

CSC 252: Computer Organization

Spring 2019: Lecture 10

Instructor: Yuhao Zhu


Department of Computer Science
University of Rochester

Action Items:

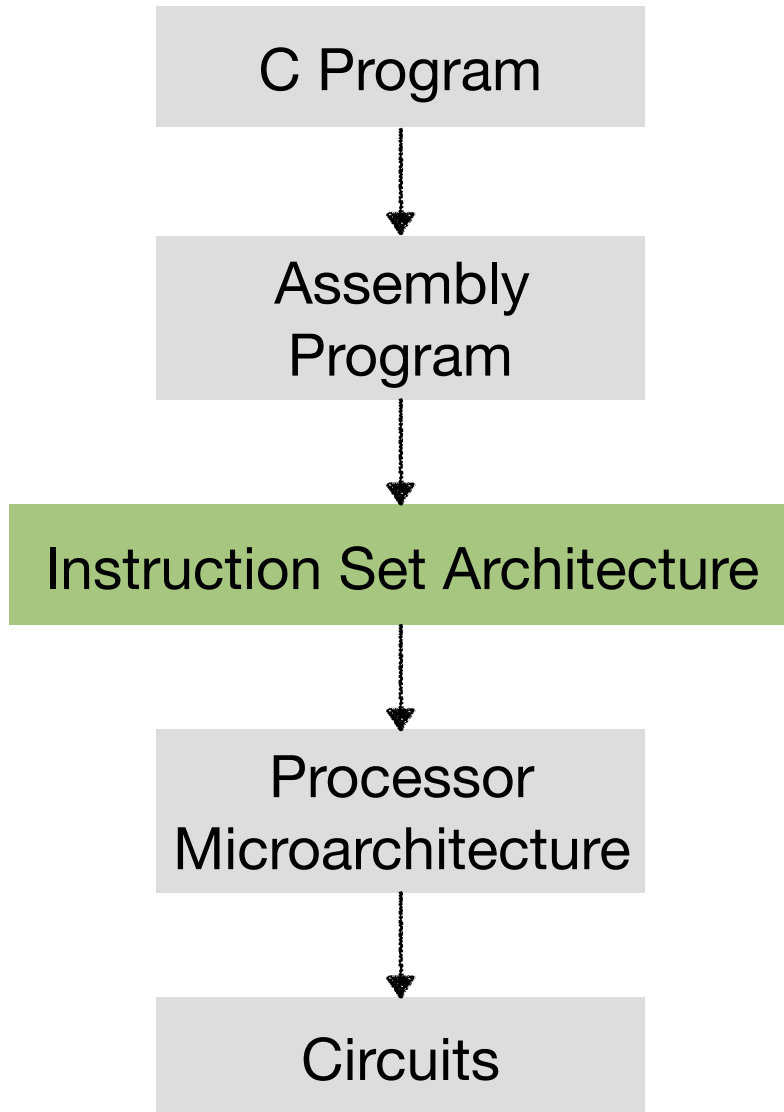
- **Trivia 3 was just due**
- **Assignment 3 is due March 1, midnight**

Announcement

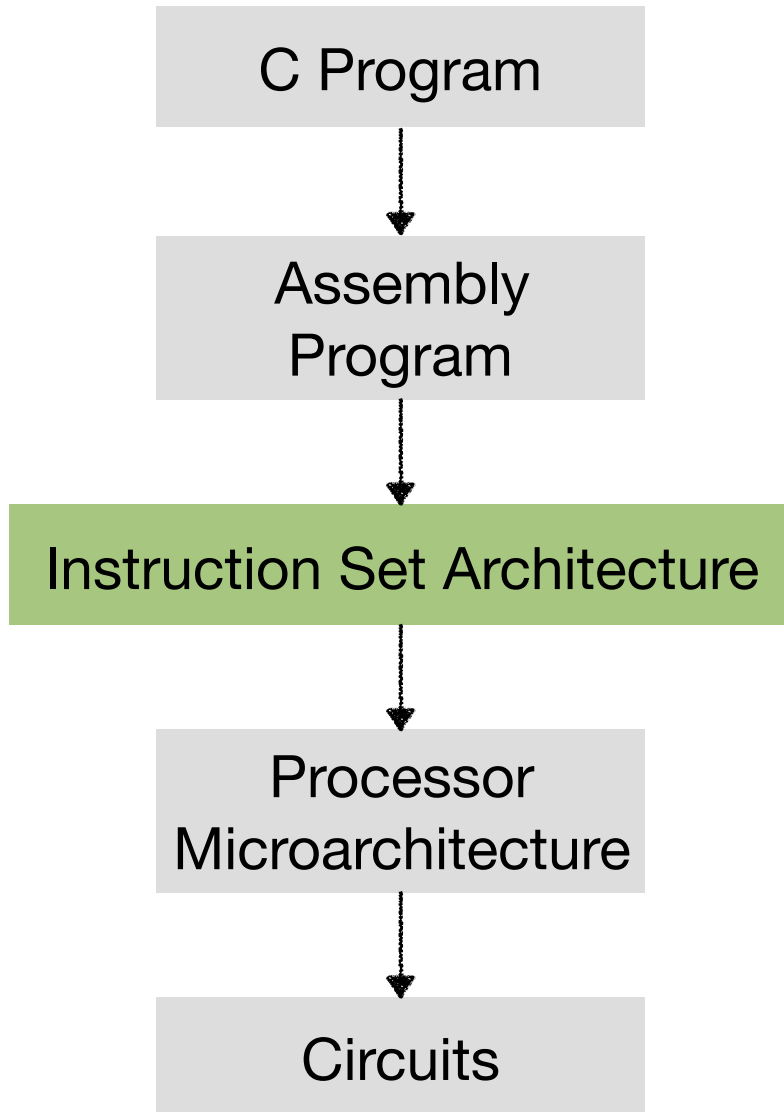
- Programming Assignment 3 is out
 - Due on **March 1, midnight**

18	19	20	21	22	23	24
	Today					
25	26	27	28	Mar 1	2	3
				due		

So far in 252...

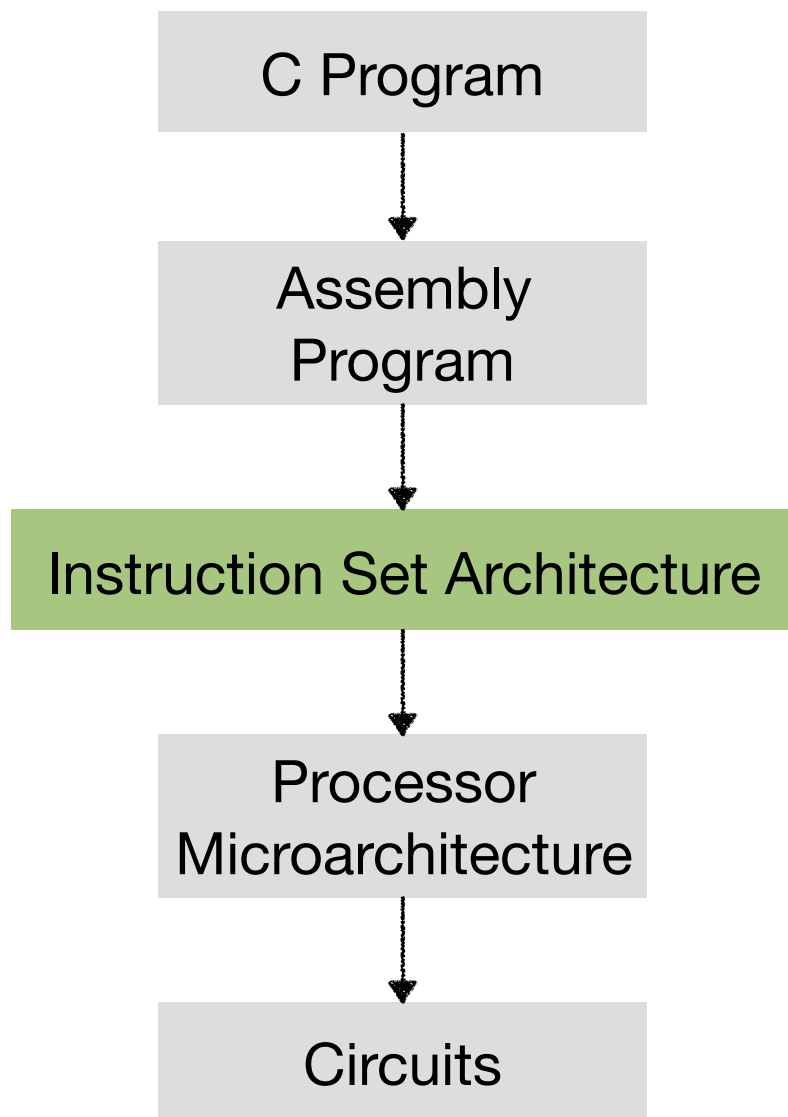


So far in 252...



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

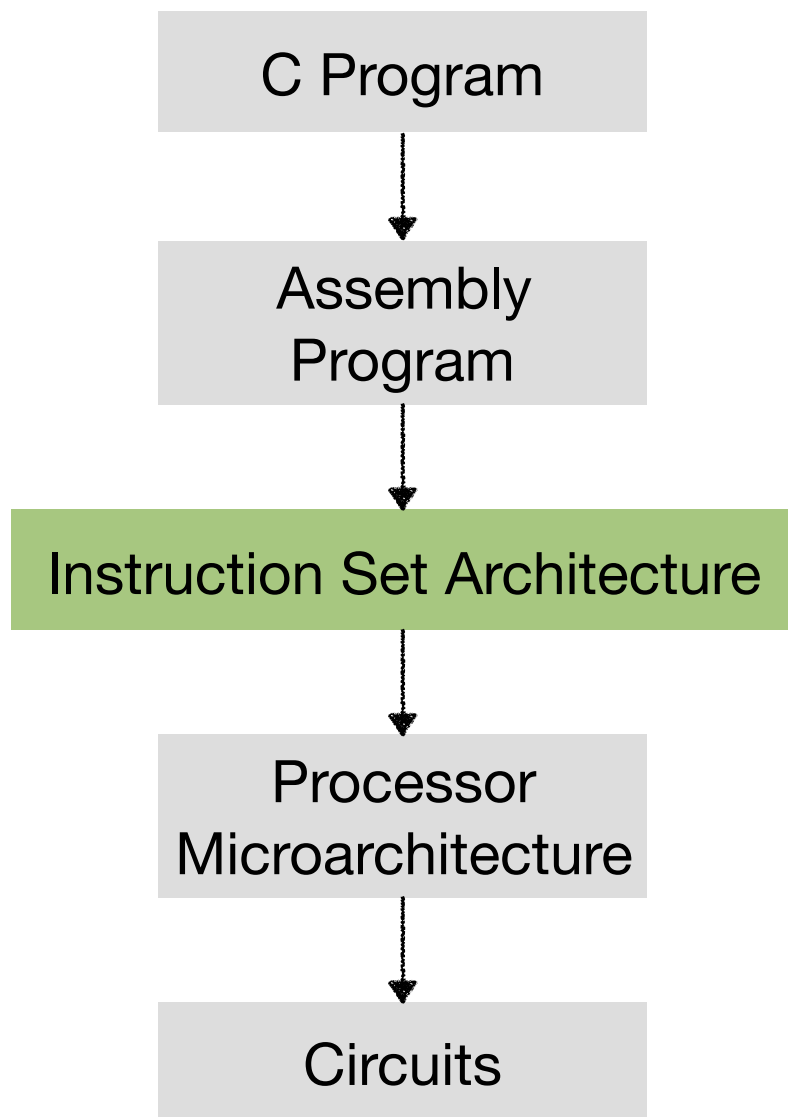
So far in 252...



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

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

So far in 252...

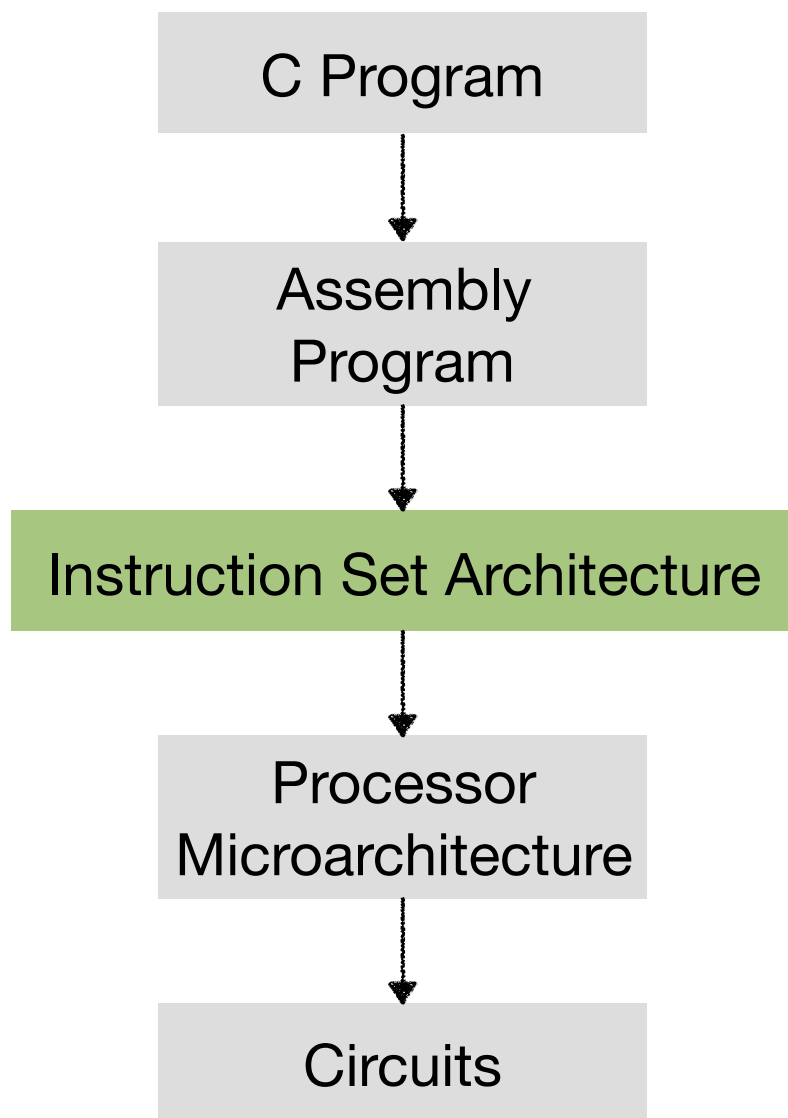


```
int, float  
if, else  
+, -, >>
```

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

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

So far in 252...



