CSC 252: Computer Organization
Spring 2023: Lecture 11

Instructor: Yuhao Zhu

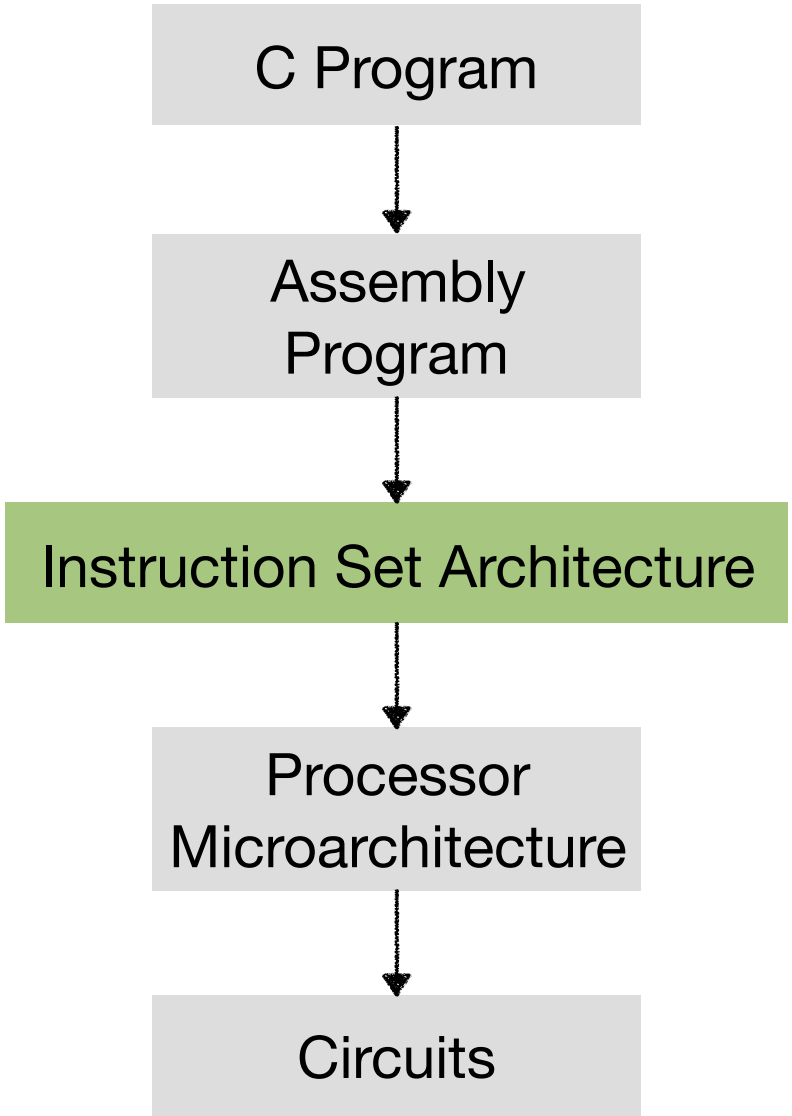
Department of Computer Science
University of Rochester

Announcement

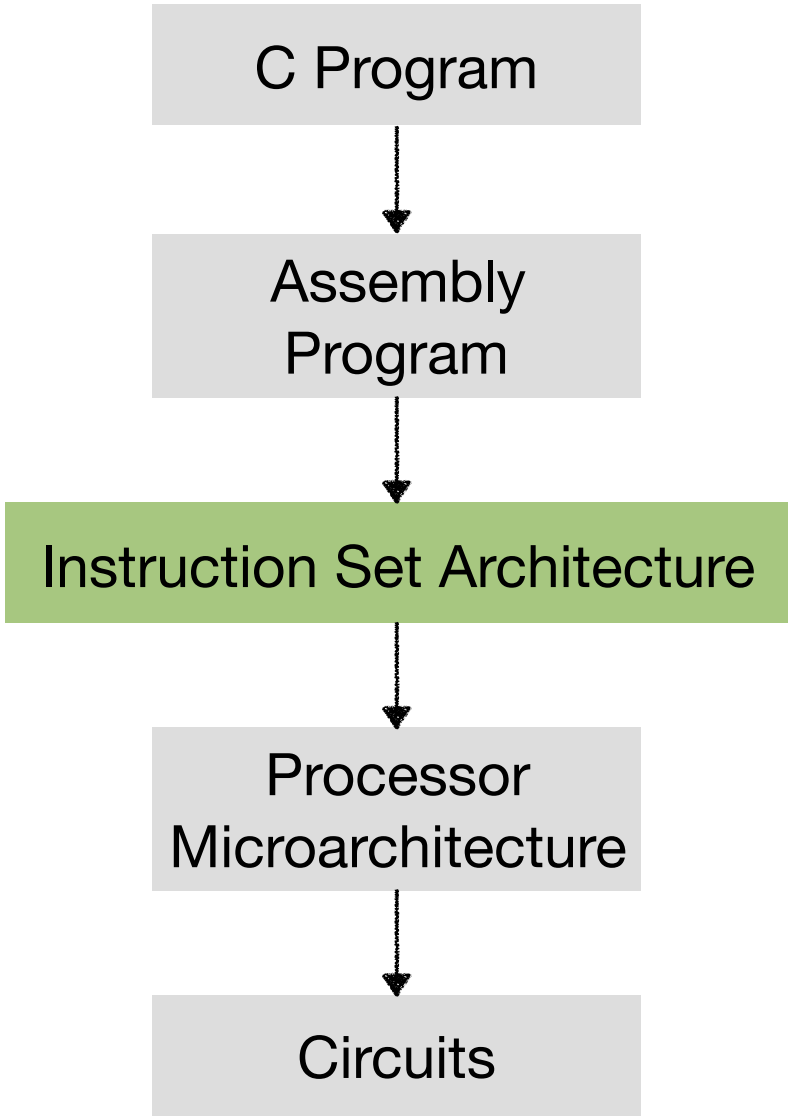
- Programming assignment 3 out.

