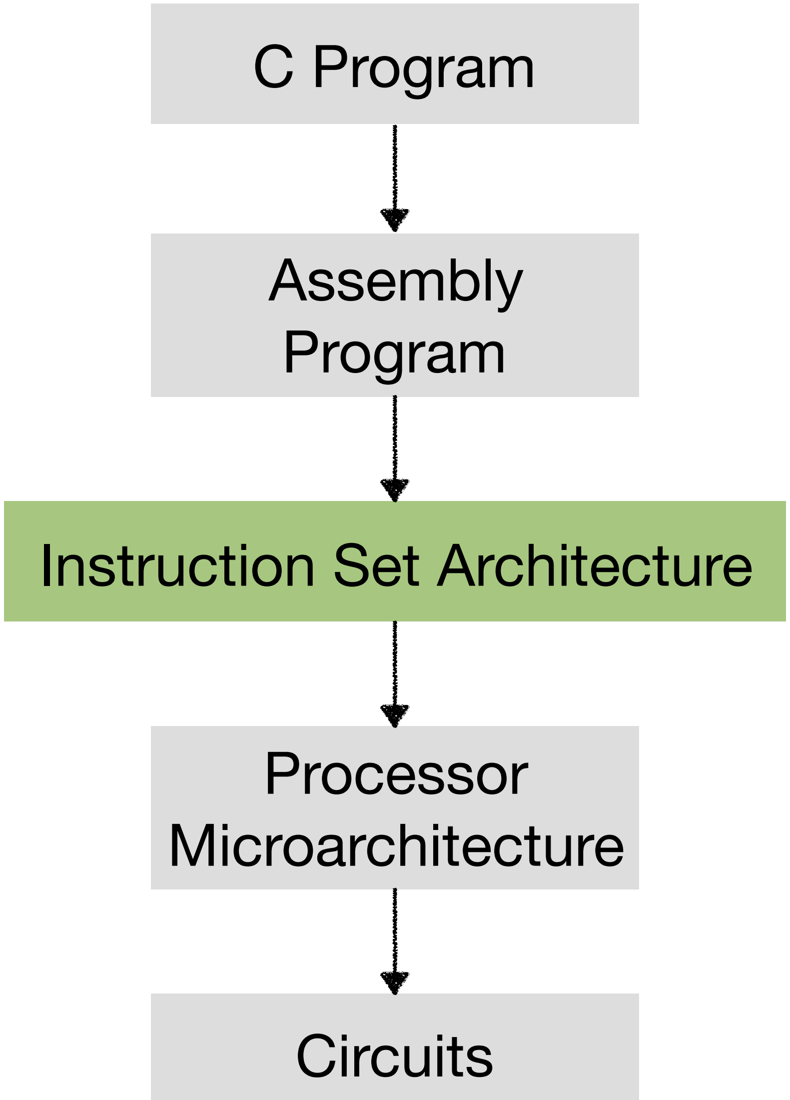
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:

So far in 252…



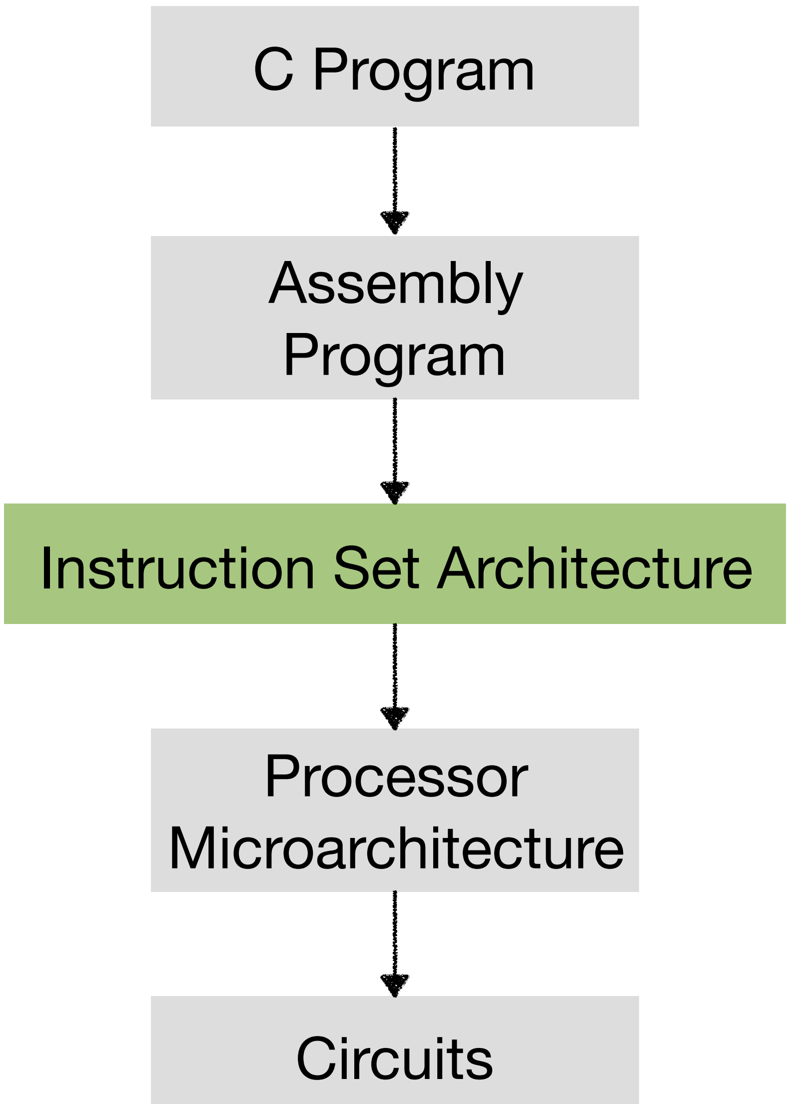
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?

So far in 252…



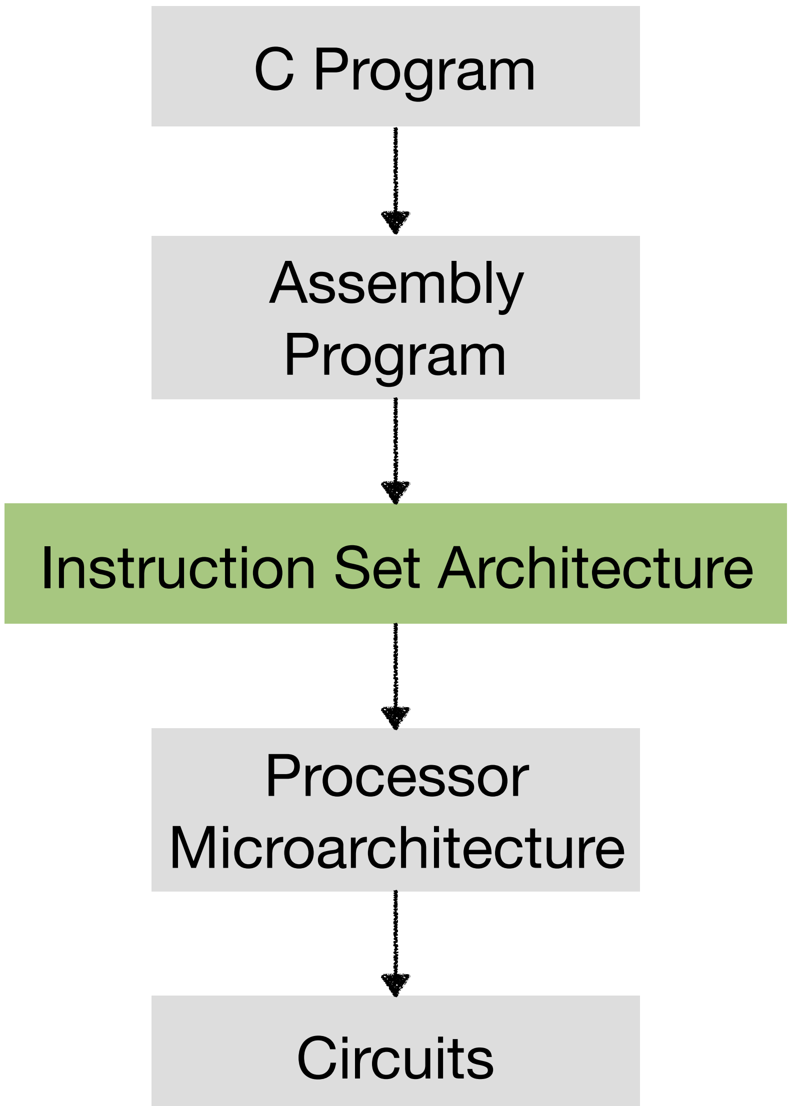
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.