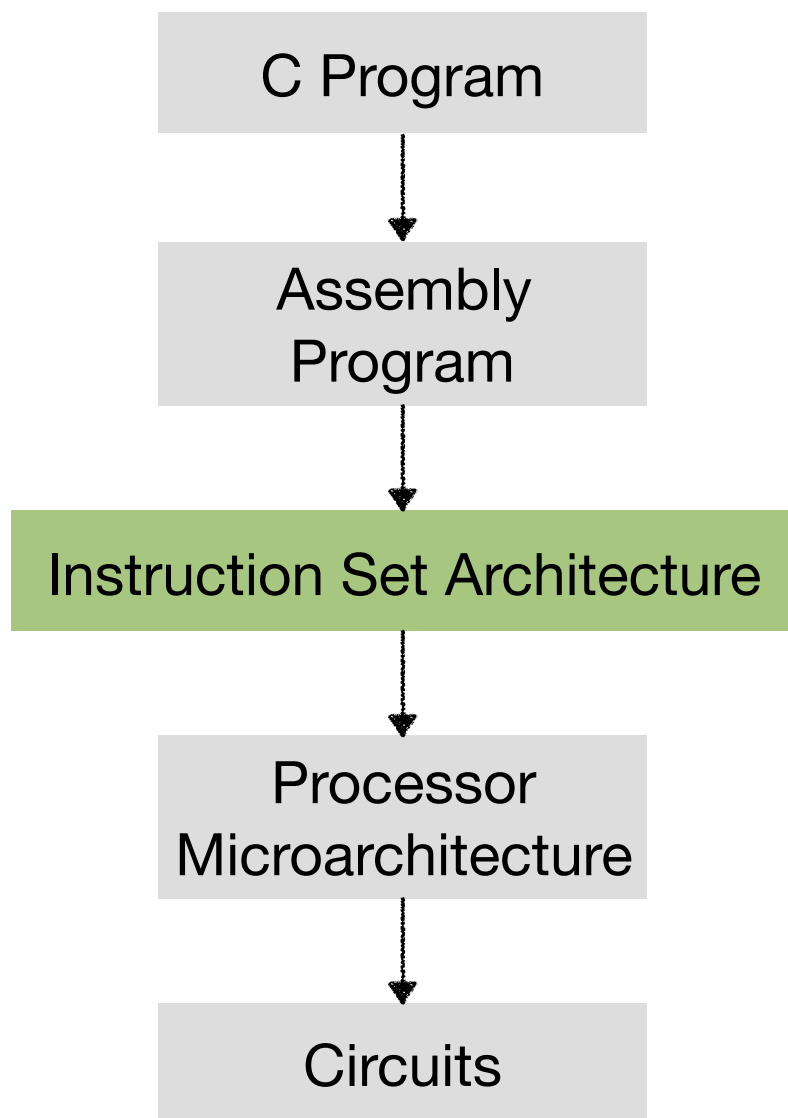
```
int, float  
if, else  
+, -, >>
```

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

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

Logic gates

So far in 252...



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

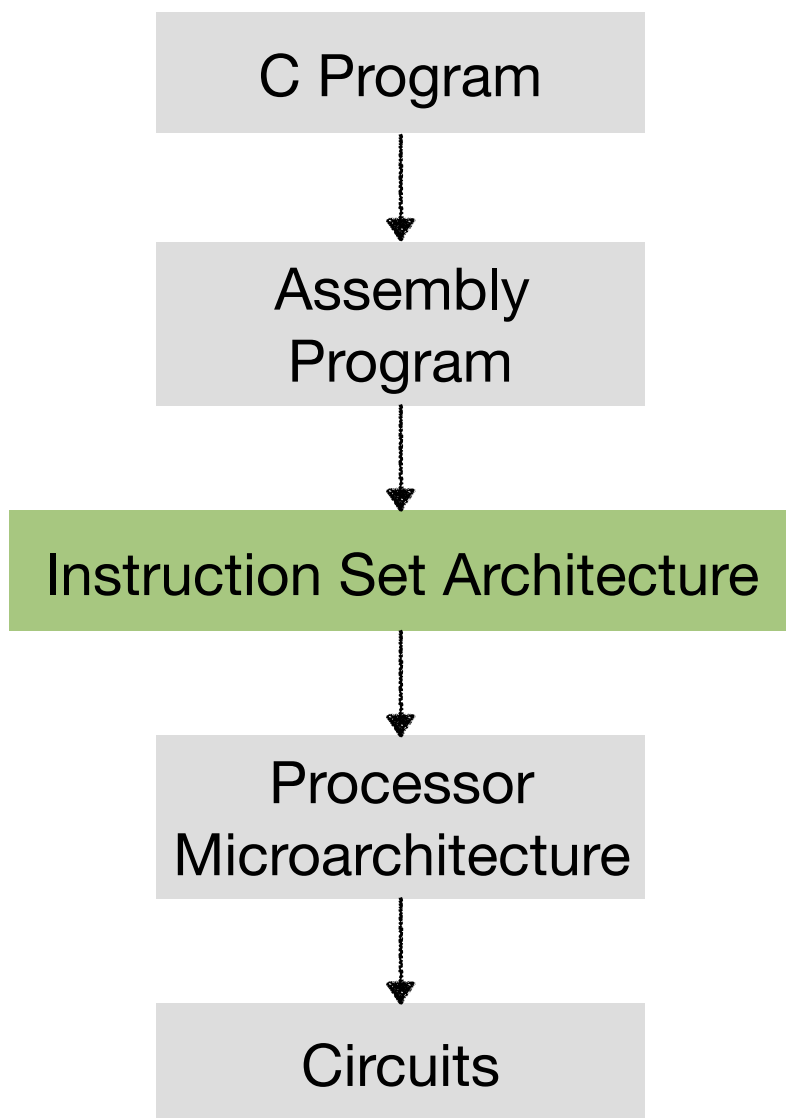
```
movq    %rsi, %rax  
imulq   %rdx, %rax  
jmp     .done
```

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

Logic gates

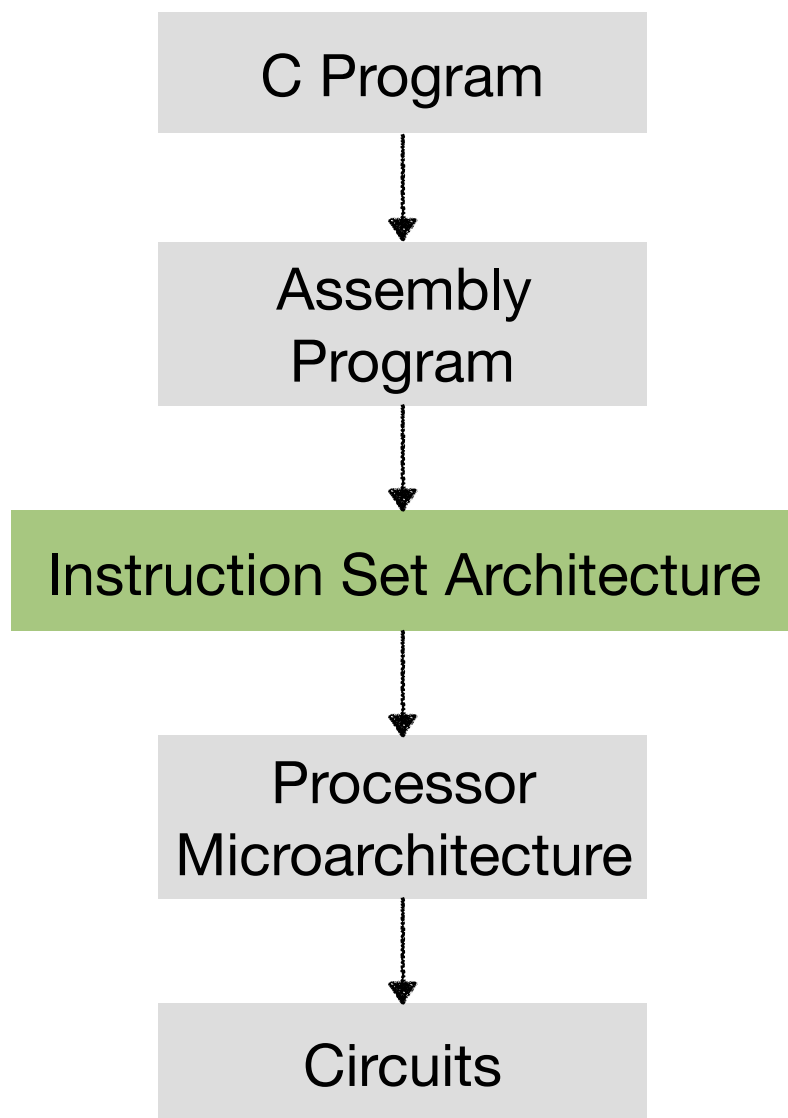
Transistors

So far in 252...



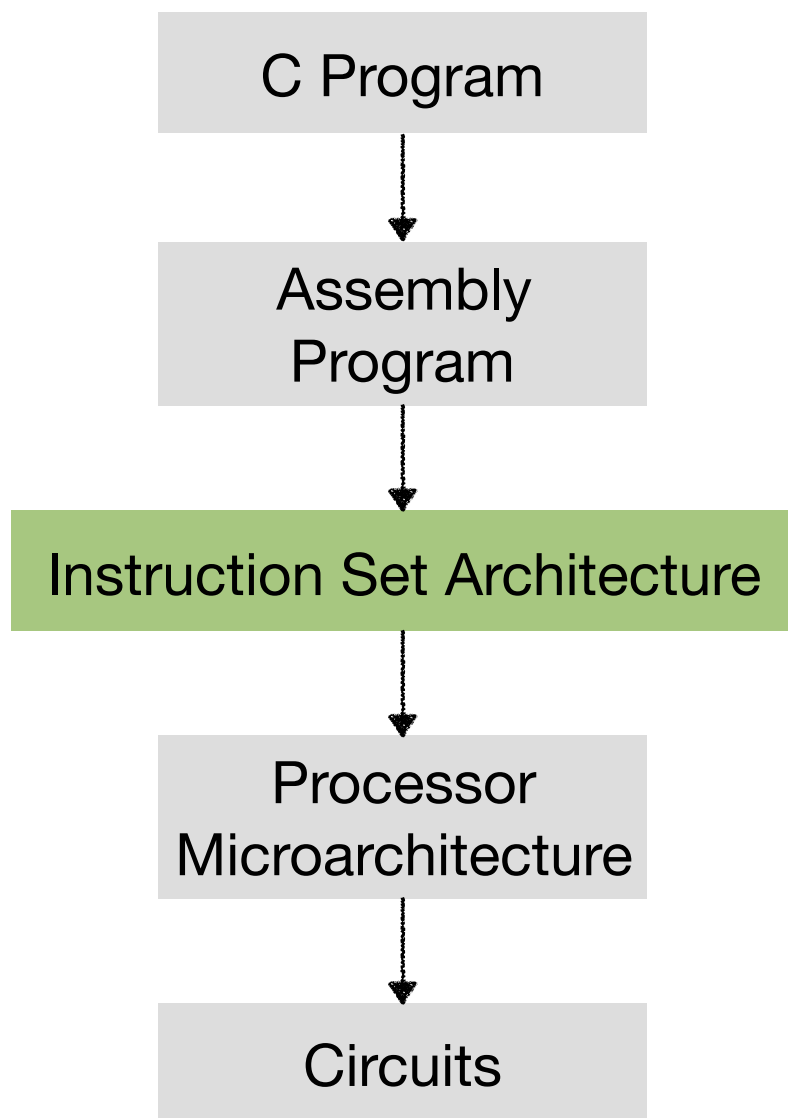
- ISA is the interface between assembly programs and microarchitecture

So far in 252...