12	13	14	15 **Today**	16	17	18
19	20	21	22	23	24	25
26	27	28	Mar 1 **Due**	2	3	4

So far in 252…

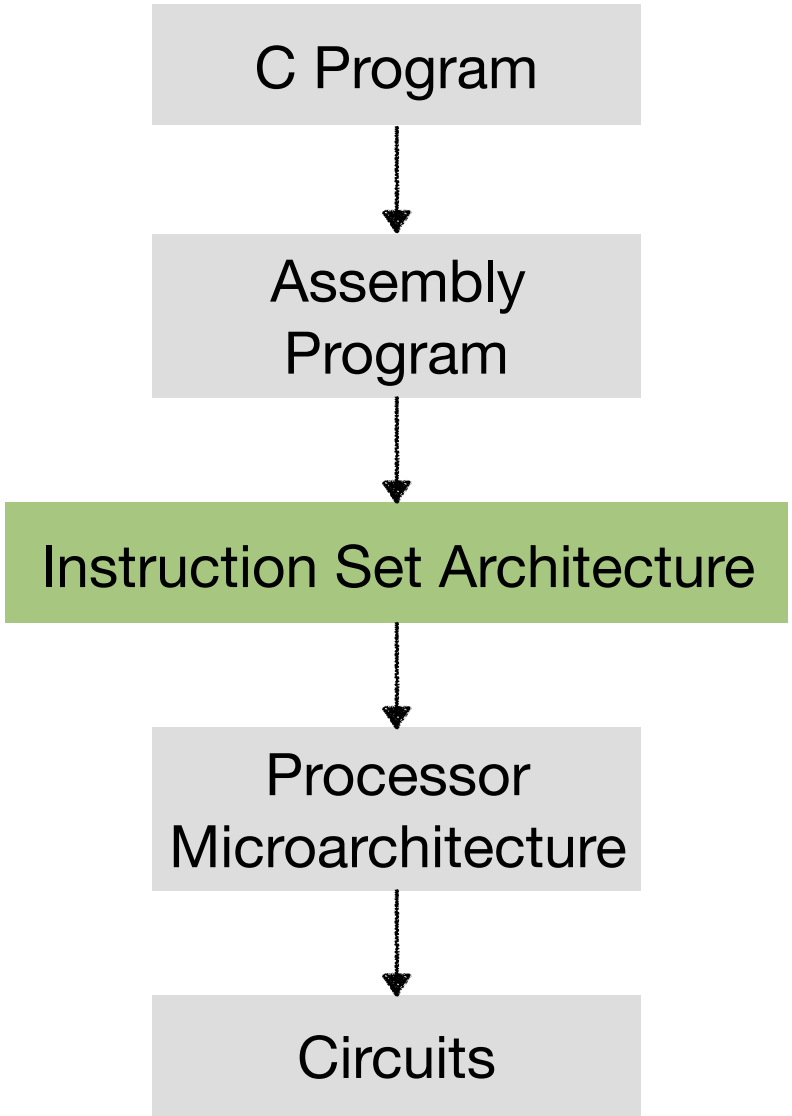

C Program
Assembly Program
Instruction Set Architecture
Processor Microarchitecture
Circuits