So far in 252…



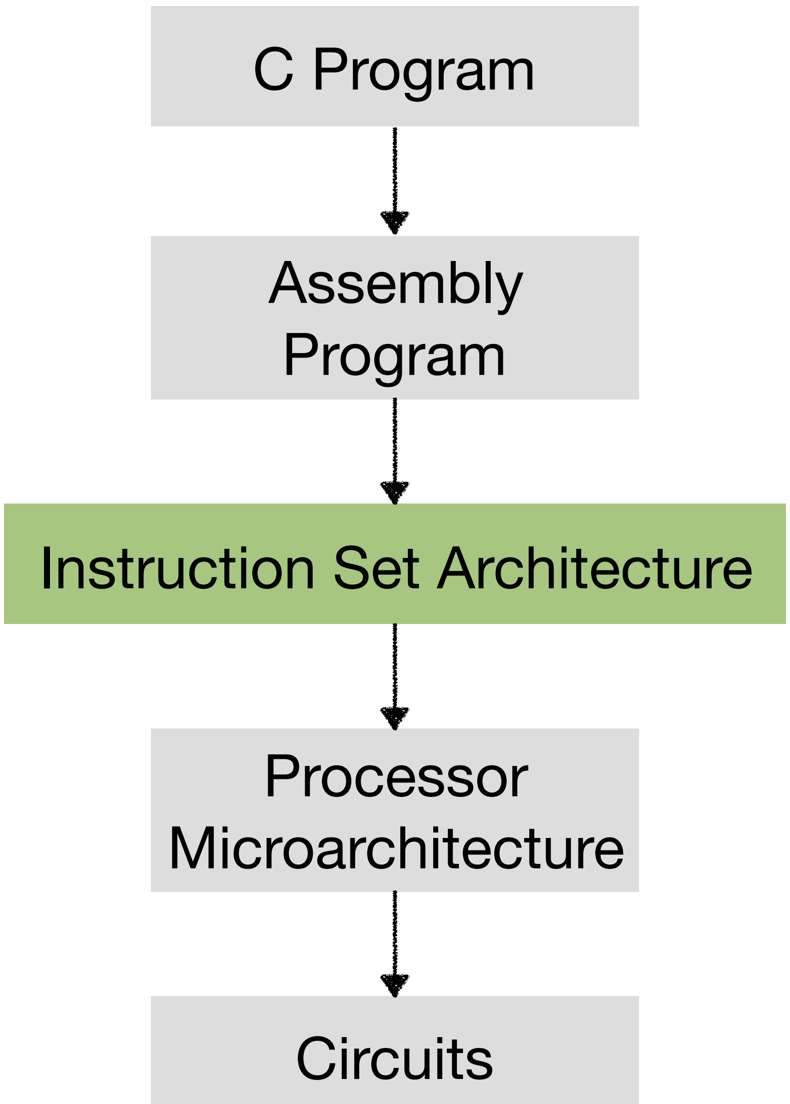
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.
- Microarchitecture view:

So far in 252…



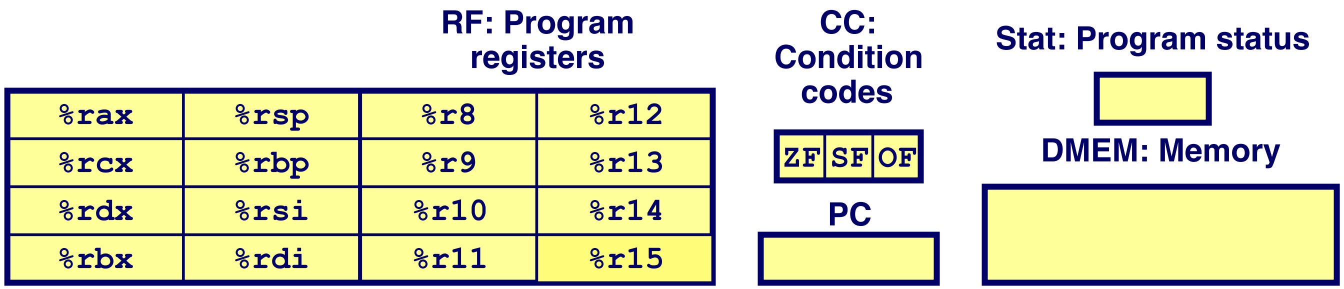
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.
- Microarchitecture view:
  - What hardware needs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

So far in 252…



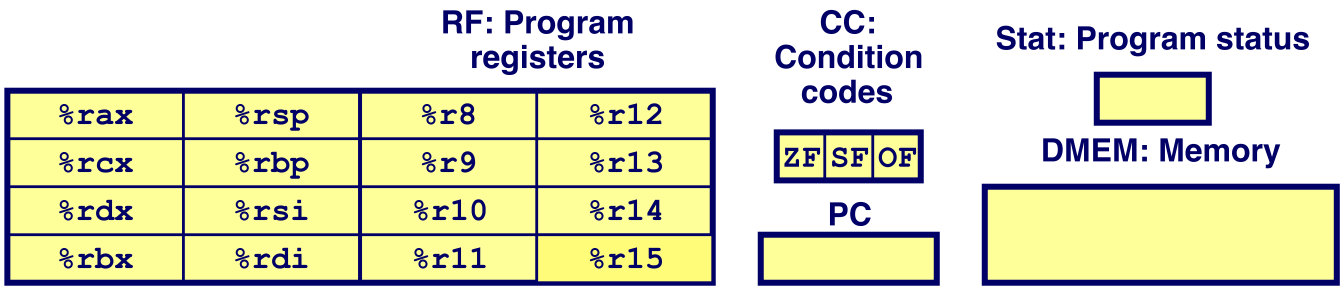
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.
- Microarchitecture view:
  - What hardware needs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

(Simplified) x86 Processor State



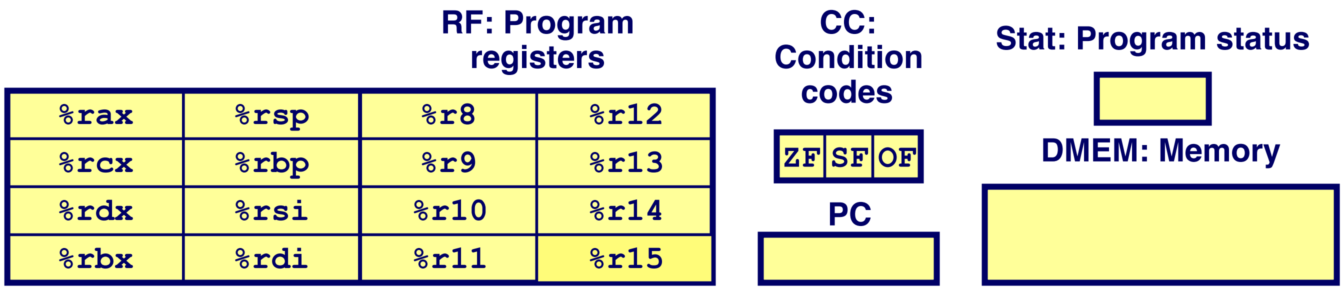
- Processor state is what's visible to assembly programs. Also known as architecture state.

(Simplified) x86 Processor State



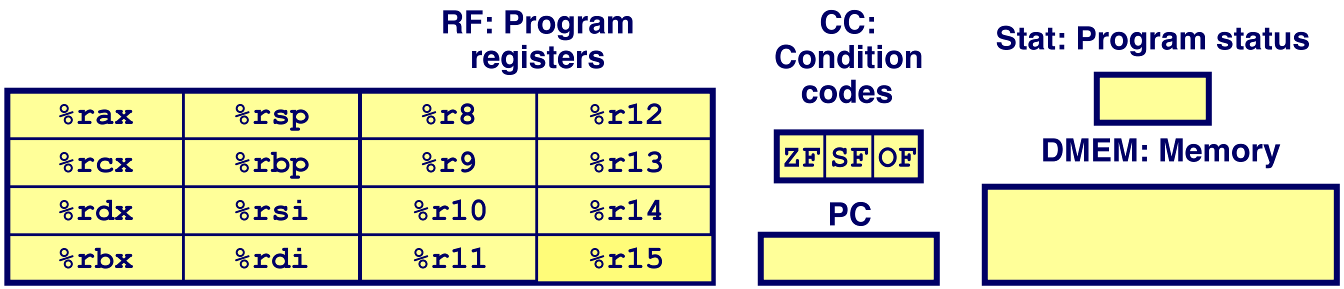
- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 16 registers.

(Simplified) x86 Processor State



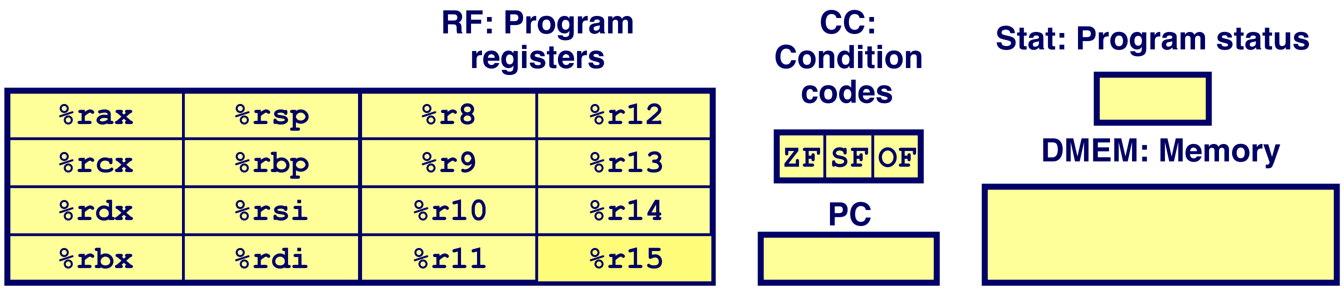
- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 16 registers.
- Condition Codes: Single-bit flags set by arithmetic or logical instructions (ZF, SF, OF)

(Simplified) x86 Processor State



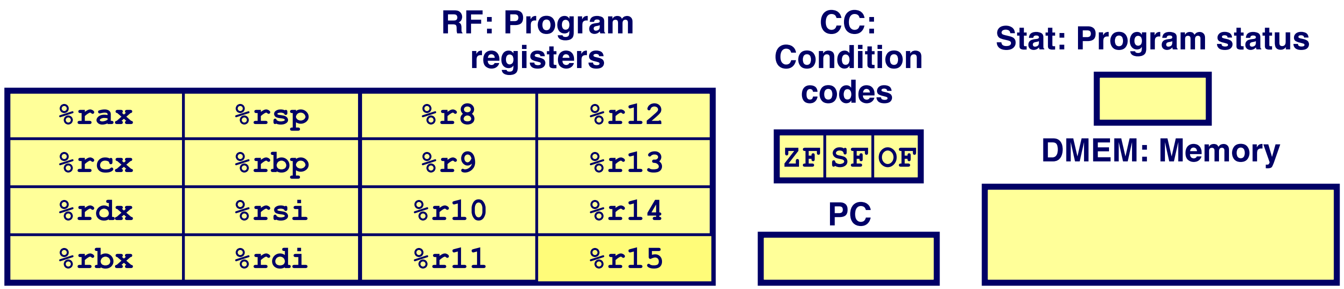
- Processor state is what's visible to assembly programs. Also known as architecture state.
- **Program Registers:** 16 registers.
- **Condition Codes:** Single-bit flags set by arithmetic or logical instructions (ZF, SF, OF)
- **Program Counter:** Indicates address of next instruction

(Simplified) x86 Processor State



- 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

(Simplified) x86 Processor State



- 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?