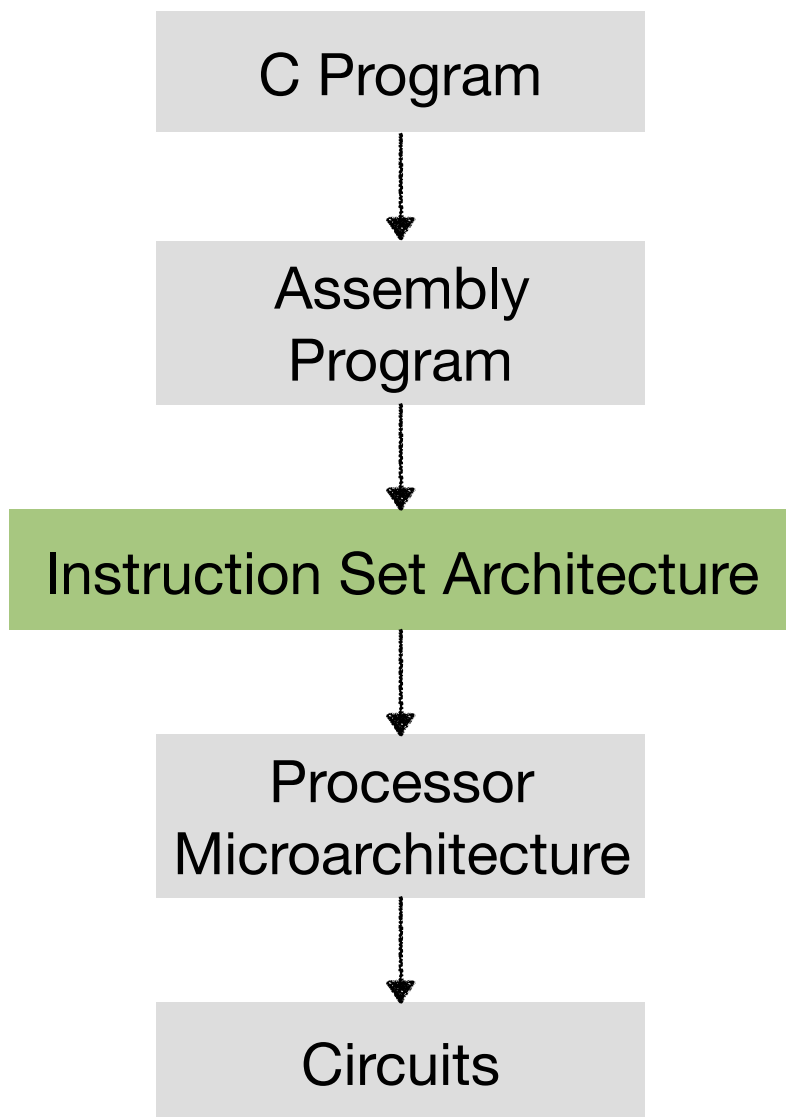
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:

So far in 252...



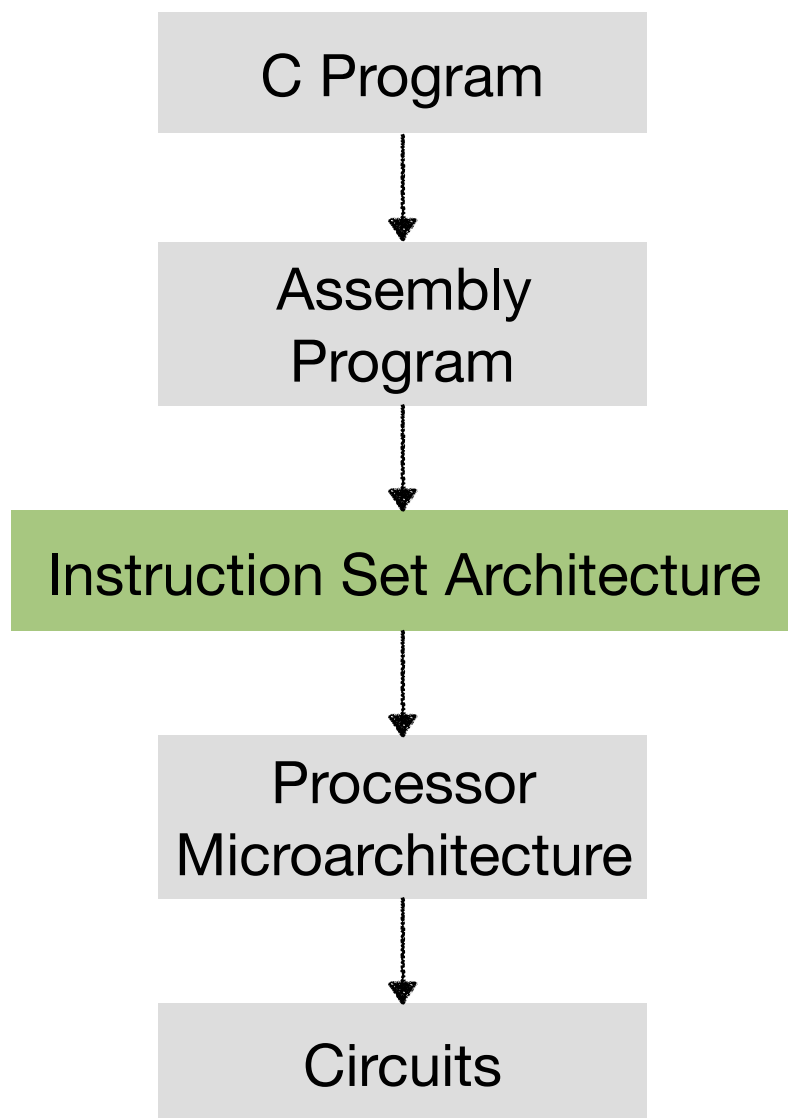
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
 - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?

So far in 252...



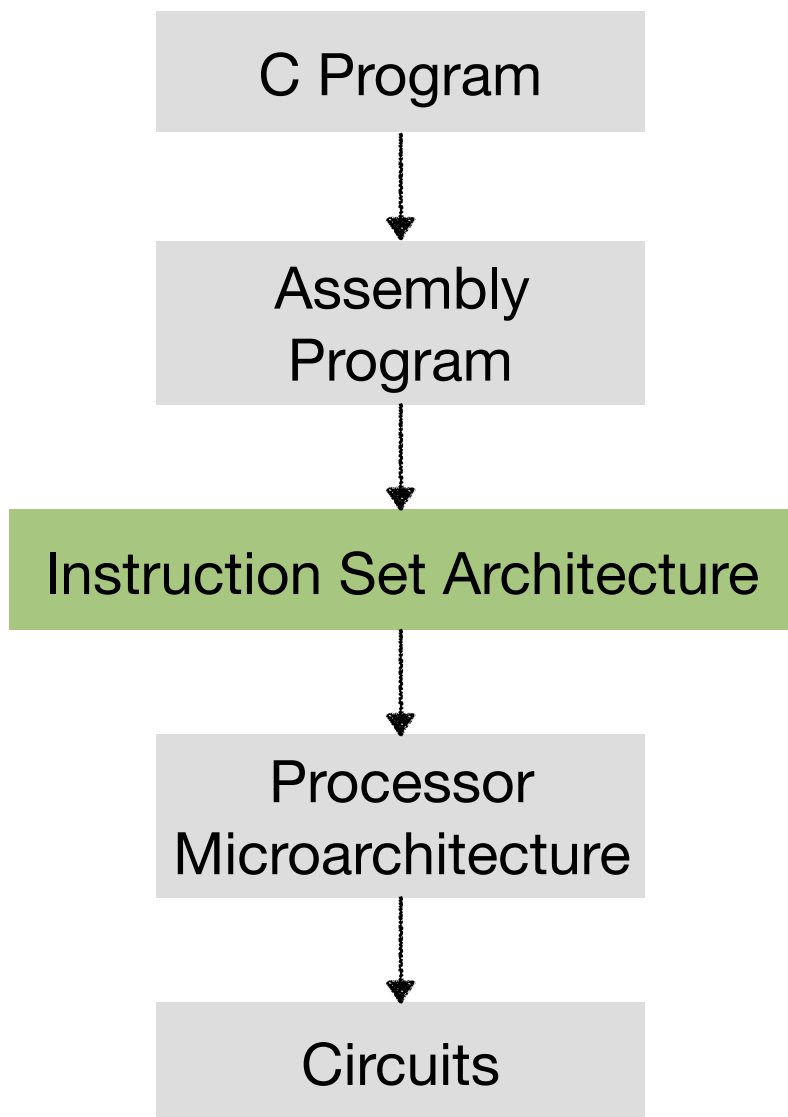
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
 - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
 - Instructions are executed sequentially.

So far in 252...



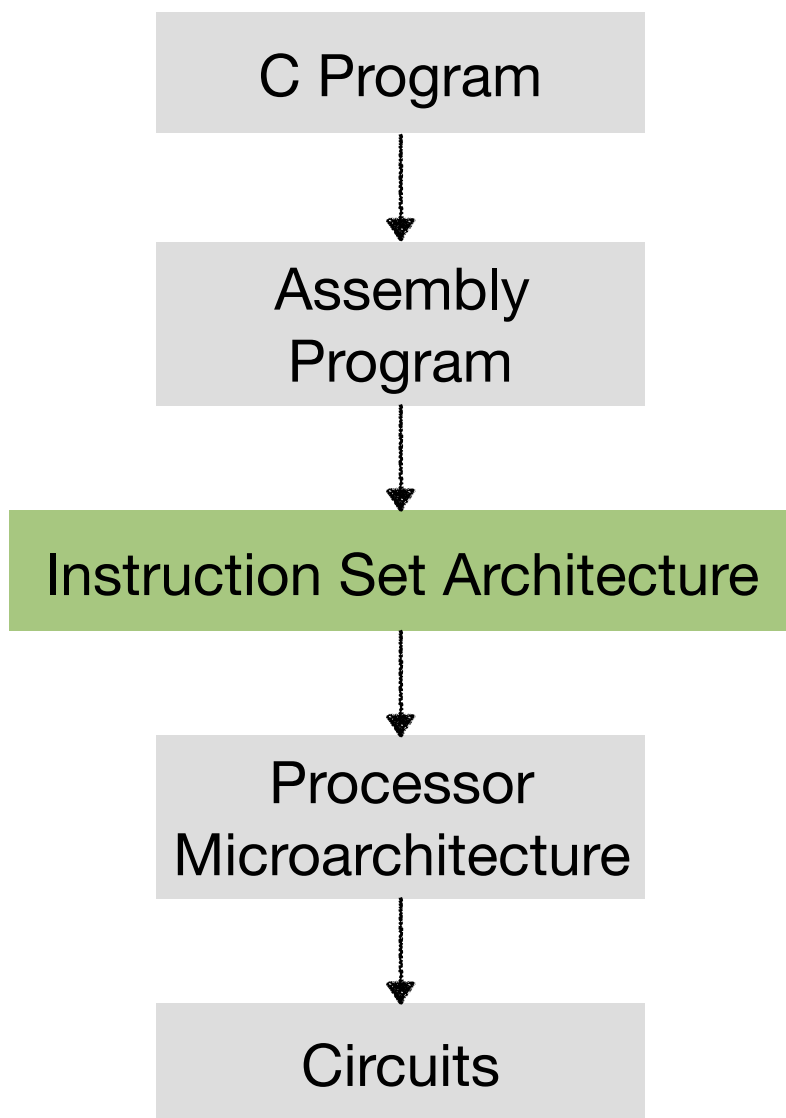
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
 - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
 - Instructions are executed sequentially.
- Microarchitecture view:

So far in 252...



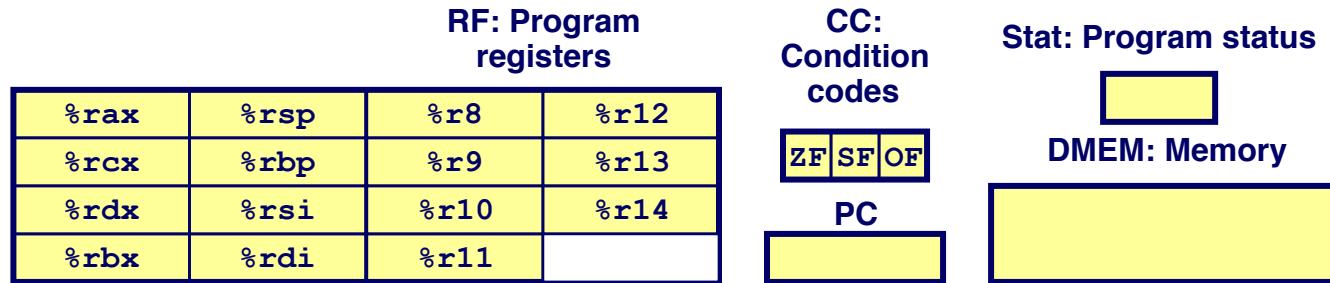
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
 - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
 - Instructions are executed sequentially.
- Microarchitecture view:
 - What hardware needs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

So far in 252...