So far in 252…




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

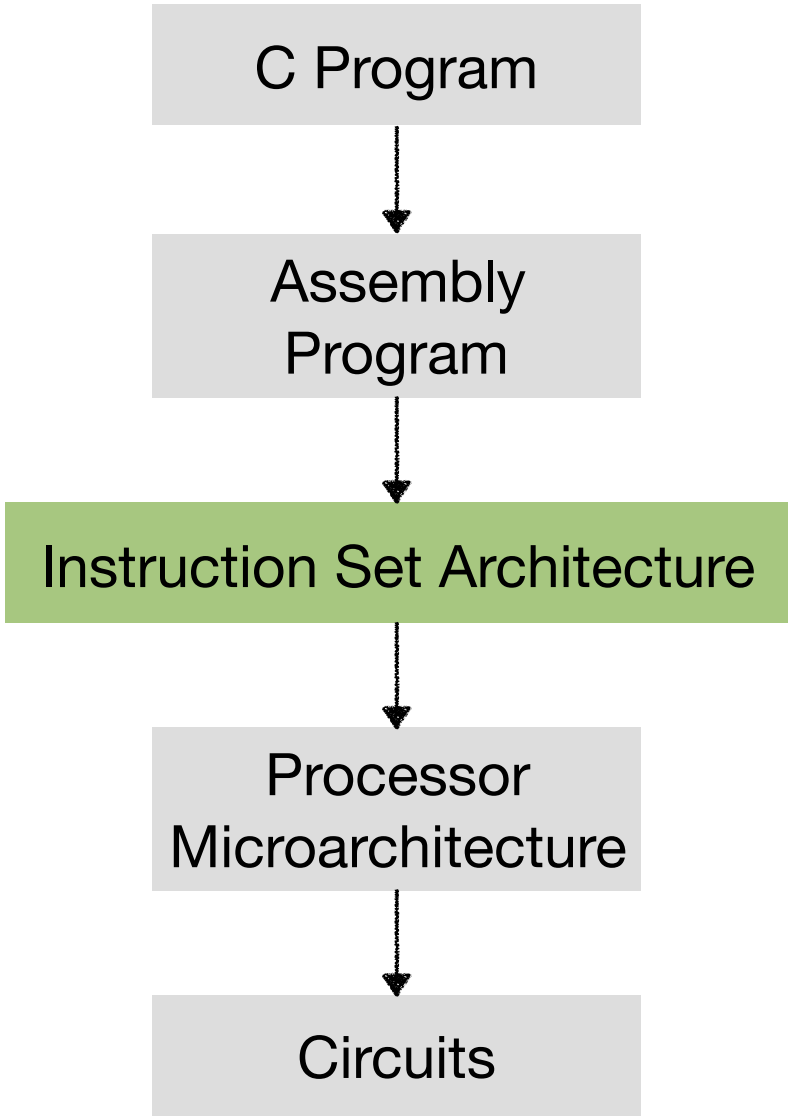
So far in 252…




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

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

So far in 252…

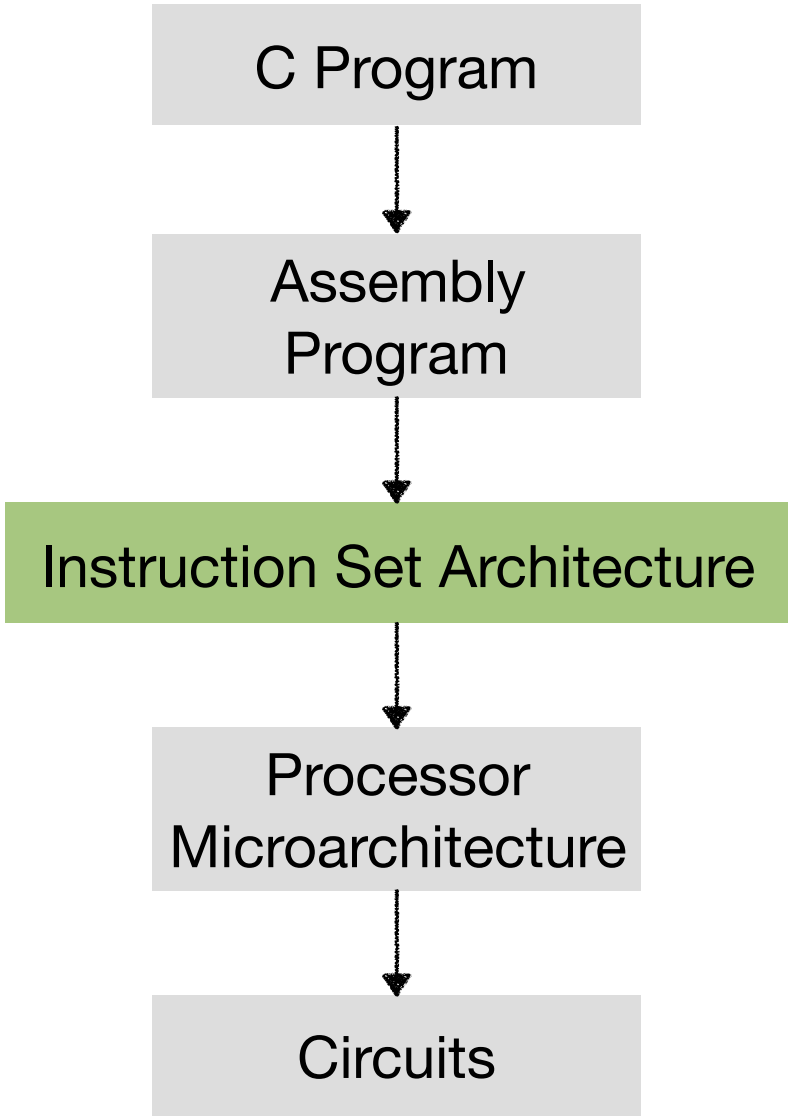



```
int, float
if, else
+, -, >>
```

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