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

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

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

`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

Y86-64 Instructions

`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

`jmp`

`jle`

`jl`

`je`

`jne`

`jge`

`jg`

Y86-64 Instructions

`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`

Y86-64 Instructions

`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`

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 operand

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 operand
- Or: group similar instructions, use one opcode for them, and then use more bits to indicate specific instructions within a group.

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 operand
- 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

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 operand
- 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.

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 operand
- 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 operand
  - 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

`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

Instruction Example

Addition Instruction

Assembly Form

`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

Instruction Example

Addition Instruction

Assembly Form

Encoded 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

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

Move Instructions

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								

Move Instructions

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								

`irmovq $0xabcd, %rdx`

Move Instructions

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	V							
`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								

Move Instructions

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	V							
`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								

`rmmovq %rsi,0x41c(%rsp)`

Move Instructions

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	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`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								

Move Instructions

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	V
`rmmovq` rA, D(rB)	4	0	rA	rB	D
`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	

```
mrmovq -12(%rbp),%rcx
```

Move Instructions

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	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`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								

Move Instructions

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	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`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								

The instruction length limits the immediate value and displacement.

Move Instruction Examples

Y86-64

```
irmovq $0xabcd, %rdx
```

Encoding: `30 82 cd ab 00 00 00 00 00 00`

```
rrmovq %rsp, %rbx
```

Encoding: `20 43`

```
mrmovq -12(%rbp),%rcx
```

Encoding: `50 15 f4 ff ff ff ff ff ff ff`

```
rmmovq %rsi,0x41c(%rsp)
```

Encoding: `40 64 1c 04 00 00 00 00 00 00`

Jump/Call Instructions

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	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`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								

Jump/Call Instructions

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	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`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								

`jle .L4`

Jump/Call Instructions

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	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`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								

The assembly would assume a start address of the program, and then calculates the address of each instruction.

`jle .L4`

Jump/Call Instructions

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	V
`rmmovq rA, D(rB)`	4 0	rA rB	D
`mrmovq D(rB), rA`	5 0	rA rB	D
`OPq rA, rB`	6 fn	rA rB	
`jXX Dest`	7 fn	Dest (essentially the target address)	
`call Dest`	8 0		
`ret`	9 0		
`pushq rA`	A 0	rA F	
`popq rA`	B 0	rA F	

The assembly would assume a start address of the program, and then calculates the address of each instruction.

Jump/Call Instructions

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	V							
`rmmovq` rA, D(rB)	4 0	rA rB	D							
`mrmovq` D(rB), rA	5 0	rA rB	D							
`OPq` rA, rB	6 fn	rA rB								
`jXX` Dest	7 fn	Dest (essentially the target address)								
`call` Dest	8 0									
`ret`	9 0									
`pushq` rA	A 0	rA F								
`popq` rA	B 0	rA F								

The assembly would assume a start address of the program, and then calculates the address of each instruction.

`call foo`

Jump/Call Instructions

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	V					
`rmmovq` rA, D(rB)	4	0	rA	rB	D					
`mrmovq` D(rB), rA	5	0	rA	rB	D					
`OPq` rA, rB	6	fn	rA	rB						
`jXX` Dest	7	fn	Dest (essentially the target address)							
`call` Dest	8	0	Dest (essentially the start address of the callee)							
`ret`	9	0								
`pushq` rA	A	0	rA	F						
`popq` rA	B	0	rA	F						

The assembly would assume a start address of the program, and then calculates the address of each instruction.

How Does An Assembler Work?

`rmmovq` rA, D(rB)	4	0	rA	rB	D
`OPq` rA, rB	6	fn	rA	rB	
`jXX` Dest	7	fn	Dest		
`call` Dest	8	0	Dest		

How Does An Assembler Work?

Instruction					
`rmmovq rA, D(rB)`	4	0	rA	rB	D
`OPq rA, rB`	6	fn	rA	rB	
`jXX Dest`	7	fn	Dest		
`call Dest`	8	0	Dest		

```
0x100   <foo>   rmmovq %rsi,0x41c(%rsp)
                ... ...
                ret
                addq %rax,%rsi
                call <foo>
                jmp .L0
                ... ...
.L0             irmovq $0xabcd, %rdx
```

How Does An Assembler Work?

`rmmovq` rA, D(rB)	4	0	rA	rB	D
`OPq` rA, rB	6	fn	rA	rB	
`jXX` Dest	7	fn	Dest		
`call` Dest	8	0	Dest		

```
0x100   <foo>   rmmovq %rsi,0x41c(%rsp)     40 64 1c 04 00 00 00 00 00 00
                … …
                ret
                addq %rax,%rsi
                call <foo>
                jmp .L0
                … …
        .L0     irmovq $0xabcd, %rdx
```

How Does An Assembler Work?

`rmmovq` rA, D(rB)	4	0	rA	rB	D
`OPq` rA, rB	6	fn	rA	rB	
`jXX` Dest	7	fn	Dest		
`call` Dest	8	0	Dest		

```
0x100   <foo>   rmmovq %rsi,0x41c(%rsp)     40 64 1c 04 00 00 00 00 00 00
                ...  ...
                ret                          90
                addq %rax,%rsi
                call <foo>
                jmp .L0
                ...  ...
.L0             irmovq $0xabcd, %rdx
```

How Does An Assembler Work?

rmmovq rA, D(rB)	4	0	rA	rB	D
OPq rA, rB	6	fn	rA	rB	
jXX Dest	7	fn	Dest		
call Dest	8	0	Dest		

0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
		addq %rax,%rsi	60 06
		call <foo>	
		jmp .L0	
		… …	
	.L0	irmovq $0xabcd, %rdx	

How Does An Assembler Work?

`rmmovq` rA, D(rB)	4	0	rA	rB	D
`OPq` rA, rB	6	fn	rA	rB	
`jXX` Dest	7	fn	Dest		
`call` Dest	8	0	Dest		

`0x100`	`<foo>`	`rmmovq %rsi,0x41c(%rsp)`	`40 64 1c 04 00 00 00 00 00 00`
		… …	
		`ret`	`90`
		`addq %rax,%rsi`	`60 06`
		`call <foo>`	`80 00 01 00 00 00 00 00 00`
		`jmp .L0`	
		… …	
	`.L0`	`irmovq $0xabcd, %rdx`	

How Does An Assembler Work?

rmmovq rA, D(rB)	4	0	rA	rB	D
OPq rA, rB	6	fn	rA	rB	
jXX Dest	7	fn	Dest		
call Dest	8	0	Dest		

0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
		addq %rax,%rsi	60 06
		call <foo>	80 00 01 00 00 00 00 00 00
		jmp .L0	70 ????????
		… …	
	.L0	irmovq $0xabcd, %rdx	

How Does An Assembler Work?

Instruction	Encoding
rmmovq rA, D(rB)	4 | 0 | rA | rB | D
OPq rA, rB	6 | fn | rA | rB
jXX Dest	7 | fn | Dest
call Dest	8 | 0 | Dest

Address / Label		Instruction	Bytes
0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
		addq %rax,%rsi	60 06
		call <foo>	80 00 01 00 00 00 00 00 00
		jmp .L0	70 ????????
		… …	
.L0		irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

How Does An Assembler Work?

`rmmovq` rA, D(rB)	4	0	rA	rB	D
`OPq` rA, rB	6	fn	rA	rB	
`jXX` Dest	7	fn	Dest		
`call` Dest	8	0	Dest		

0x100	**<foo>**	**rmmovq %rsi,0x41c(%rsp)**	**40 64 1c 04 00 00 00 00 00 00**
		… …	
		ret	**90**
		addq %rax,%rsi	**60 06**
		call <foo>	**80 00 01 00 00 00 00 00 00**
		jmp .L0	**70 ????????**
		… …	
	.L0	**irmovq $0xabcd, %rdx**	**30 f2 cd ab 00 00 00 00 00 00**

0x100 + the lengths of all instructions in-between

How Does An Assembler Work?

`rmmovq` rA, D(rB)	4	0	rA	rB	D
`OPq` rA, rB	6	fn	rA	rB	
`jXX` Dest	7	fn	Dest		
`call` Dest	8	0	Dest		

0x100	**<foo>**	**rmmovq %rsi,0x41c(%rsp)**	**40 64 1c 04 00 00 00 00 00 00**
		… …	
		ret	**90**
		addq %rax,%rsi	**60 06**
		call <foo>	**80 00 01 00 00 00 00 00 00**
		jmp .L0	**70 ????????**
		… …	
0x200	**.L0**	**irmovq $0xabcd, %rdx**	**30 f2 cd ab 00 00 00 00 00 00**

0x100 + the lengths of all instructions in-between

How Does An Assembler Work?

`rmmovq rA, D(rB)`	4	0	rA	rB	D
`OPq rA, rB`	6	fn	rA	rB	
`jXX Dest`	7	fn	Dest		
`call Dest`	8	0	Dest		

Address	Label	Instruction	Encoding
0x100	<foo>	`rmmovq %rsi,0x41c(%rsp)`	40 64 1c 04 00 00 00 00 00 00
		… …	
		`ret`	90
		`addq %rax,%rsi`	60 06
		`call <foo>`	80 00 01 00 00 00 00 00 00
		`jmp .L0`	70 00 02 00 00 00 00 00 00
		… …	
0x200	.L0	`irmovq $0xabcd, %rdx`	30 f2 cd ab 00 00 00 00 00 00

0x100 + the lengths of all instructions in-between

How Does An Assembler Work?