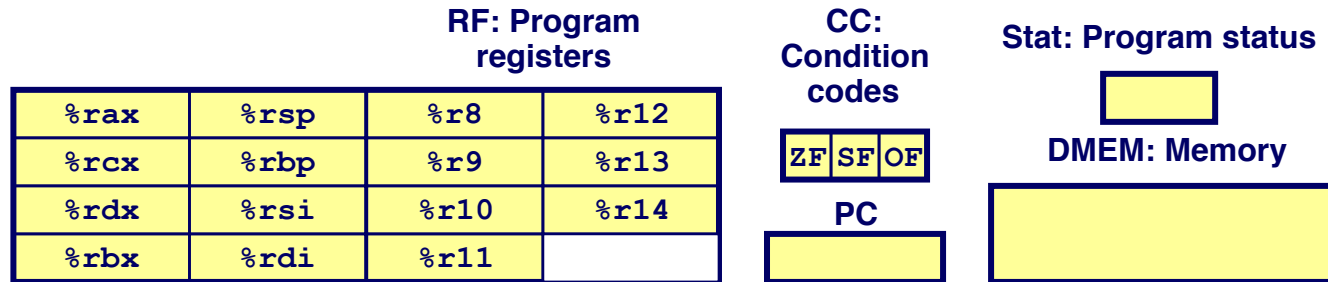
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
 - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
 - Instructions are executed sequentially.
- Microarchitecture view:
 - What hardware needs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

(Simplified) x86 Processor State



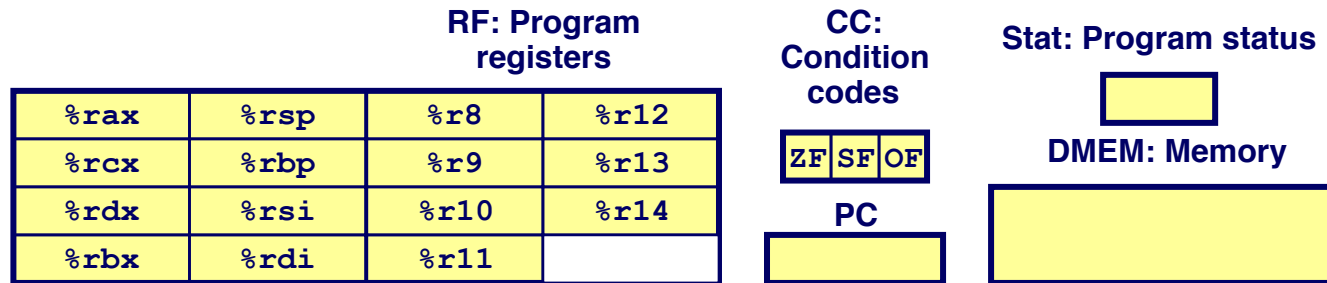
- Processor state is what's visible to assembly programs. Also known as architecture state.

(Simplified) x86 Processor State



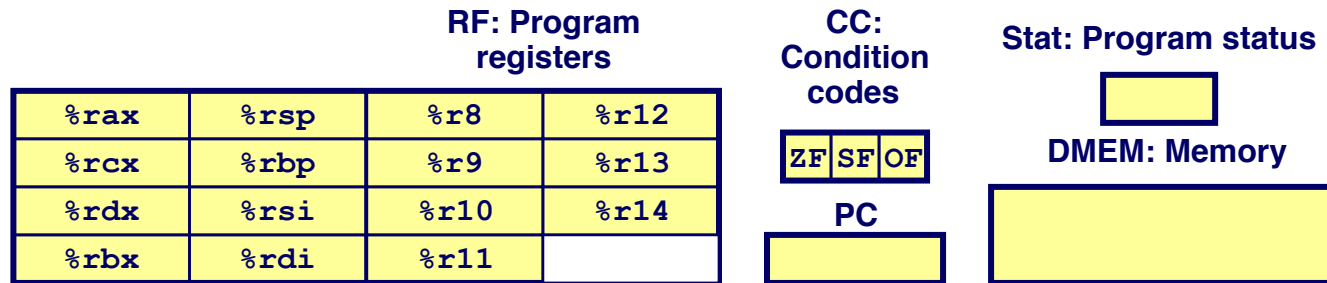
- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 15 registers (omit %r15). Each 64 bits

(Simplified) x86 Processor State



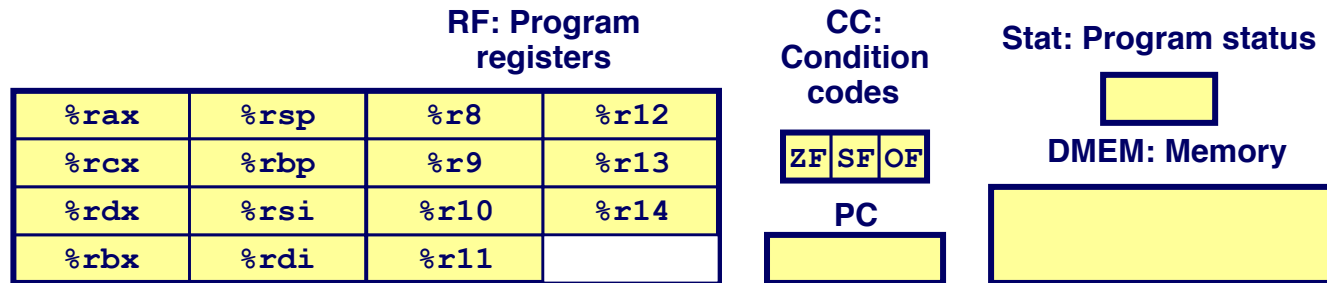
- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 15 registers (omit %r15). Each 64 bits
- Condition Codes: Single-bit flags set by arithmetic or logical instructions (ZF, SF, OF)

(Simplified) x86 Processor State



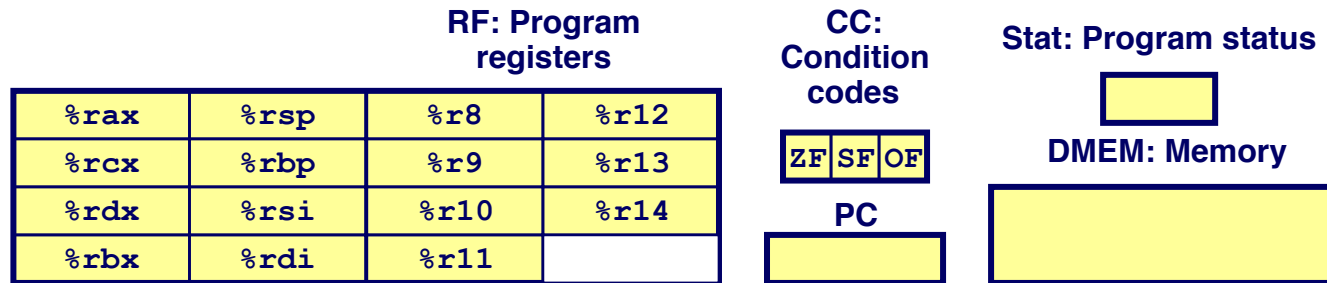
- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 15 registers (omit %r15). Each 64 bits
- Condition Codes: Single-bit flags set by arithmetic or logical instructions (ZF, SF, OF)
- Program Counter: Indicates address of next instruction

(Simplified) x86 Processor State



- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 15 registers (omit %r15). Each 64 bits
- 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: 15 registers (omit `%r15`). Each 64 bits
- 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

Characteristics of a Good ISA

- x86 is just one kind of ISA; there are many (ARM, MIPS, etc.)
- Must be unambiguous
- Must be expressive
 - Easily describes all the algorithms that will run on this platform
- Instructions are used
 - Very complex instructions might not be used often
- (Relatively) easy to compile
- (Relatively) easy to implement well
 - Has to be implementable
 - And, implementation provides good performance, cost, etc.
- ISAs often highly reliant on microarchitecture and vice-versa
 - Some ISAs easy to implement on some microarchitectures
 - Some microarchitectures make some instructions easy to implement

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?
 - There are 1 instruction ISAs

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?
 - There are 1 instruction ISAs
 - `subjle a, b, c ;`

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?
 - There are 1 instruction ISAs
 - `subjle a, b, c ;`
 - $\text{Mem}[b] = \text{Mem}[b] - \text{Mem}[a]$; if $(\text{Mem}[b] \leq 0)$ goto *c*

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?
 - There are 1 instruction ISAs
 - `subjle a, b, c ;`
 - $\text{Mem}[b] = \text{Mem}[b] - \text{Mem}[a]$; if $(\text{Mem}[b] \leq 0)$ goto *c*
- Number of registers per instruction

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?
 - There are 1 instruction ISAs
 - `subjle a, b, c ;`
 - $\text{Mem}[b] = \text{Mem}[b] - \text{Mem}[a]$; if $(\text{Mem}[b] \leq 0)$ goto c
- Number of registers per instruction
 - Affect number of bits per instruction

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?
 - There are 1 instruction ISAs
 - `subjle a, b, c ;`
 - $\text{Mem}[b] = \text{Mem}[b] - \text{Mem}[a]$; if $(\text{Mem}[b] \leq 0)$ goto `c`
- Number of registers per instruction
 - Affect number of bits per instruction
 - Affect number of registers the microarchitecture has to implement

Some ISA Design Tradeoffs

- Fewer instructions
 - Pros?
 - Cons?
 - There are 1 instruction ISAs
 - `subjle a, b, c ;`
 - $\text{Mem}[b] = \text{Mem}[b] - \text{Mem}[a]$; if $(\text{Mem}[b] \leq 0)$ goto c
- Number of registers per instruction
 - Affect number of bits per instruction
 - Affect number of registers the microarchitecture has to implement
 - How many?? Zero, One, Two, Three, Four, ...

Number of Registers Per Instruction

- To implement $C = A + B$, how many registers should an ISA provide?

Number of Registers Per Instruction

- To implement $C = A + B$, how many registers should an ISA provide?
- Zero
 - Stack machine (HP calculators): implied addresses
 - PUSH AddrA; PUSH AddrB; ADD; POP AddrC

Number of Registers Per Instruction

- To implement $C = A + B$, how many registers should an ISA provide?
- Zero
 - Stack machine (HP calculators): implied addresses
 - PUSH AddrA; PUSH AddrB; ADD; POP AddrC
- One (implied)
 - Accumulator-based machine
 - LOAD AddrA; ADD AddrB; STORE AddrC

Number of Registers Per Instruction

- To implement $C = A + B$, how many registers should an ISA provide?
- Zero
 - Stack machine (HP calculators): implied addresses
 - PUSH AddrA; PUSH AddrB; ADD; POP AddrC
- One (implied)
 - Accumulator-based machine
 - LOAD AddrA; ADD AddrB; STORE AddrC
- Two (same register, src and dest), e.g., x86
 - One source is destination
 - LOAD R1, AddrA; LOAD R2, AddrB;
 - ADD R1, R2; STORE R1, AddrC

Number of Registers Per Instruction

- To implement $C = A + B$, how many registers should an ISA provide?
- Zero
 - Stack machine (HP calculators): implied addresses
 - PUSH AddrA; PUSH AddrB; ADD; POP AddrC
- One (implied)
 - Accumulator-based machine
 - LOAD AddrA; ADD AddrB; STORE AddrC
- Two (same register, src and dest), e.g., x86
 - One source is destination
 - LOAD R1, AddrA; LOAD R2, AddrB;
 - ADD R1, R2; STORE R1, AddrC
- Three
 - Current ($D = S1 \text{ OP } S2$)
 - LOAD R1, AddrA; LOAD R2, AddrB;
 - ADD R3, R1, R2; STORE R3, AddrC

Number of Registers Per Instruction

- To implement $C = A + B$, how many registers should an ISA provide?
- Zero
 - Stack machine (HP calculators): implied addresses
 - PUSH AddrA; PUSH AddrB; ADD; POP AddrC
- One (implied)
 - Accumulator-based machine
 - LOAD AddrA; ADD AddrB; STORE AddrC
- Two (same register, src and dest), e.g., x86
 - One source is destination
 - LOAD R1, AddrA; LOAD R2, AddrB;
 - ADD R1, R2; STORE R1, AddrC
- Three
 - Current ($D = S1 \text{ OP } S2$)
 - LOAD R1, AddrA; LOAD R2, AddrB;
 - ADD R3, R1, R2; STORE R3, AddrC
- Four and above

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 in a stored program computer, 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