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

So far in 252…



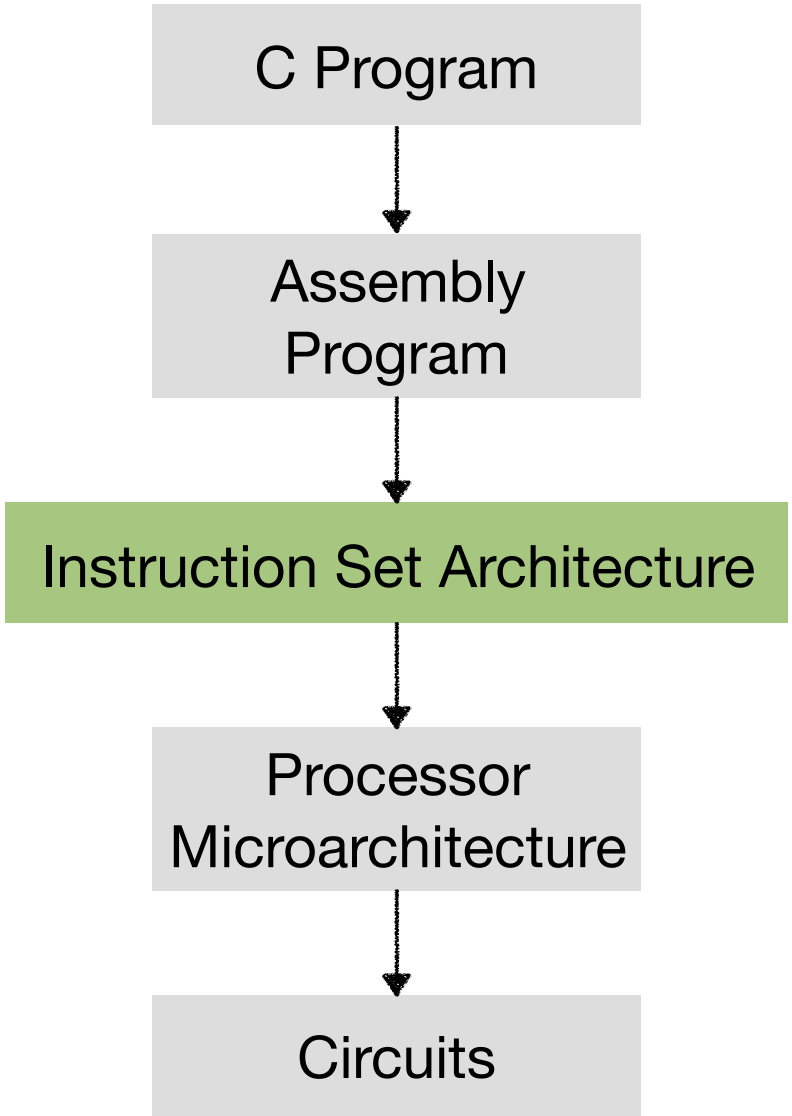

```
int, float
if, else
+, -, >>
```

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

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

Logic gates

So far in 252…




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

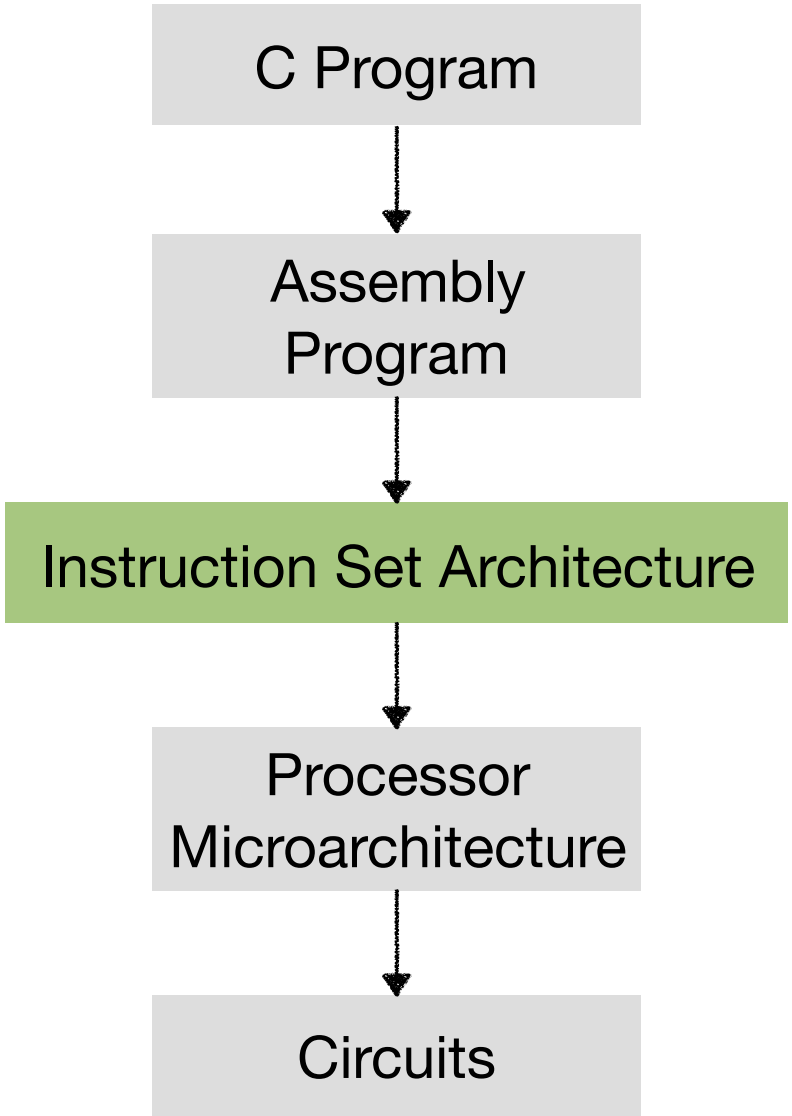
```
movq    %rsi, %rax
imulq   %rdx, %rax
jmp     .done
```