- The assembler is a program that translates assembly code to binary code
- The OS tells the assembler the start address of the code (sort of…)
- Translate the assembly program line by line
- Need to build a “label map” that maps each label to its address

Address	Label	Instruction	Encoding
0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
0x100 + the lengths of all instructions in-between		addq %rax,%rsi	60 06
		call <foo>	80 00 01 00 00 00 00 00 00
		jmp .L0	70 00 02 00 00 00 00 00 00
		… …	
0x200	.L0	irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

Jump Instructions

Jump Unconditionally

`jmp` **Dest**	7	0	Dest

Jump When Less or Equal

`jle` **Dest**	7	1	Dest

Jump When Less

`jl` **Dest**	7	2	Dest

Jump When Equal

`je` **Dest**	7	3	Dest

Jump When Not Equal

`jne` **Dest**	7	4	Dest

Jump When Greater or Equal

`jge` **Dest**	7	5	Dest

Jump When Greater

`jg` **Dest**	7	6	Dest

Subroutine Call and Return

`call` **Dest**	8	0	Dest

- Push address of next instruction onto stack
- Start executing instructions at Dest
- Like x86-64

`ret`	9	0

- Pop value from stack
- Use as address for next instruction
- Like x86-64

One More Complication…

Byte	0	1	2	3	4	5	6	7	8	9	
`jXX` Dest	7	fn	Dest (essentially the target address)								**`jle .L4`**
`call` Dest	8	0	Dest (essentially the start address of the callee)								**`call foo`**

One More Complication…

Byte	0	1	2	3	4	5	6	7	8	9	
`jXX` Dest	7	fn	Dest (essentially the target address)								`jle .L4`
`call` Dest	8	0	Dest (essentially the start address of the callee)								`call foo`

- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?

One More Complication…

Byte	0	1	2	3	4	5	6	7	8	9	
`jXX` Dest	7	fn	Dest (essentially the target address)								**`jle .L4`**
`call` Dest	8	0	Dest (essentially the start address of the callee)								**`call foo`**

- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?
- One alternative: use a super long instruction encoding format.
  - Simple to encode, but space inefficient (waste bits for jumps to short addr.)

One More Complication…

Byte	0	1	2–9	
`jXX` Dest	7	fn	Dest (essentially the target address)	**`jle .L4`**
`call` Dest	8	0	Dest (essentially the start address of the callee)	**`call foo`**

- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?
- One alternative: use a super long instruction encoding format.
  - Simple to encode, but space inefficient (waste bits for jumps to short addr.)
- Another alternative: encode the relative address, not the absolute address
  - E.g., encode (.L4 - current address) in Dest

Using Relative Addresses for Jumps

- What if the ISA encoding uses relative address for jump and call?

0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
		addq %rax,%rsi	60 06
0x180		call <foo>	80 06 00 01 00 00 00 00 00 00
0x185		jmp .L0	70 00 02 00 00 00 00 00 00
		… …	
0x200	.L0	irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

Using Relative Addresses for Jumps

- What if the ISA encoding uses relative address for jump and call?

Address	Label	Instruction	Encoding
0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
	relative addr: -0x80	ret	90
		addq %rax,%rsi	60 06
0x180		call <foo>	80 06 00 01 00 00 00 00 00 00
0x185		jmp .L0	70 00 02 00 00 00 00 00 00
		… …	
0x200	.L0	irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

Using Relative Addresses for Jumps

- What if the ISA encoding uses relative address for jump and call?

0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
		addq %rax,%rsi	60 06
0x180		call <foo>	80 00 00 00 11 11 11 11 11
0x185		jmp .L0	70 00 02 00 00 00 00 00 00
		… …	
0x200	.L0	irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

relative addr:
-0x80

Using Relative Addresses for Jumps

- What if the ISA encoding uses relative address for jump and call?

Address	Label	Instruction	Encoding
0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
		addq %rax,%rsi	60 06
0x180		call <foo>	80 00 00 00 11 11 11 11 11
0x185		jmp .L0	70 00 02 00 00 00 00 00 00
		… …	
0x200	.L0	irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

relative addr: -0x80

0x7B

Using Relative Addresses for Jumps

- What if the ISA encoding uses relative address for jump and call?

Address	Label	Instruction	Encoding
0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
		ret	90
		addq %rax,%rsi	60 06
0x180		call <foo>	80 00 00 00 11 11 11 11 11
0x185		jmp .L0	70 7B 00 00 00 00 00 00 00
		… …	
0x200	.L0	irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

relative addr: -0x80

0x7B

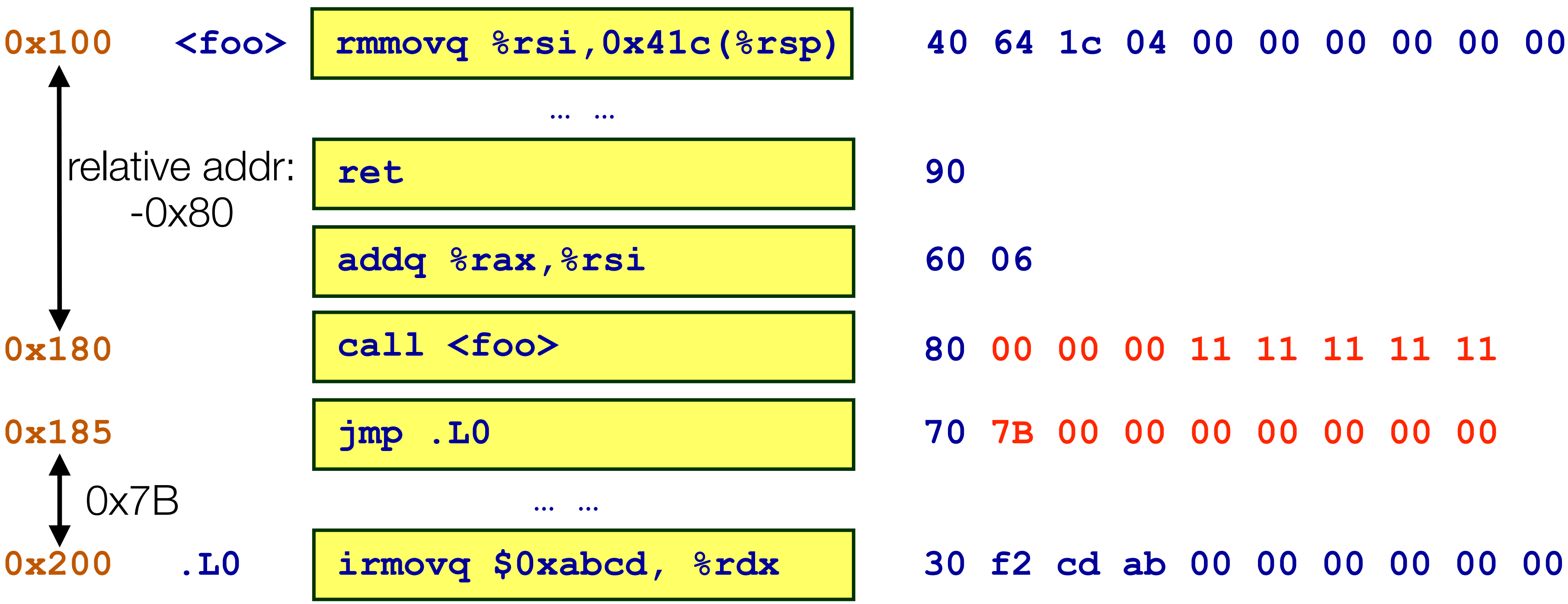
Using Relative Addresses for Jumps

- What if the ISA encoding uses relative address for jump and call?
- If we use relative address, the exact start address of the code doesn't matter. Why?

Address	Label	Instruction	Encoding
0x100	<foo>	rmmovq %rsi,0x41c(%rsp)	40 64 1c 04 00 00 00 00 00 00
		… …	
relative addr: -0x80		ret	90
		addq %rax,%rsi	60 06
0x180		call <foo>	80 00 00 00 11 11 11 11 11
0x185		jmp .L0	70 7B 00 00 00 00 00 00 00
0x7B		… …	
0x200	.L0	irmovq $0xabcd, %rdx	30 f2 cd ab 00 00 00 00 00 00

Using Relative Addresses for Jumps

- What if the ISA encoding uses relative address for jump and call?
- If we use relative address, the exact start address of the code doesn't matter. Why?
- This code is called Position-Independent Code (PIC)



Miscellaneous Instructions

nop	1	0

- Don’t do anything

halt	0	0

- Stop executing instructions
- Usually can’t be executed in the user mode, only by the OS
- Encoding ensures that program hitting memory initialized to zero will halt

Variable Length Instructions

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.
  - Or you can have a combination of both: e.g., 16-bit ISA with 32-bit extensions (e..g, ARM Thumb-extension).

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.
  - Or you can have a combination of both: e.g., 16-bit ISA with 32-bit extensions (e..g, ARM Thumb-extension).
- Advantages of variable length ISAs

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.
  - Or you can have a combination of both: e.g., 16-bit ISA with 32-bit extensions (e..g, ARM Thumb-extension).
- Advantages of variable length ISAs
  - More compact. Some instructions do not need that many bits. (Actually what's the optimal way of encoding instructions in a variable length ISA?)

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.
  - Or you can have a combination of both: e.g., 16-bit ISA with 32-bit extensions (e..g, ARM Thumb-extension).
- Advantages of variable length ISAs
  - More compact. Some instructions do not need that many bits. (Actually what's the optimal way of encoding instructions in a variable length ISA?)
  - Can have arbitrary number of instructions: easy to add new inst.

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.
  - Or you can have a combination of both: e.g., 16-bit ISA with 32-bit extensions (e..g, ARM Thumb-extension).
- Advantages of variable length ISAs
  - More compact. Some instructions do not need that many bits. (Actually what's the optimal way of encoding instructions in a variable length ISA?)
  - Can have arbitrary number of instructions: easy to add new inst.
- What is the down side?