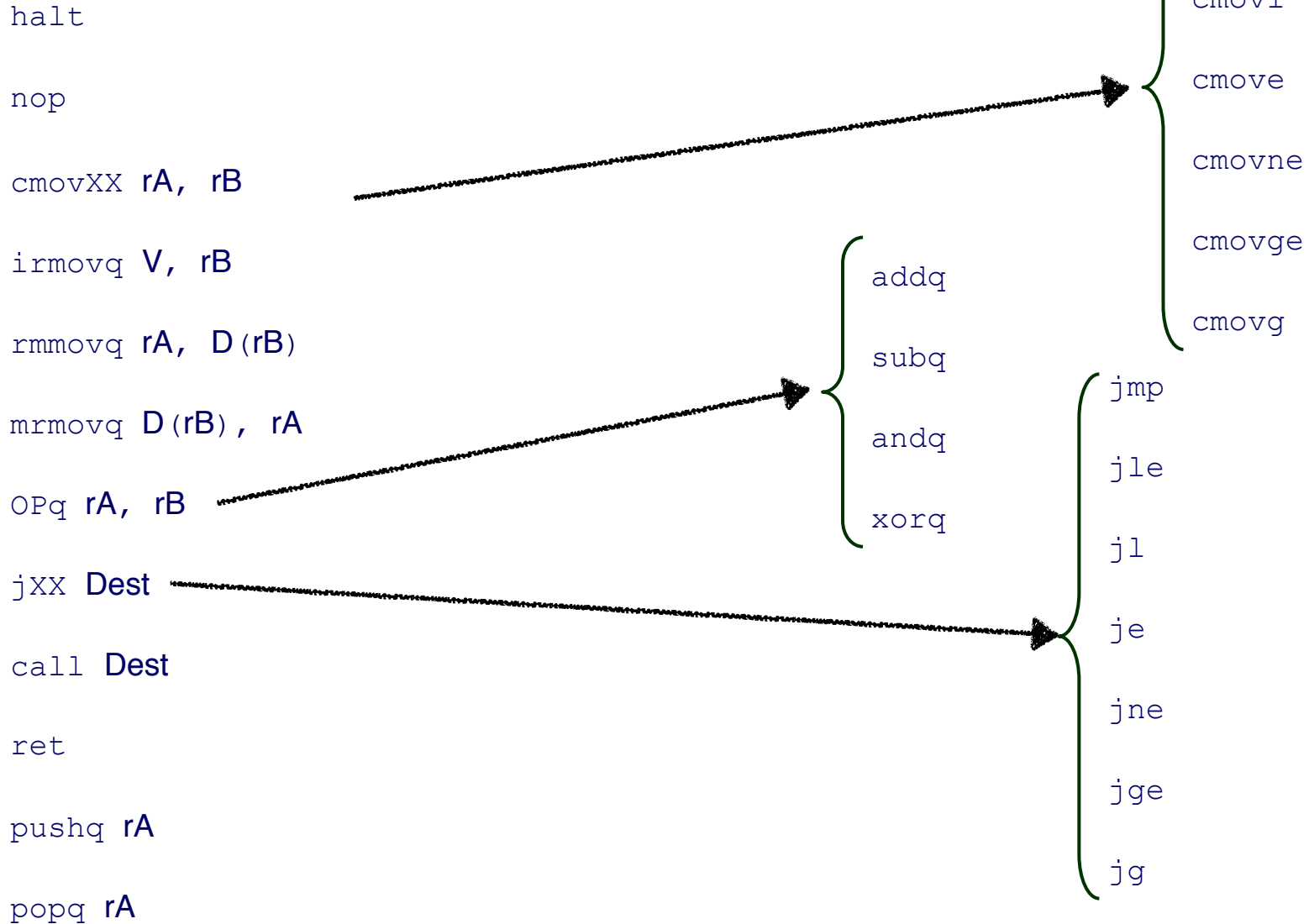
je

jne

jge

jg

Y86-64 Instructions



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

rmmovq

cmovle

cmovl

cmovbe

cmovne

cmovge

cmovg

addq

subq

andq

xorq

jmp

jle

jl

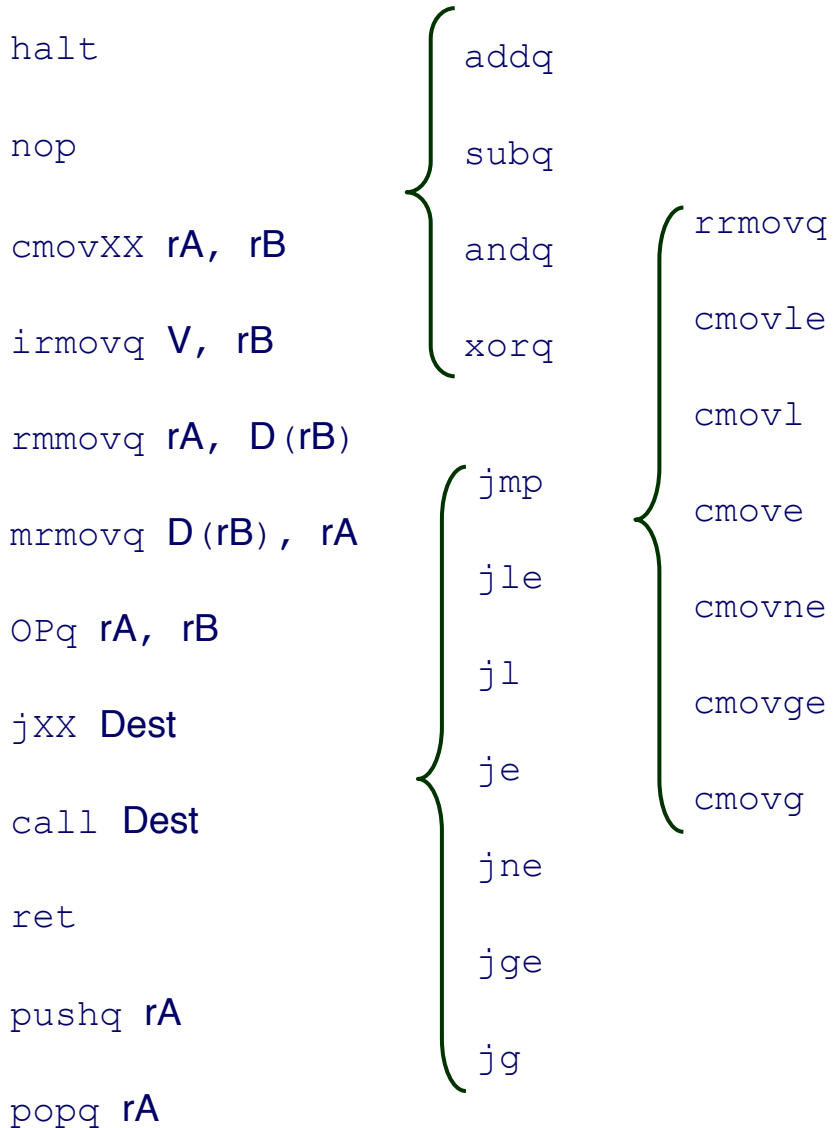
je

jne

jge

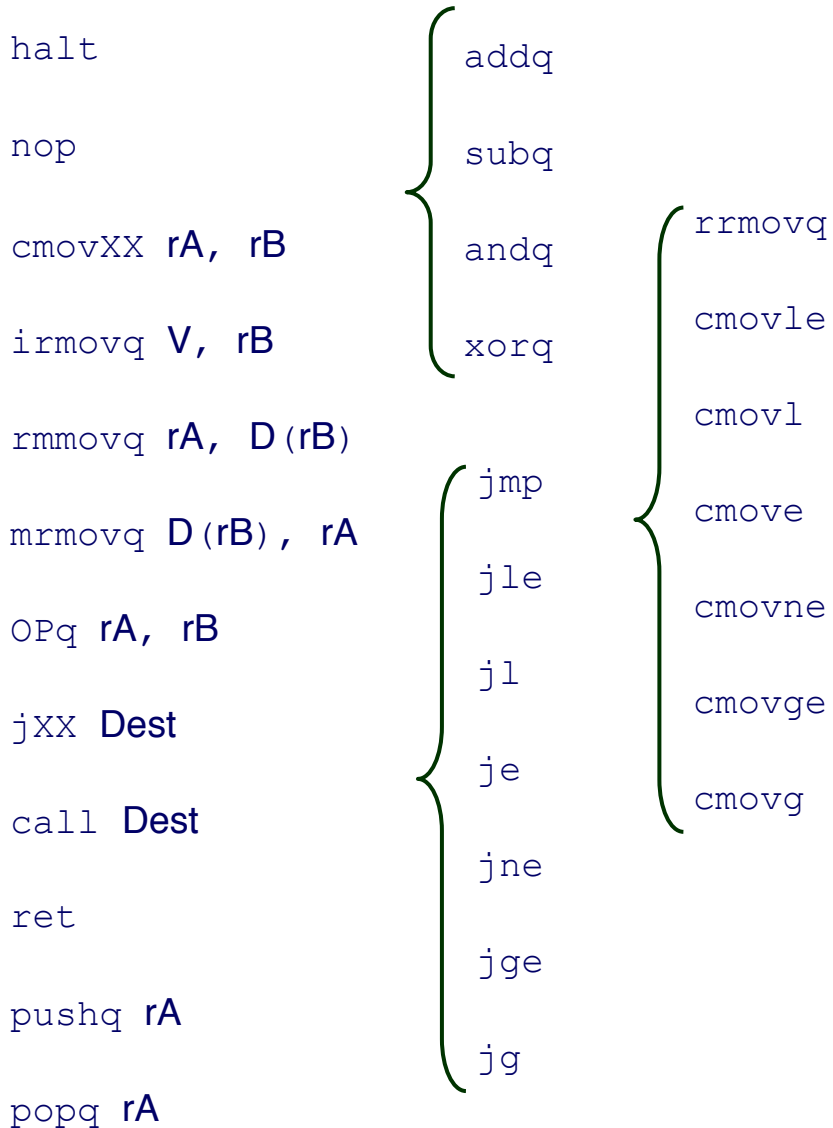
jg

Encoding Operands



- 27 Instructions, so need 5 bits for encoding the operand

Encoding Operands



- 27 Instructions, so need 5 bits for encoding the operand
- Or: group similar instructions, use one opcode for them, and then use one more bit 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 one more bit 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 one more bit 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 one more bit 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”

<code>%rax</code>	0	<code>%r8</code>	8
<code>%rcx</code>	1	<code>%r9</code>	9
<code>%rdx</code>	2	<code>%r10</code>	A
<code>%rbx</code>	3	<code>%r11</code>	B
<code>%rsp</code>	4	<code>%r12</code>	C
<code>%rbp</code>	5	<code>%r13</code>	D
<code>%rsi</code>	6	<code>%r14</code>	E
<code>%rdi</code>	7	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



- 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



- 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

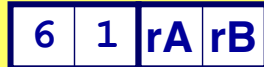
Add

`addq rA, rB`



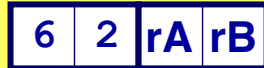
Subtract (rA from rB)

`subq rA, rB`



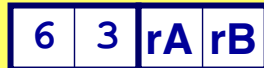
And

`andq rA, rB`



Exclusive-Or

`xorq 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

Arithmetic and Logical Operations

Function Code

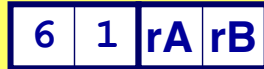
Add

`addq rA, rB`



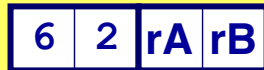
Subtract (rA from rB)

`subq rA, rB`



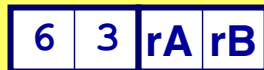
And

`andq rA, rB`



Exclusive-Or

`xorq 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

Arithmetic and Logical Operations

Instruction Code

Function Code

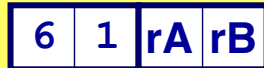
Add

`addq rA, rB`



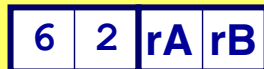
Subtract (rA from rB)

`subq rA, rB`



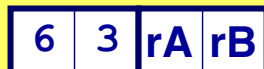
And

`andq rA, rB`



Exclusive-Or

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

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	<div style="border: 1px solid black; background-color: yellow; padding: 5px; display: inline-block;"> mrmovq -12(%rbp), %rcx </div>					
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						

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

```
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			jle .L4					
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						

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