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

Logic gates

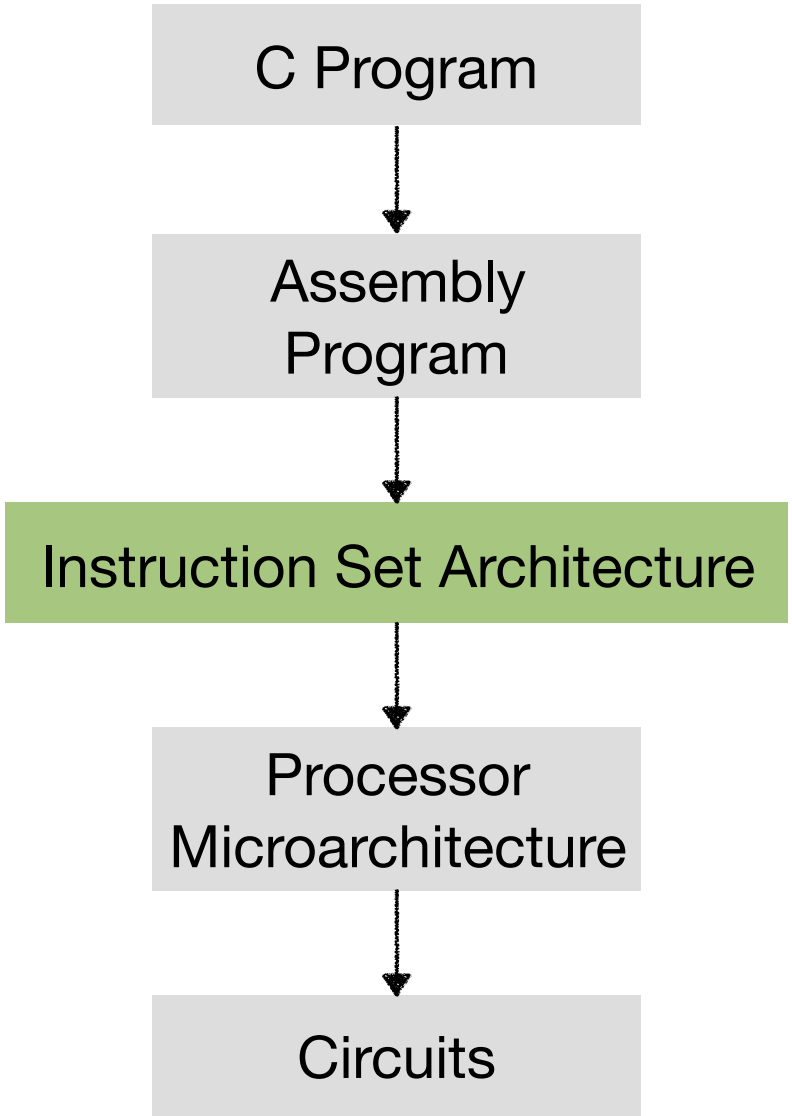
Transistors

So far in 252…



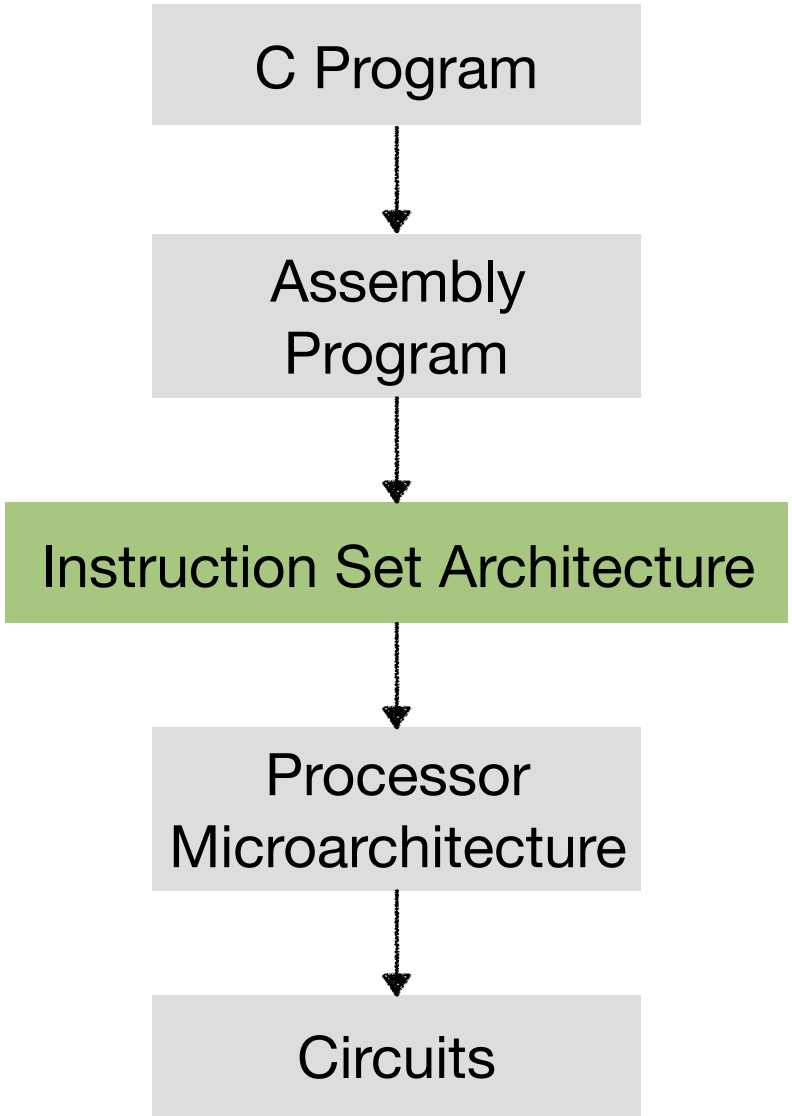