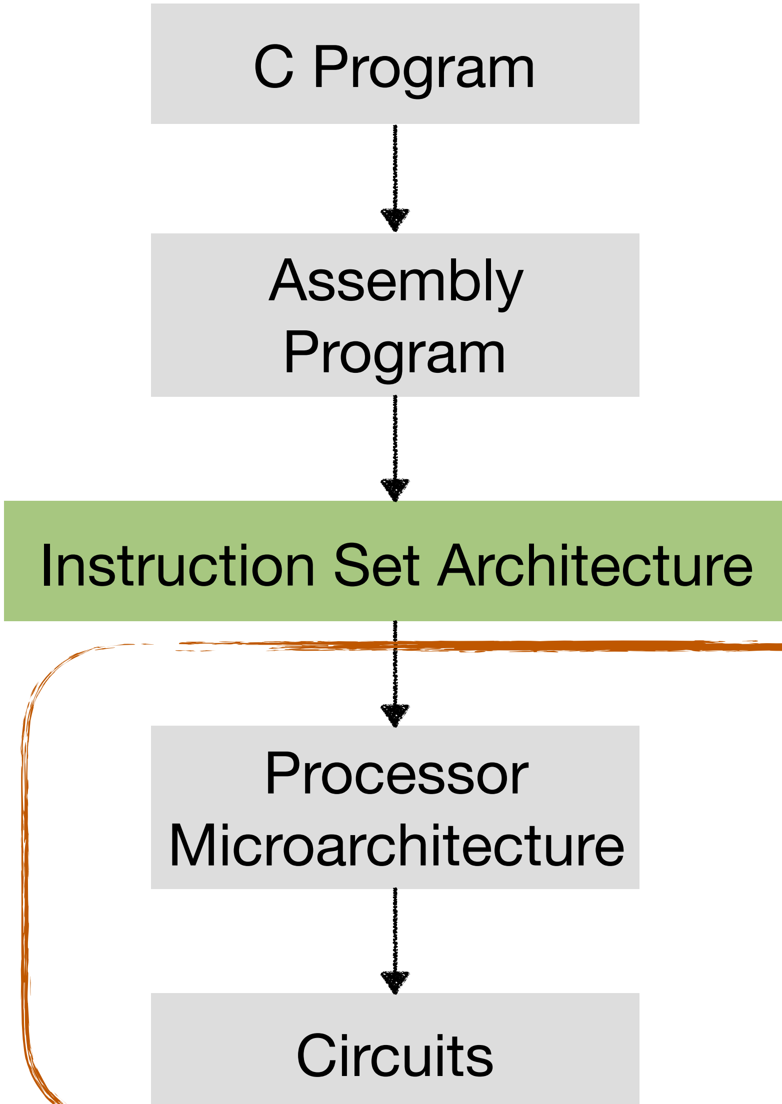
Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.
  - Or you can have a combination of both: e.g., 16-bit ISA with 32-bit extensions (e..g, ARM Thumb-extension).
- Advantages of variable length ISAs
  - More compact. Some instructions do not need that many bits. (Actually what's the optimal way of encoding instructions in a variable length ISA?)
  - Can have arbitrary number of instructions: easy to add new inst.
- What is the down side?
  - Fetch and decode are harder to implement. More on this later.

Variable Length Instructions

- X86 (and Y86) is a variable length ISA (1 to 15 bytes), where different instructions have different lengths.
- There are fixed length ISAs: all instructions have the same length
  - ARM's ISA for micro-controllers have a 4-bit ISA. Very Long Instruction Word (VLIW) ISAs have instructions that are hundreds of bytes long.
  - Or you can have a combination of both: e.g., 16-bit ISA with 32-bit extensions (e..g, ARM Thumb-extension).
- Advantages of variable length ISAs
  - More compact. Some instructions do not need that many bits. (Actually what's the optimal way of encoding instructions in a variable length ISA?)
  - Can have arbitrary number of instructions: easy to add new inst.
- What is the down side?
  - Fetch and decode are harder to implement. More on this later.
- A good writeup showing some of the complexity involved: http://www.c-jump.com/CIS77/CPU/x86/lecture.html

So far in 252…




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

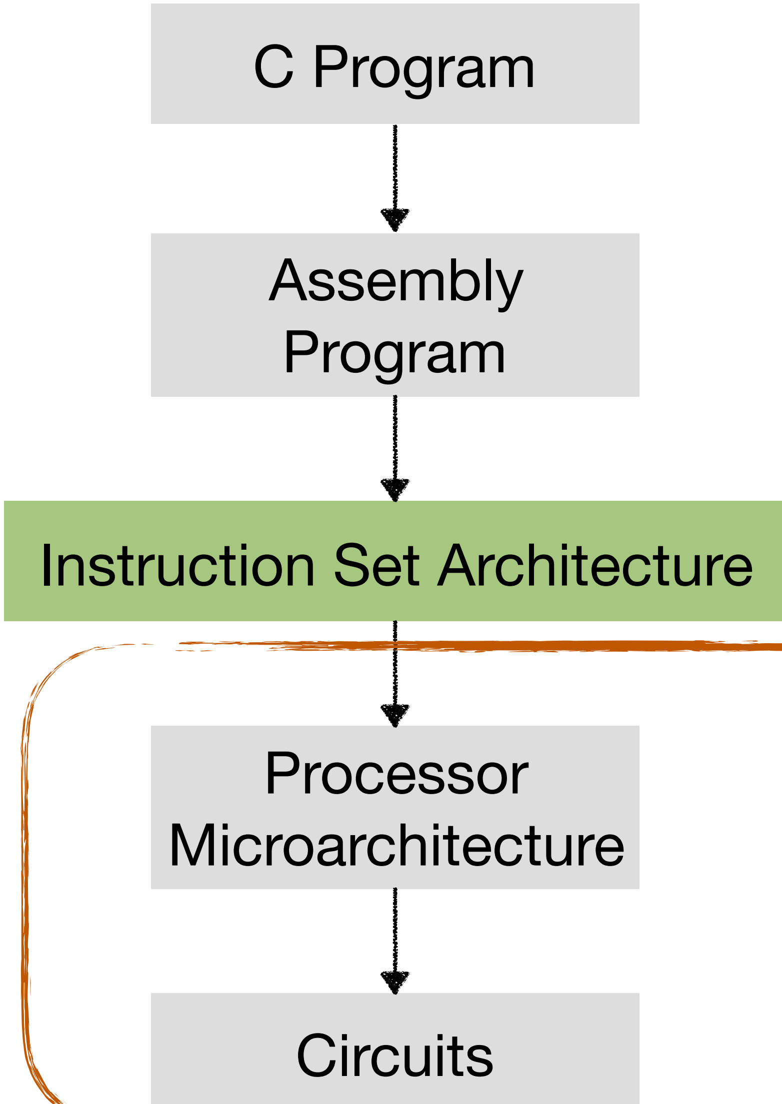
```
movq    %rsi, %rax
imulq   %rdx, %rax
jmp     .done
```

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

Logic gates

Transistors

So far in 252…




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

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

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

and their bit encodings.

Logic gates

Transistors

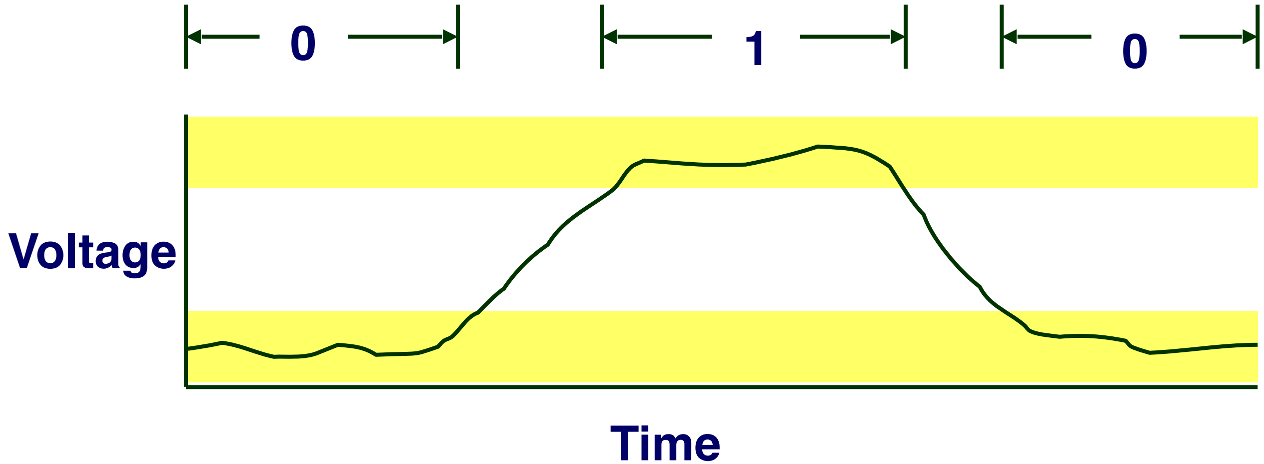
Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

Overview of Circuit-Level Design

- Fundamental Hardware Requirements
  - Communication: How to get values from one place to another. Mainly three electrical **wires**.
  - Computation: **transistors**. Combinational logic.
  - Storage: **transistors**. Sequential logic.
- Circuit design is often abstracted as logic design

Digital Signals



- Extract discrete values from continuous voltage signal
- Simplest version: 1-bit signal
    - Either high range (1) or low range (0)
    - With guard range between them
- Not strongly affected by noise or low quality circuit elements
    - Can make circuits simple, small, and fast

Basic Building Block: Transistors

Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

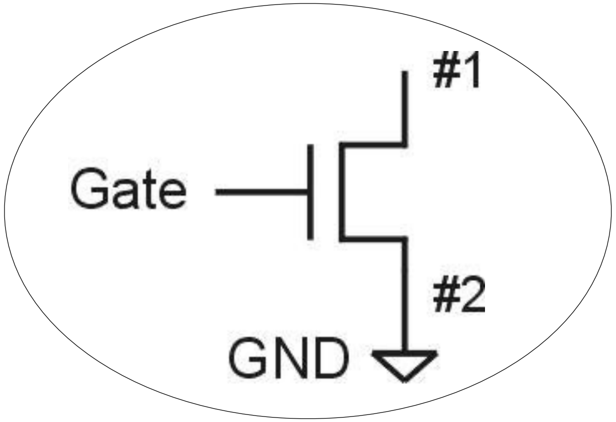
- two types: n-type and p-type

Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type (NMOS)



Terminal #2 must be connected to GND (0V).