```
jle .L4
```

jXX Dest
call Dest

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	call foo							
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	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.

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

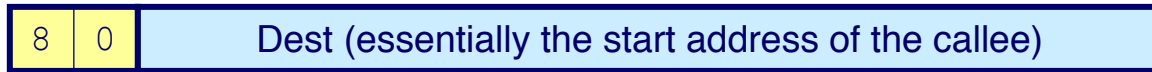
0 1 2 3 4 5 6 7 8 9

`jXX Dest`



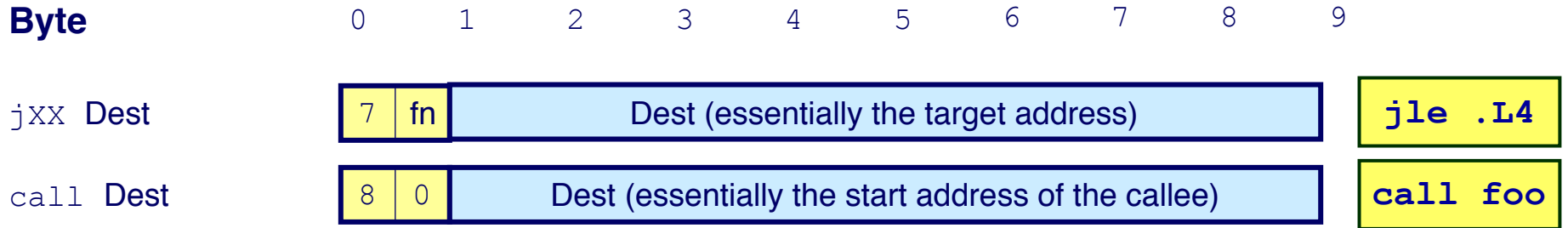
`jle .L4`

`call Dest`



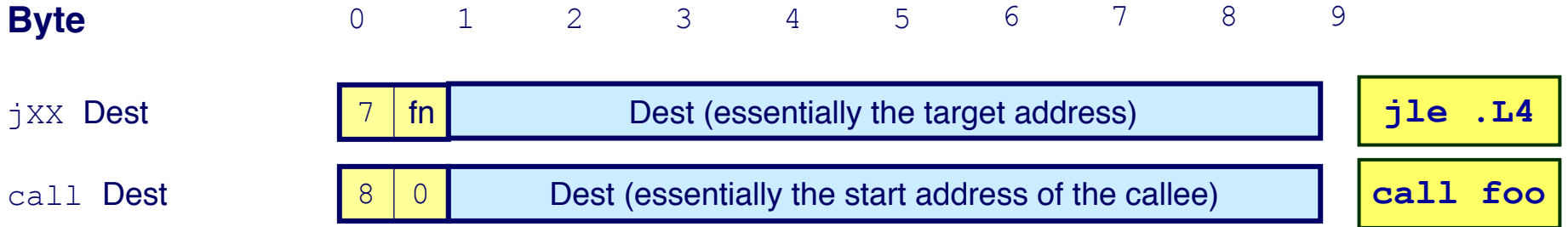
`call foo`

One More Complication...



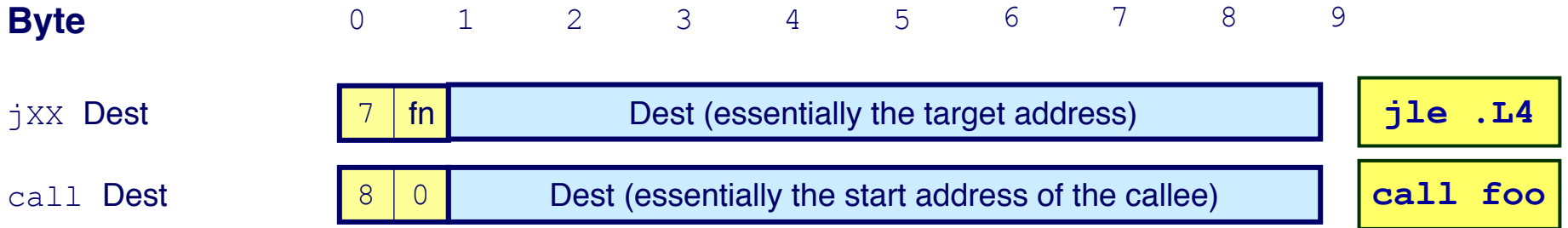
- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?

One More Complication...



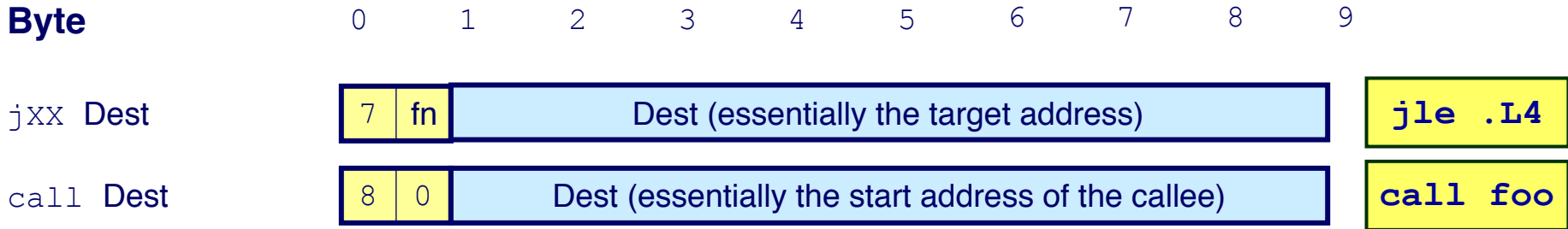
- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?
- One alternative: use a super long instruction encoding format.
 - Simple to encode, but space inefficient (waste bits for jumps to short addr.)

One More Complication...



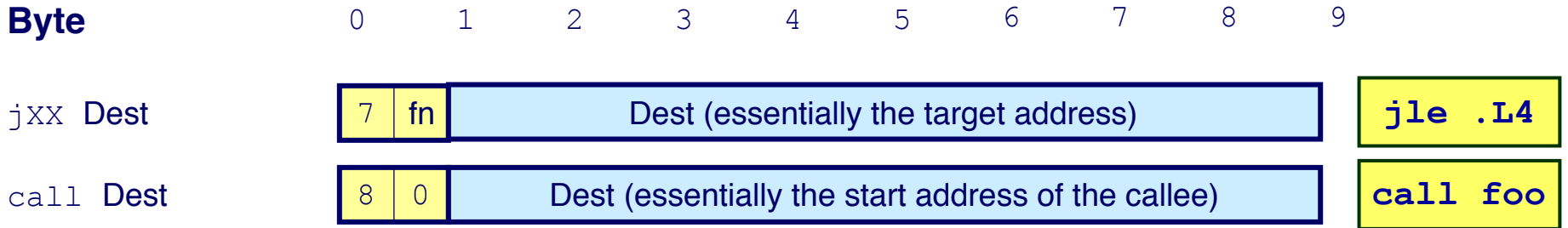
- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?
- One alternative: use a super long instruction encoding format.
 - Simple to encode, but space inefficient (waste bits for jumps to short addr.)
- Another alternative: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

One More Complication...



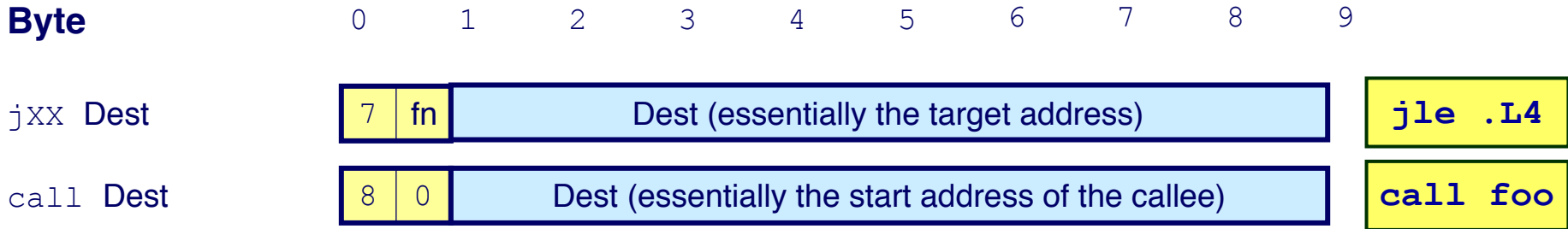
- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?
- One alternative: use a super long instruction encoding format.
 - Simple to encode, but space inefficient (waste bits for jumps to short addr.)
- Another alternative: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

One More Complication...



- 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: have to different encodings for jump/call, one with a short `Dest` field for short jumps and the other for long jumps.
- Or: encode the relative address, not the absolute address
 - E.g., encode `(.L4 - current address)` in `Dest`
- Better yet, combines the above two ideas: short relative jump and long relative jump. This is what x86 and many other ISAs do. Elegant design.

One More Complication...



- 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: have to different encodings for jump/call, one with a short `Dest` field for short jumps and the other for long jumps.
- Or: encode the relative address, not the absolute address
 - E.g., encode `(.L4 - current address)` in `Dest`
- Better yet, combines the above two ideas: short relative jump and long relative jump. This is what x86 and many other ISAs do. Elegant design.
- What if you want to jump really far away from the current instruction?
 - Alternatives: indirect jump, use a combination of absolute + relative addresses (“Far jumps” in x86).

Miscellaneous Instructions



- Don't do anything



- 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

How Does An Assembler Work?

- Translates assembly code to binary-encode
- Reads assembly program line by line, and translates according to the instruction format defined by an ISA

Add



- It sometimes needs to make two passes on the assembly program to resolve forward references
 - E.g., forward branch target address

Jump Unconditionally



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 (VLW) 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...

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
```

and their bit encodings.

Processor
Microarchitecture

Logic gates

Circuits

Transistors

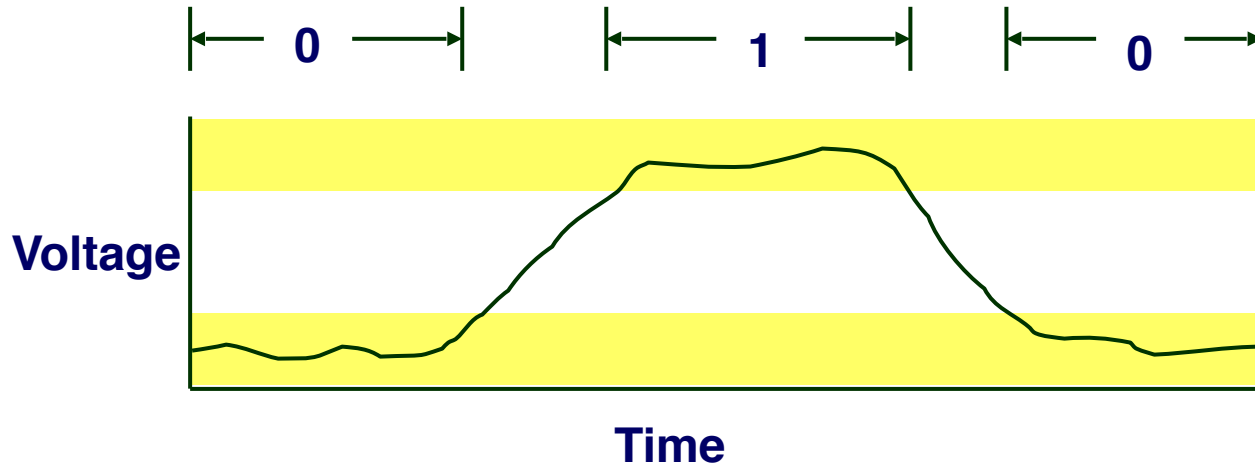
Today: Circuits Basics

- 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

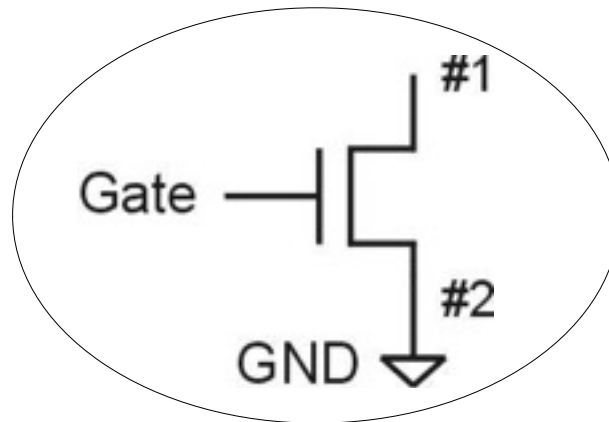
- two types: n-type and p-type

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).

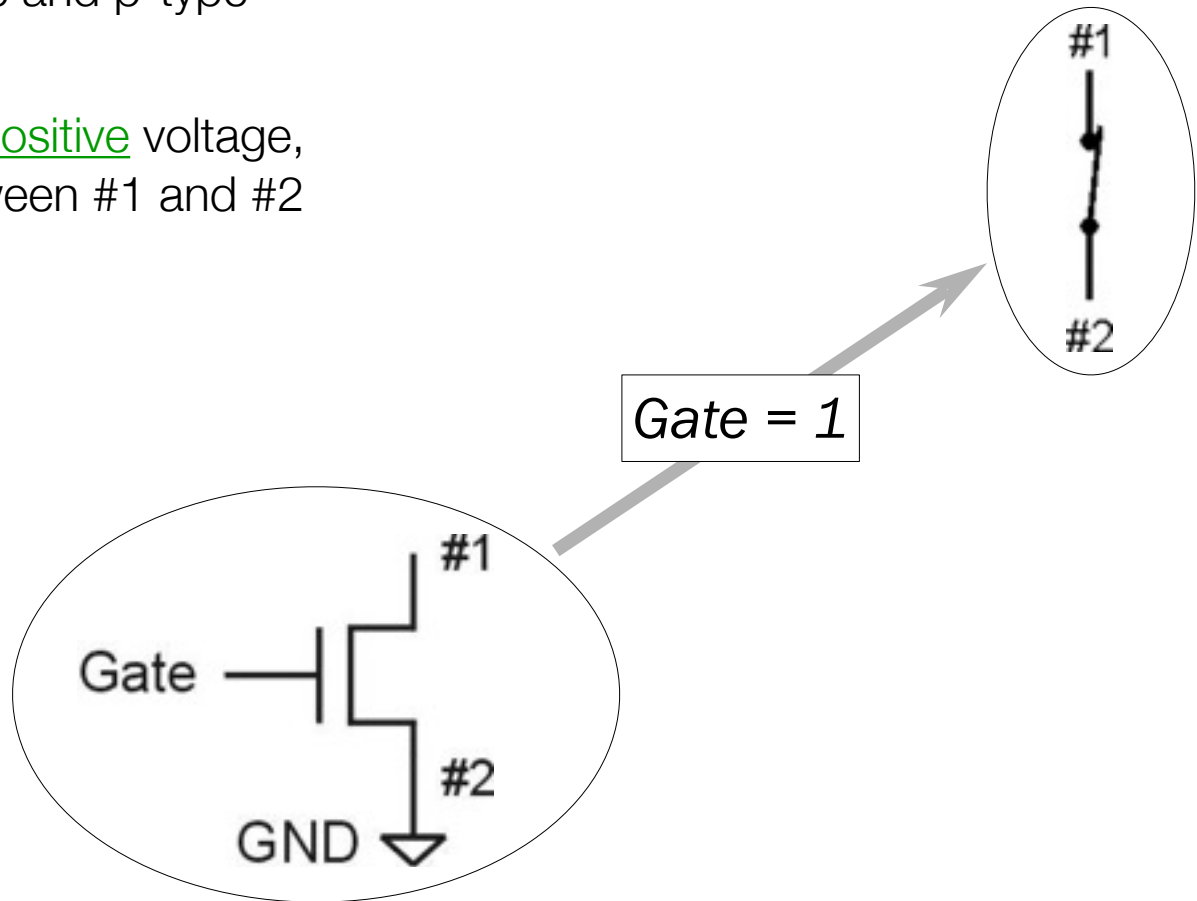
Basic Building Block: Transistors

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).