- ISA is the interface between assembly programs and microarchitecture

So far in 252…



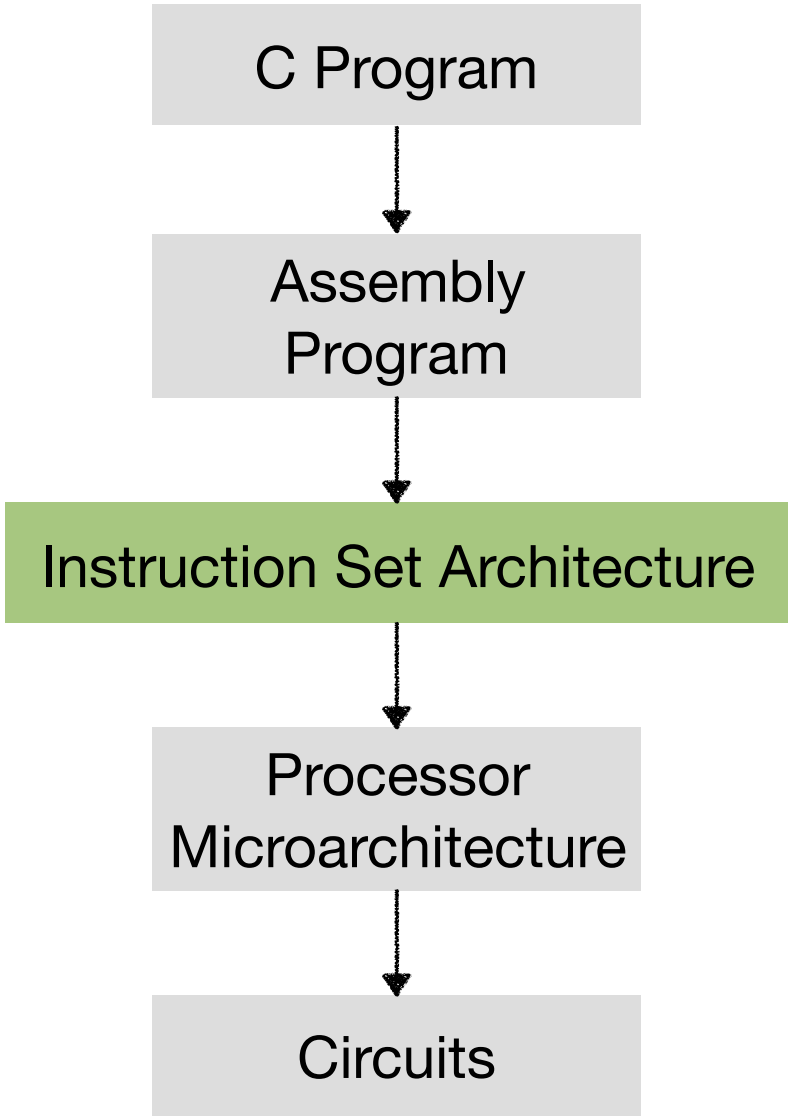
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:

So far in 252…



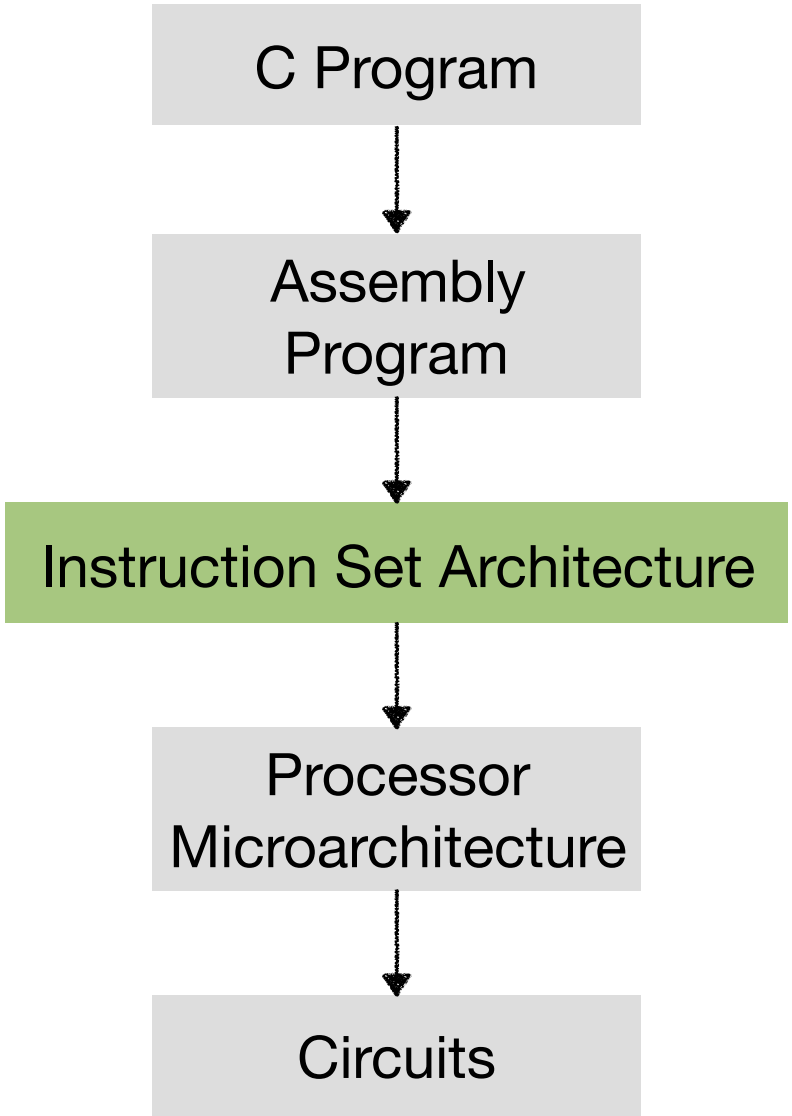
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?

So far in 252…



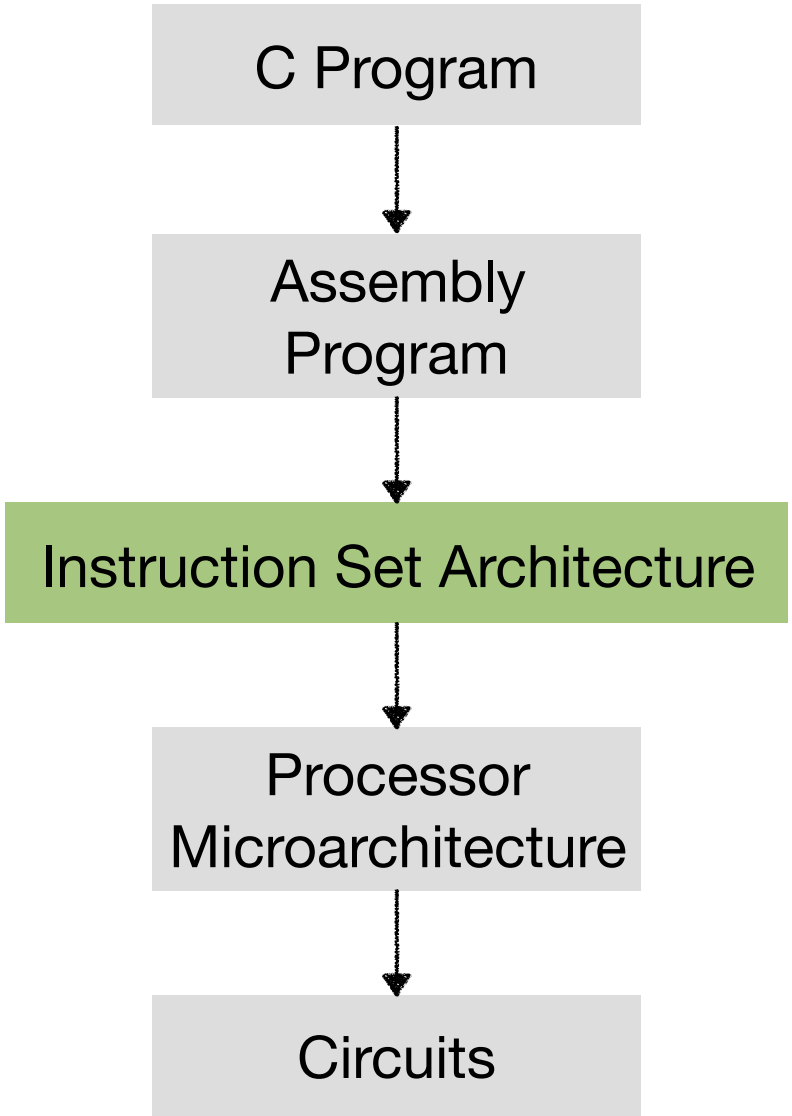
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.