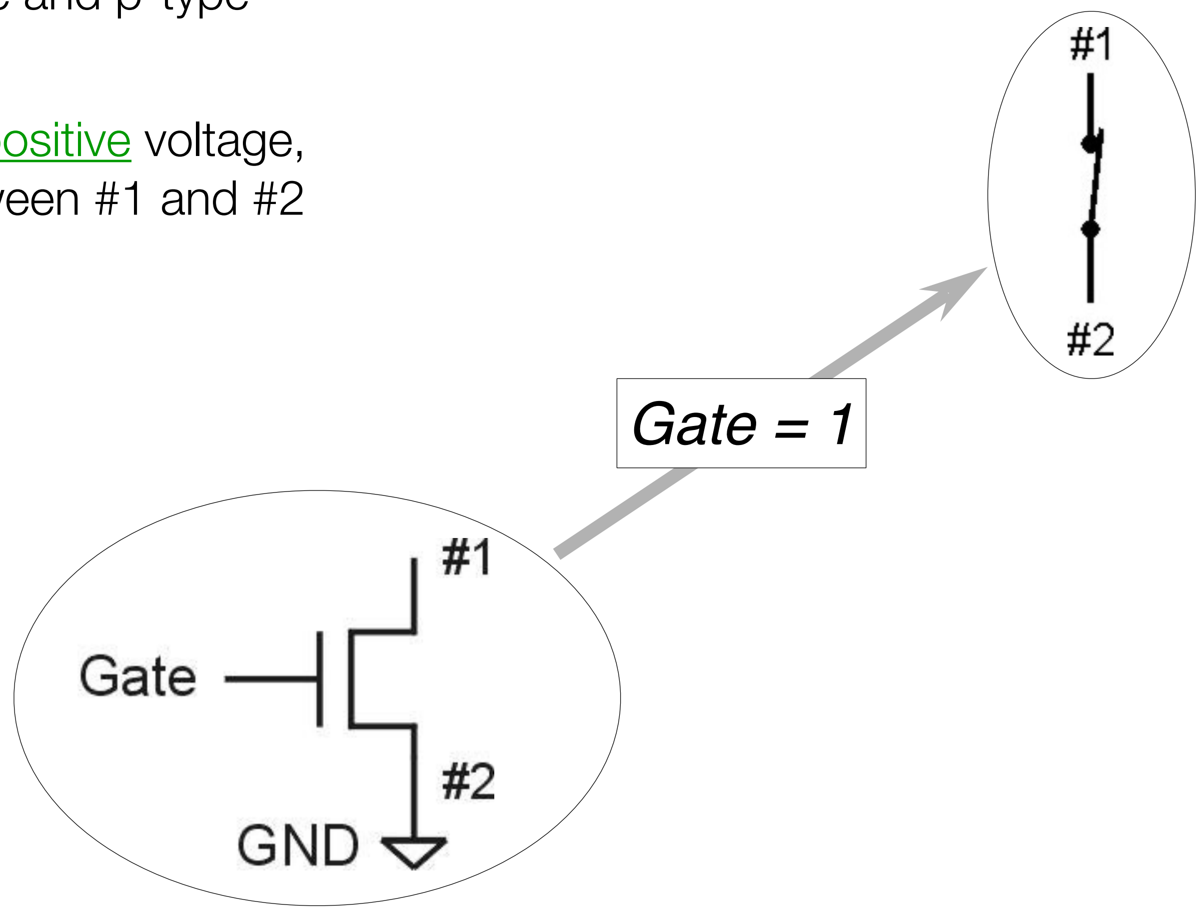
Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type (NMOS)

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)



Terminal #2 must be connected to GND (0V).

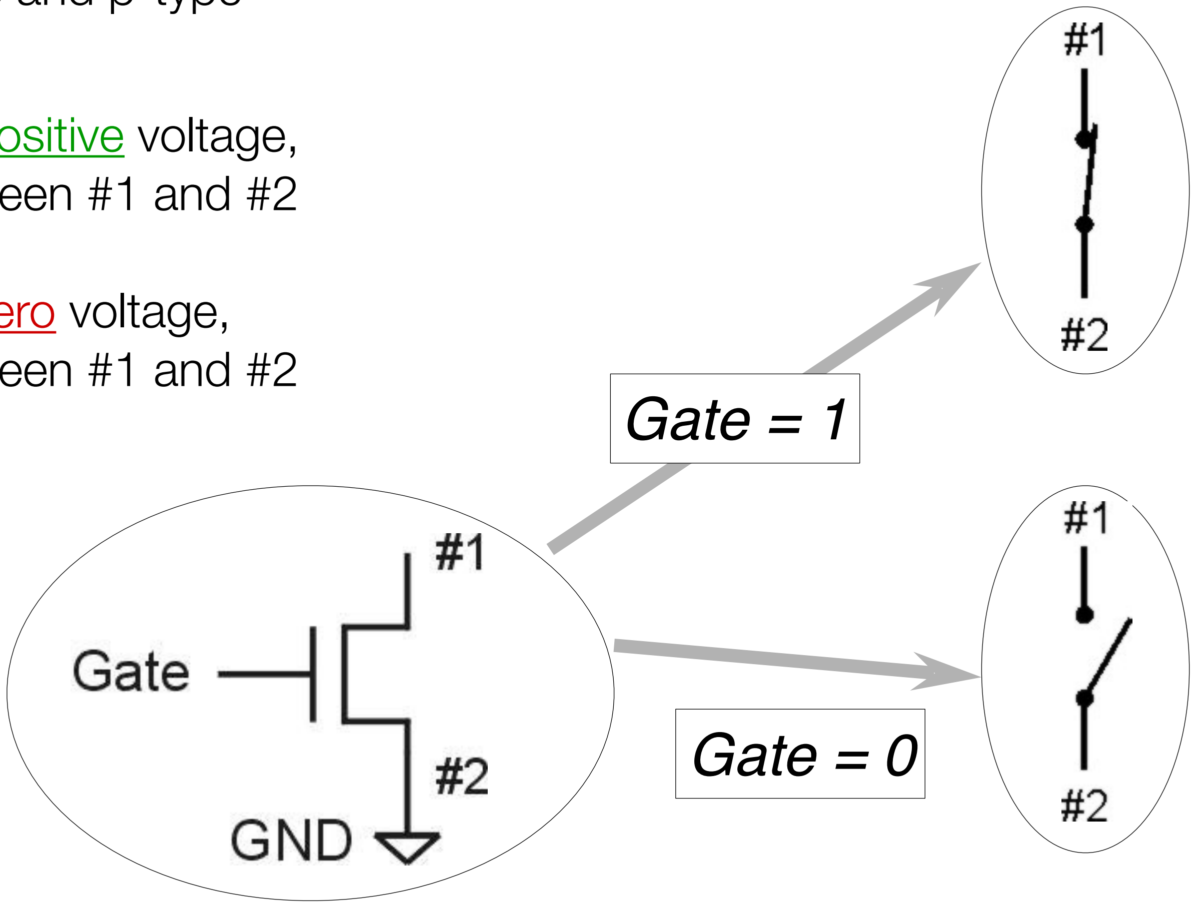
Basic Building Block: Transistors

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type (NMOS)

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)

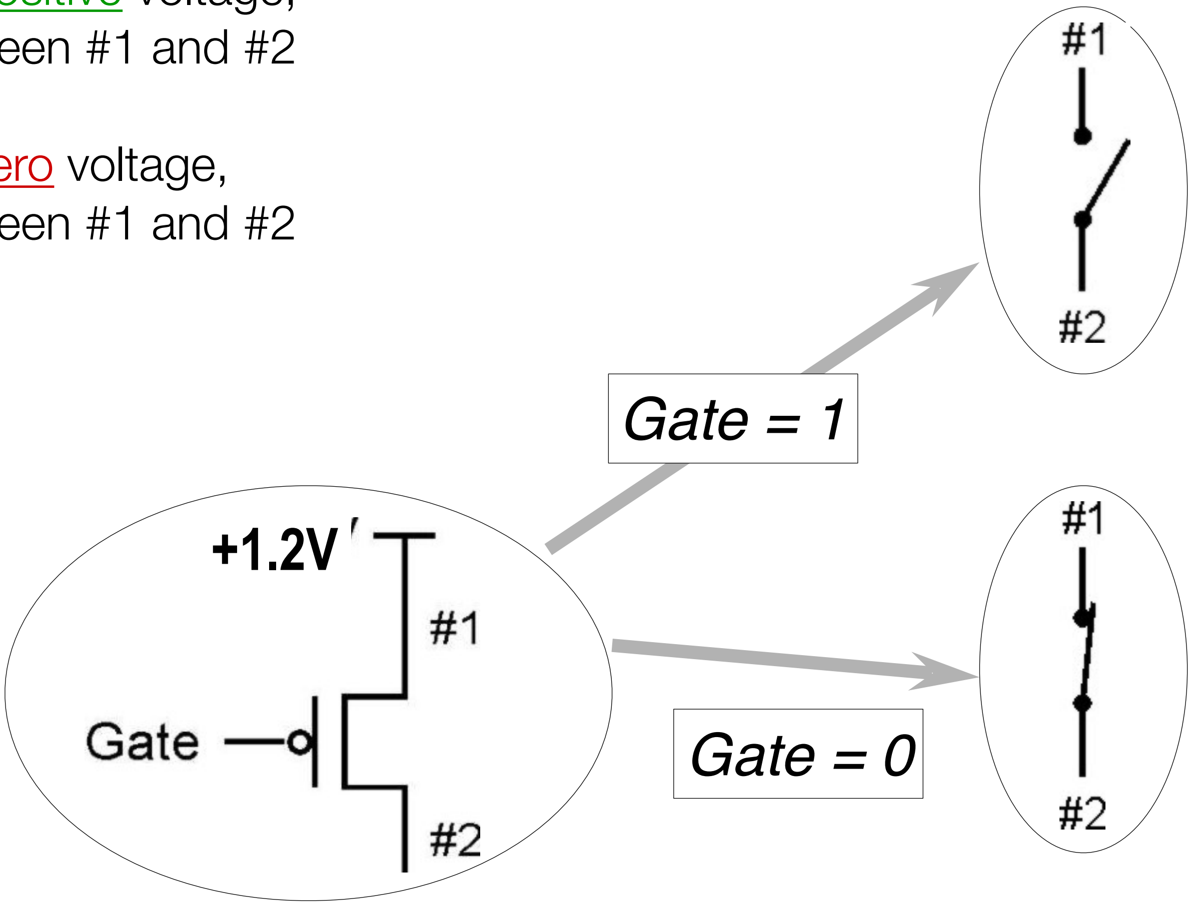


Terminal #2 must be connected to GND (0V).