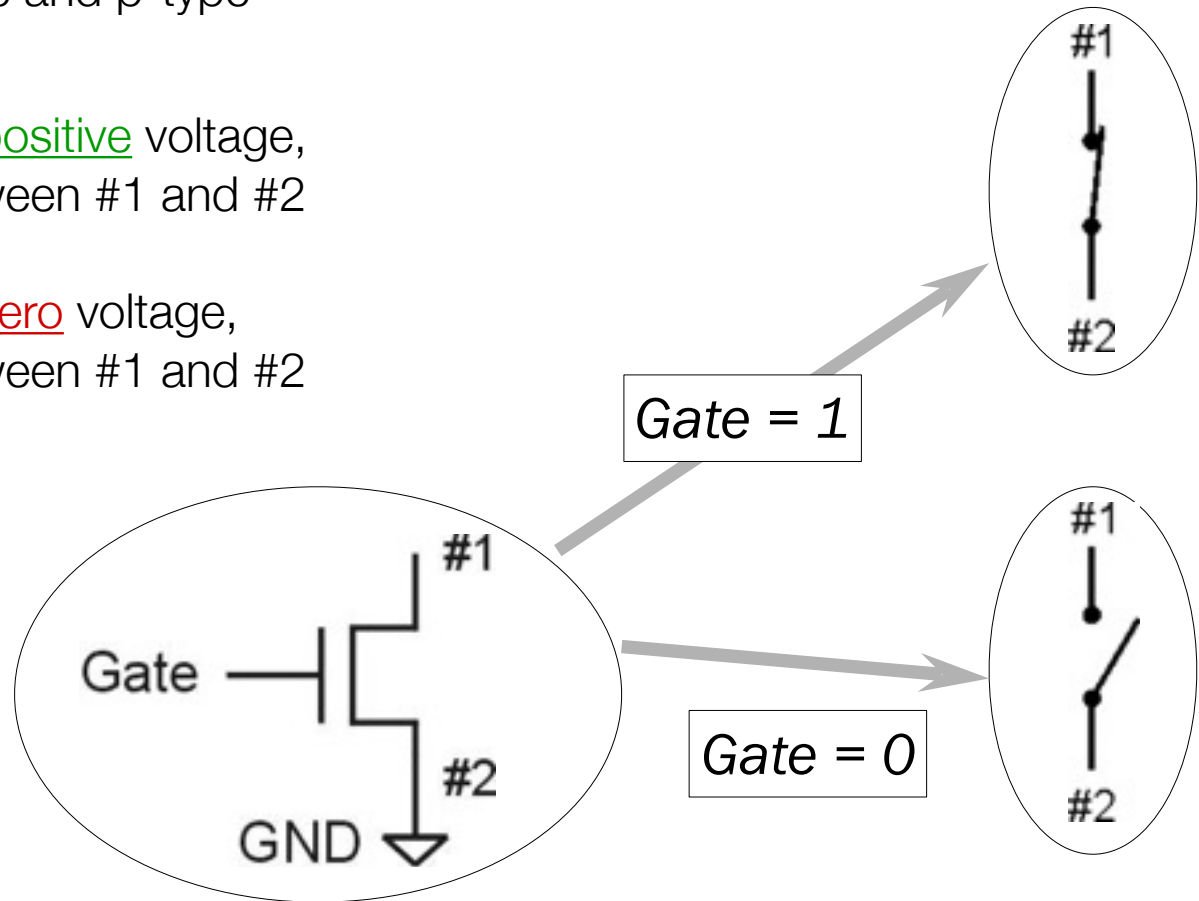
Basic Building Block: Transistors

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)

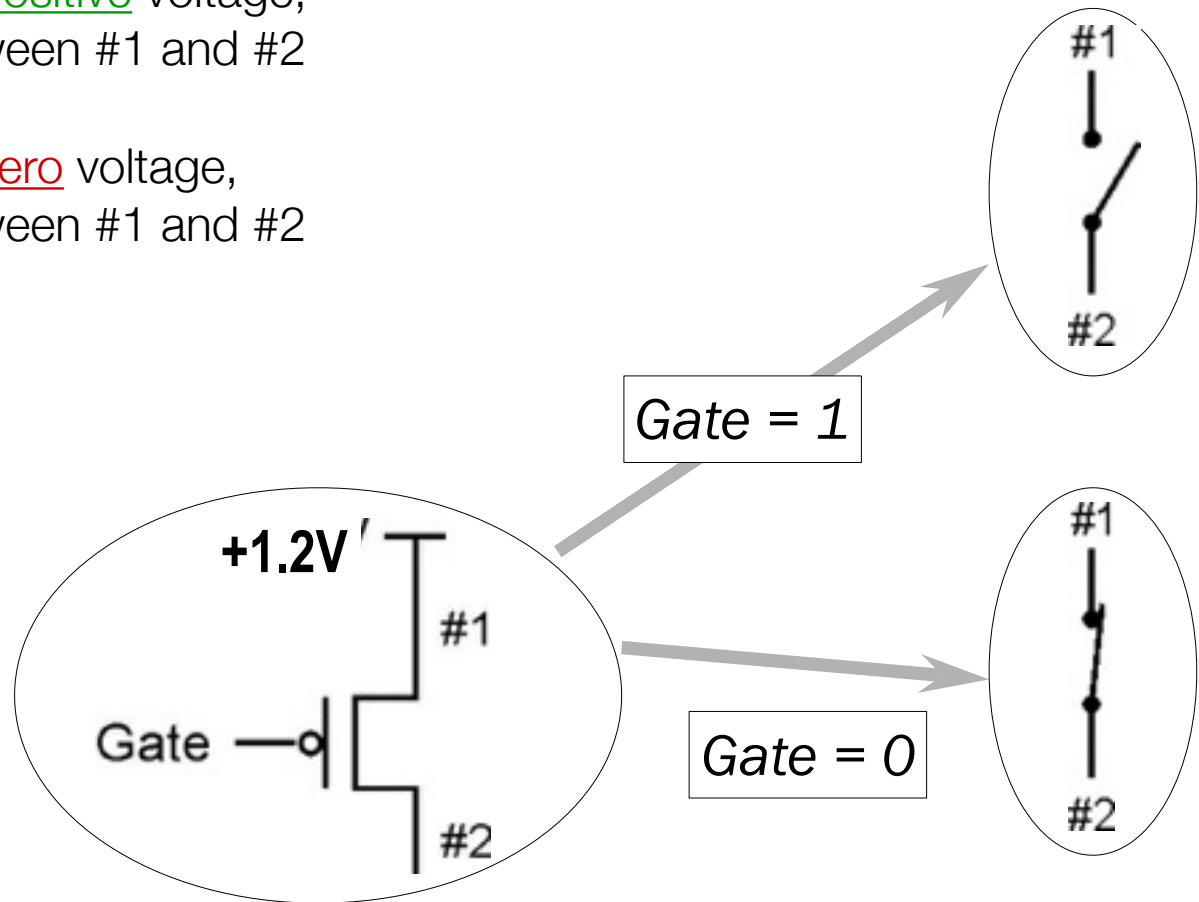


Terminal #2 must be connected to GND (0V).

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)

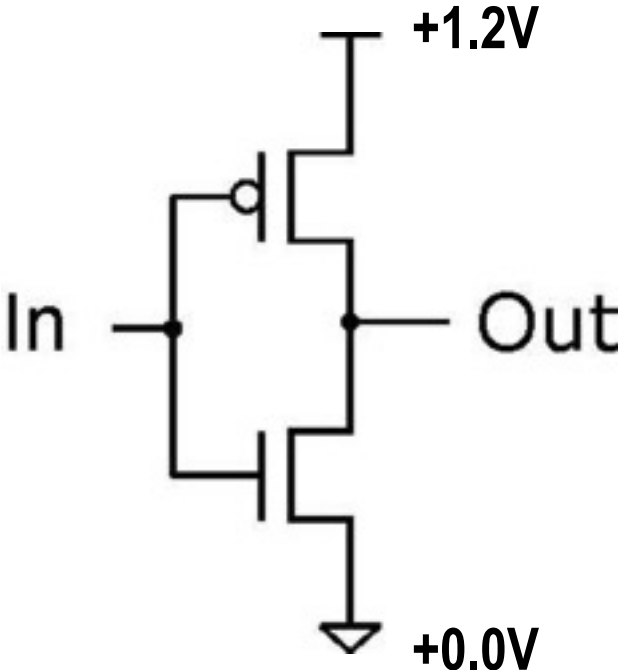


Terminal #1 must be connected to +1.2V

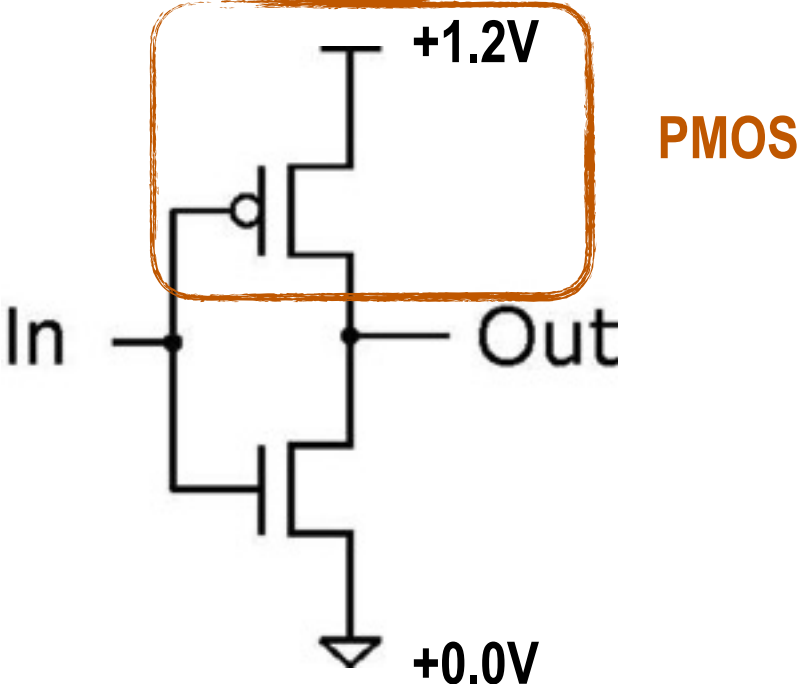
CMOS Circuit

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

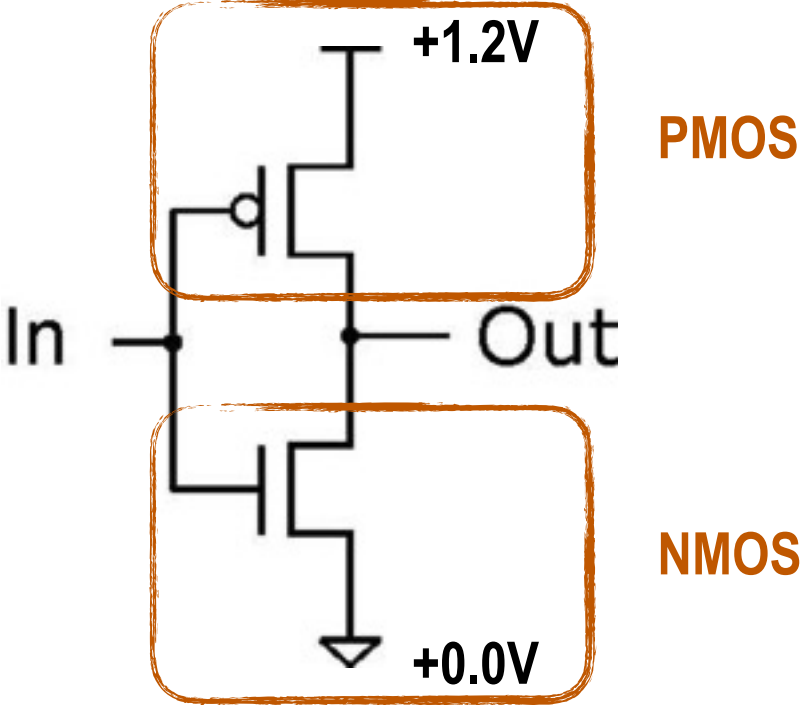
Inverter (NOT Gate)



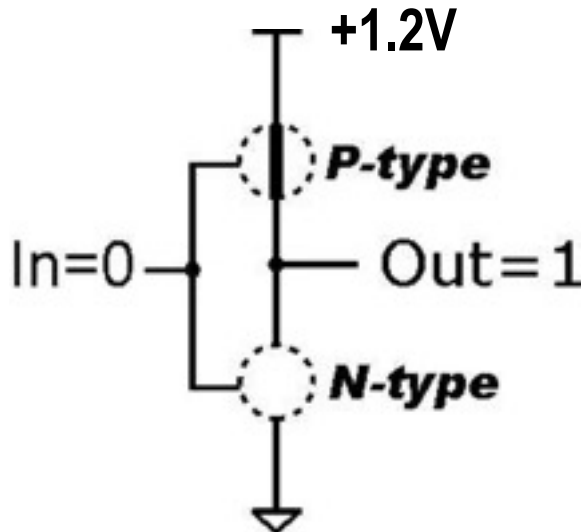
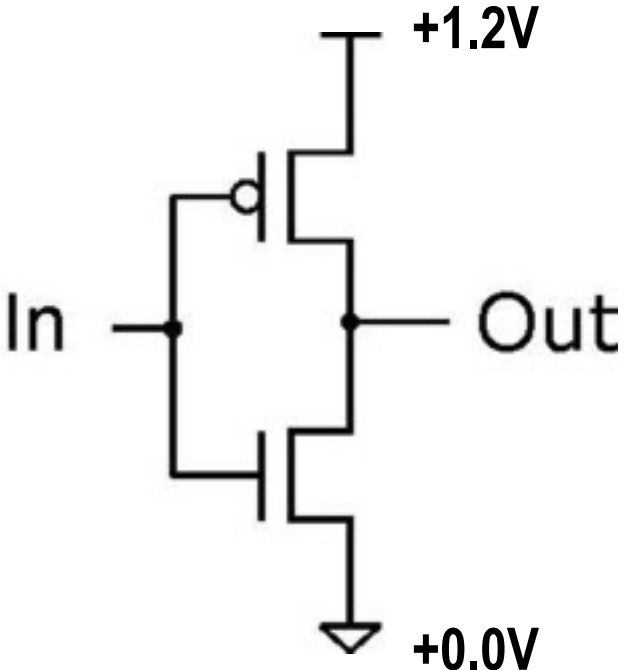
Inverter (NOT Gate)