Basic Building Block: Transistors

p-type is *complementary* to n-type (PMOS)

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)

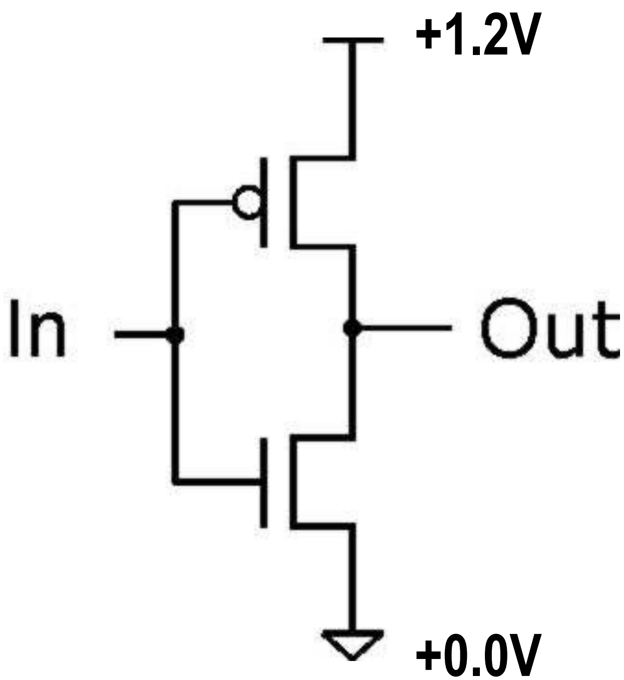


Terminal #1 must be connected to +1.2V

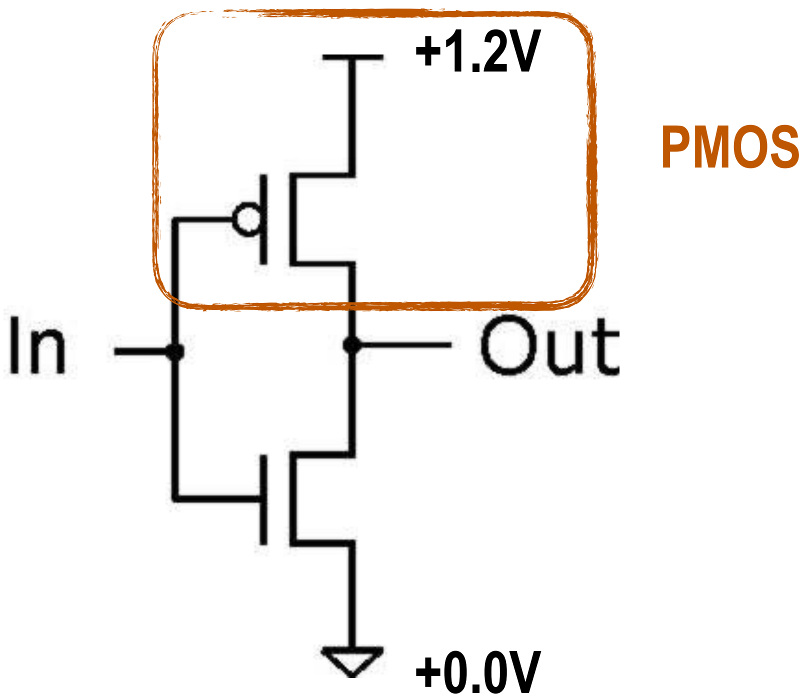
CMOS Circuit

- Complementary MOS
- Uses both n-type and p-type MOS transistors

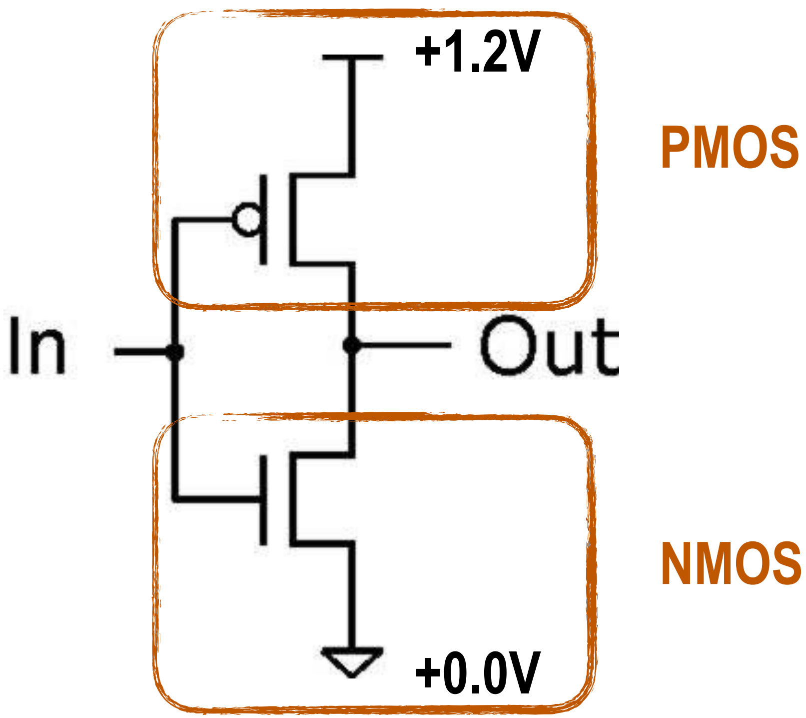
Inverter (NOT Gate)



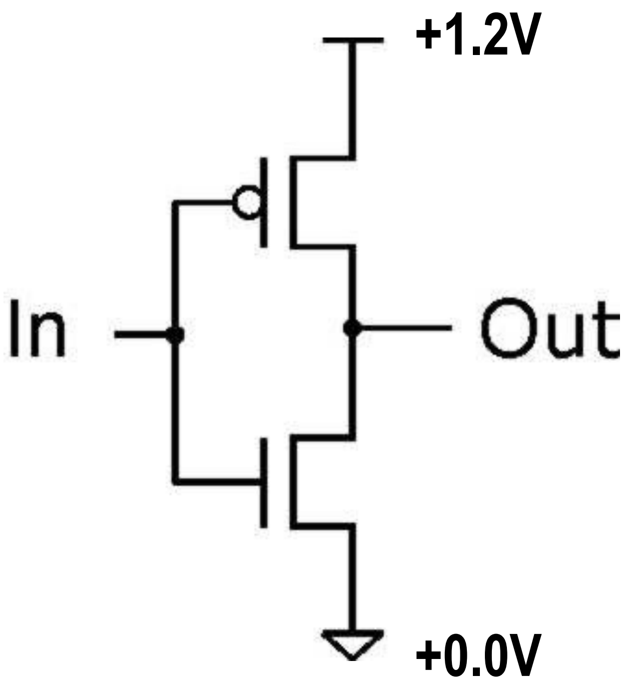
Inverter (NOT Gate)

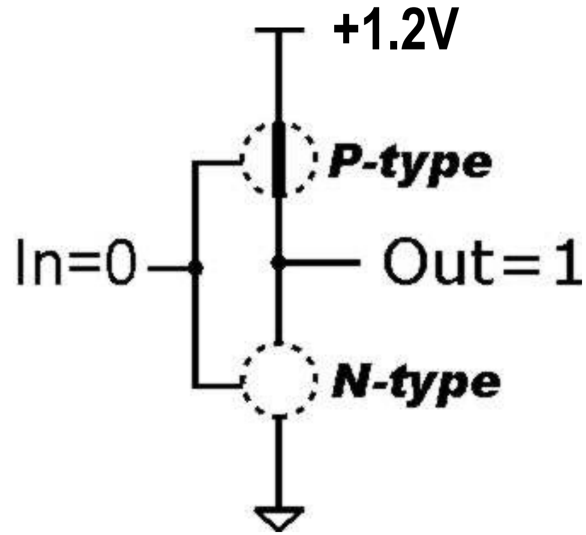


Inverter (NOT Gate)