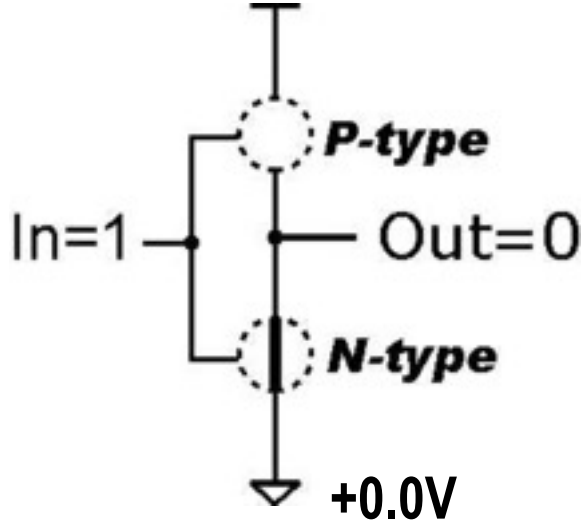
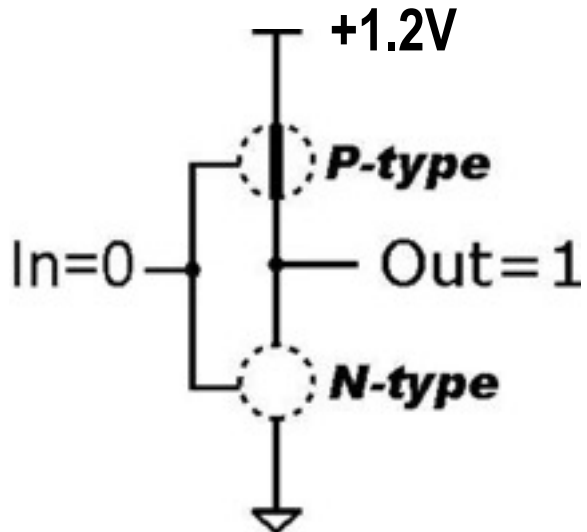
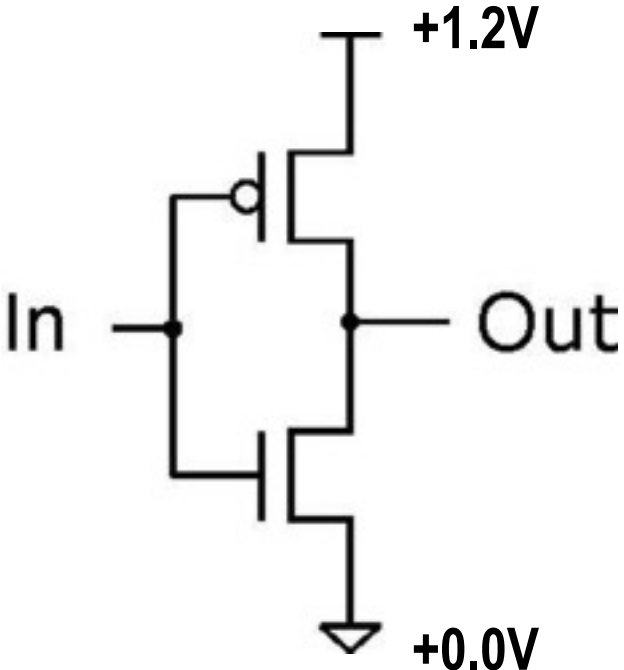
Inverter (NOT Gate)



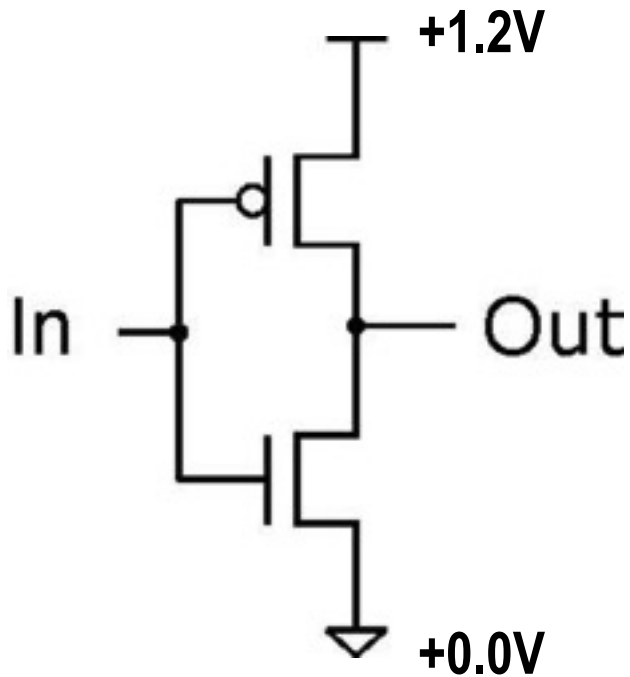
Inverter (NOT Gate)



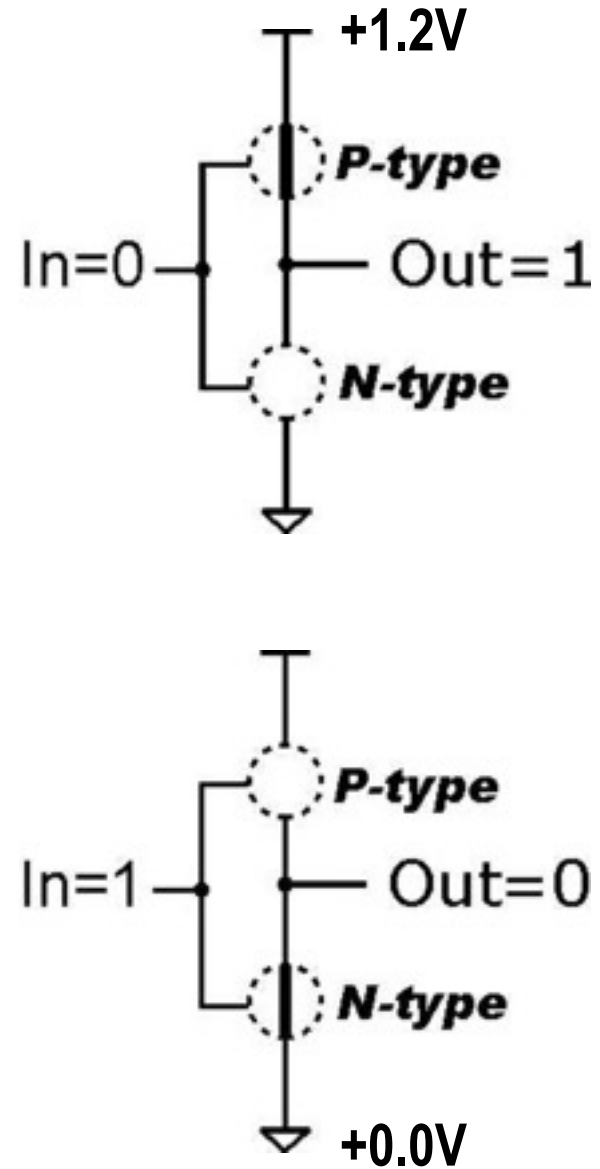
Inverter (NOT Gate)



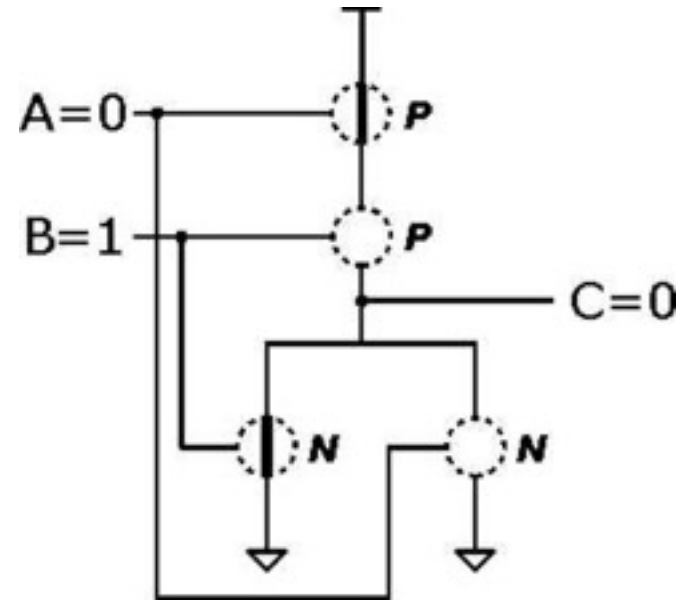
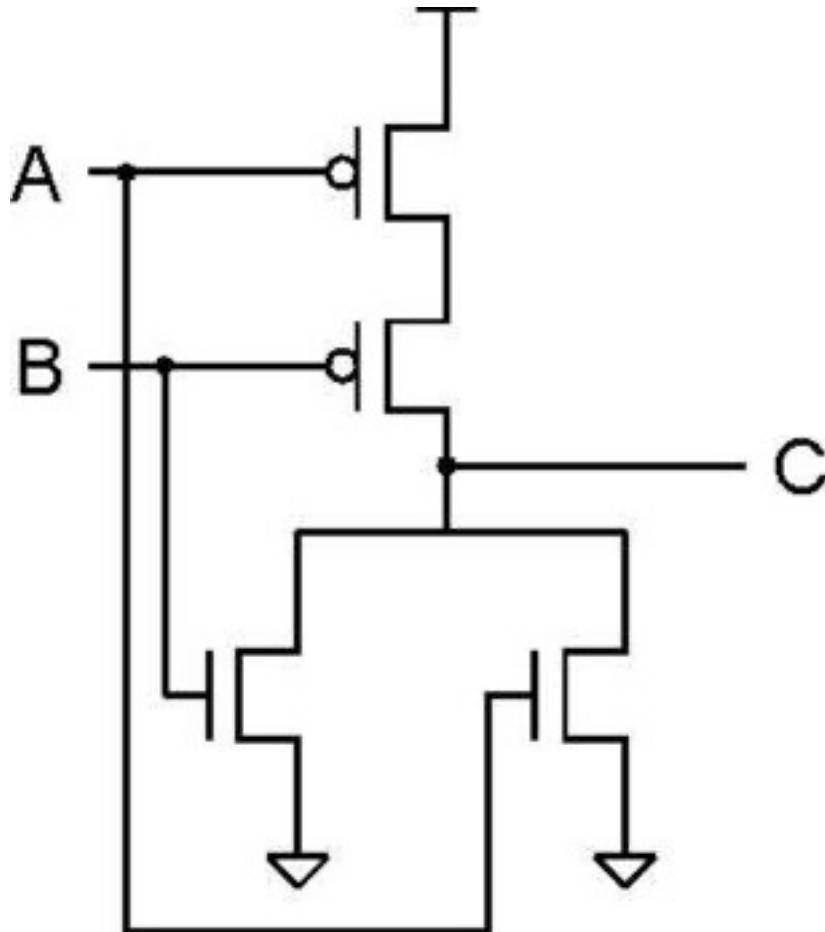
Inverter (NOT Gate)



In	Out
0	1
1	0



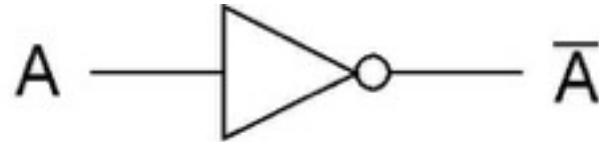
NOR Gate (NOT + OR)



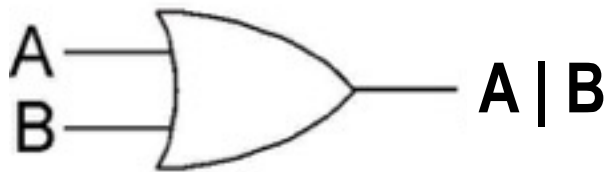
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

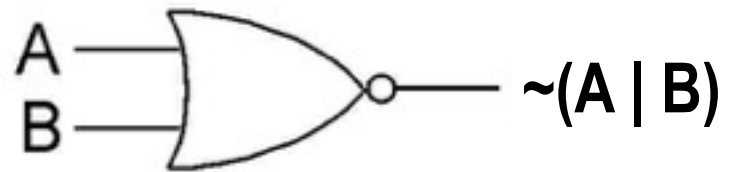
Basic Logic Gates



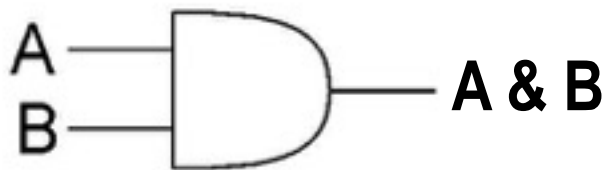
NOT



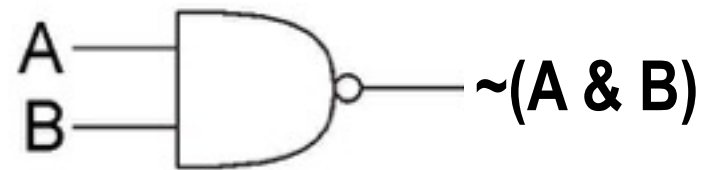
OR



NOR



AND



NAND