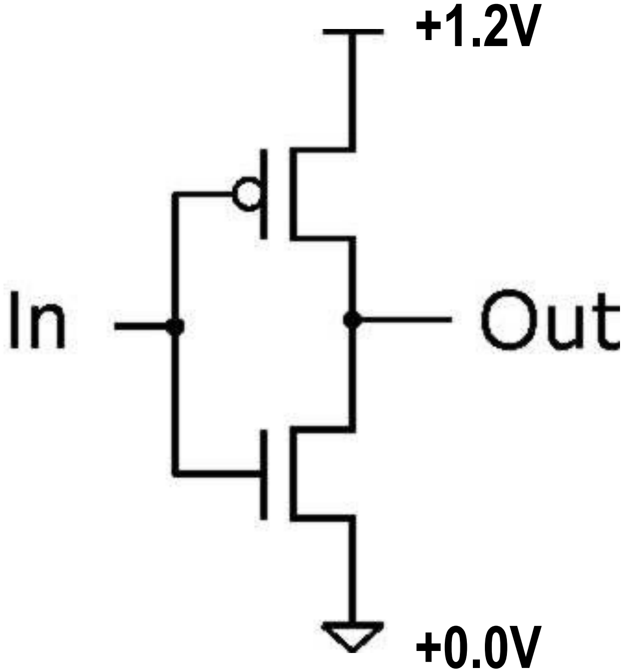


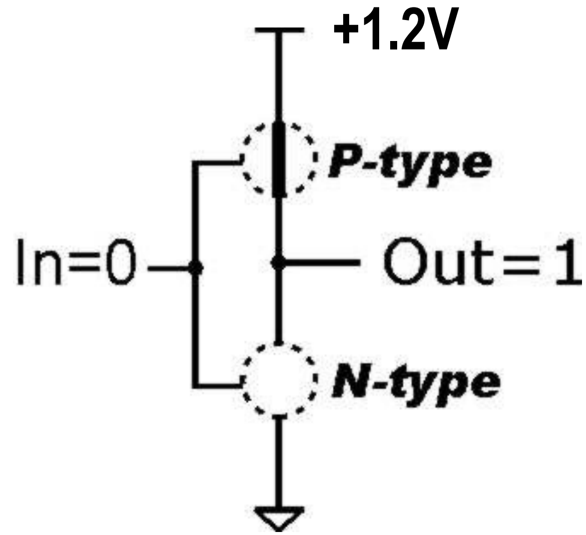
Inverter (NOT Gate)

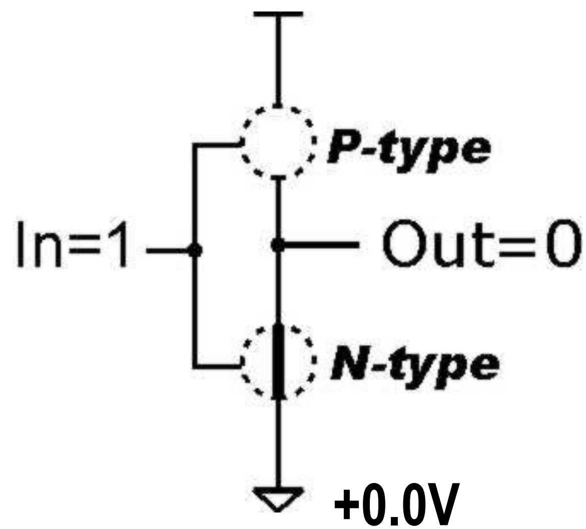




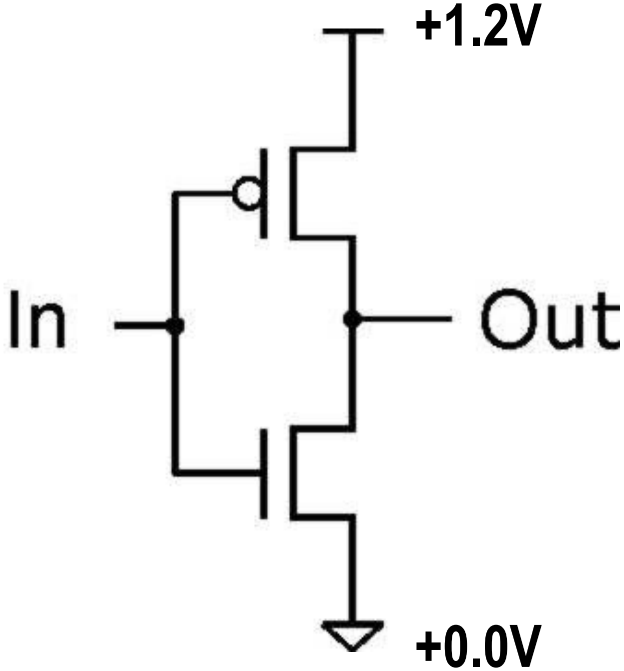
Inverter (NOT Gate)



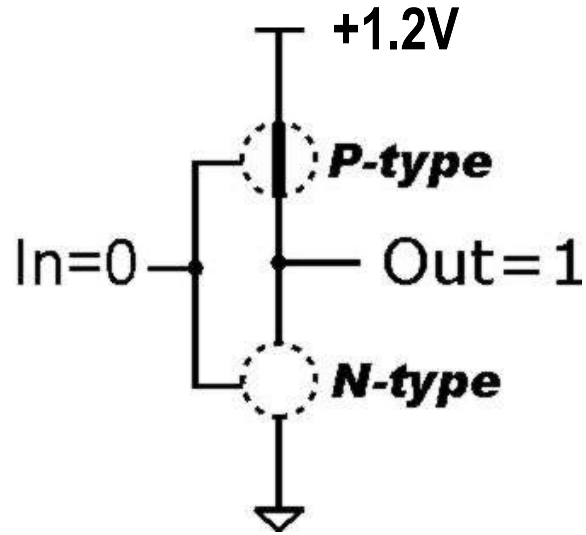


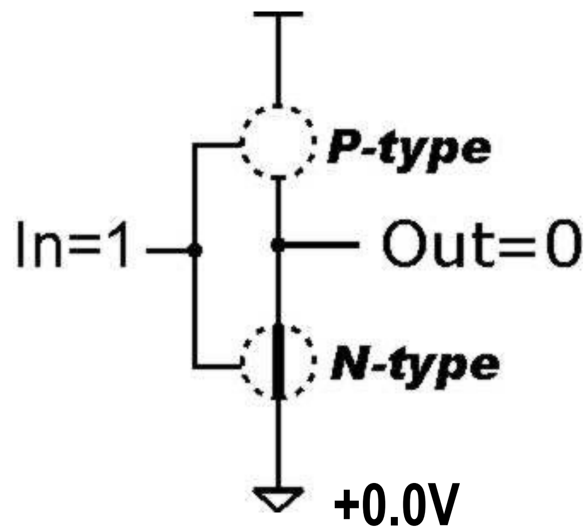


Inverter (NOT Gate)

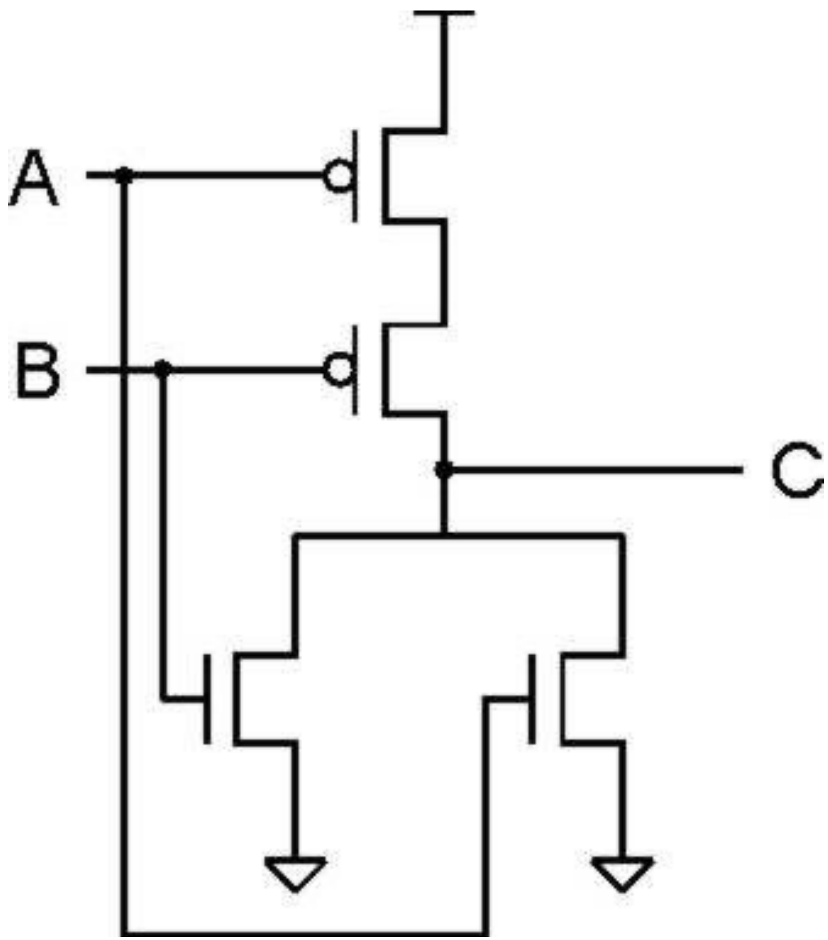


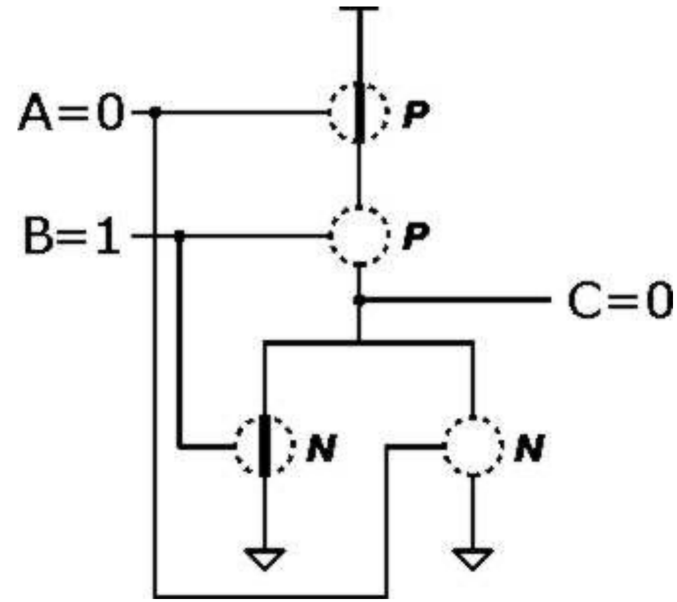
In	Out
0	1
1	0





NOR Gate (NOT + OR)





A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

Basic Logic Gates