So far in 252…



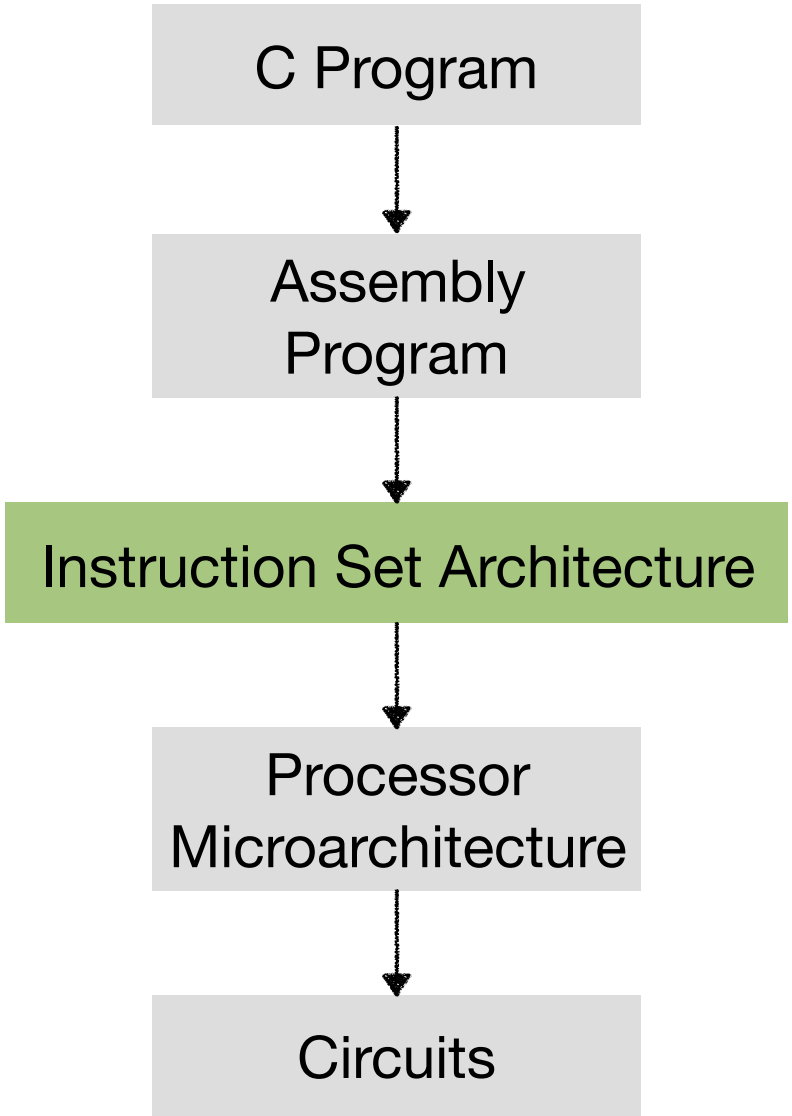
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.
- Microarchitecture view:

So far in 252…



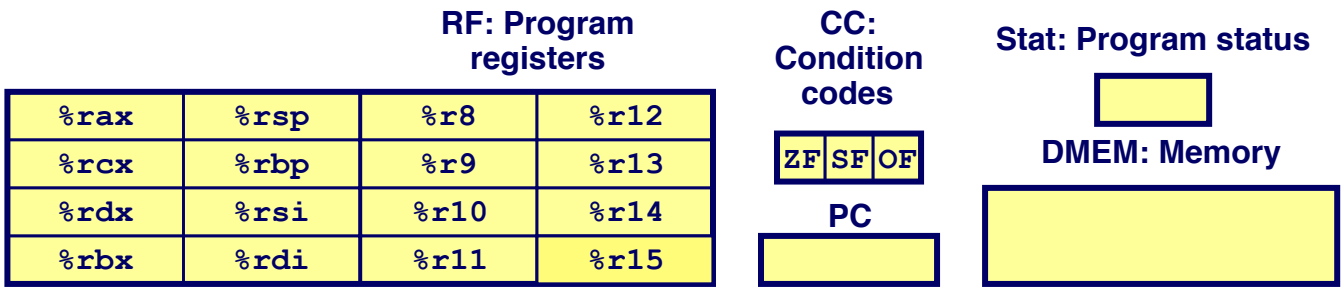
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.
- Microarchitecture view:
  - What hardware needs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

So far in 252…



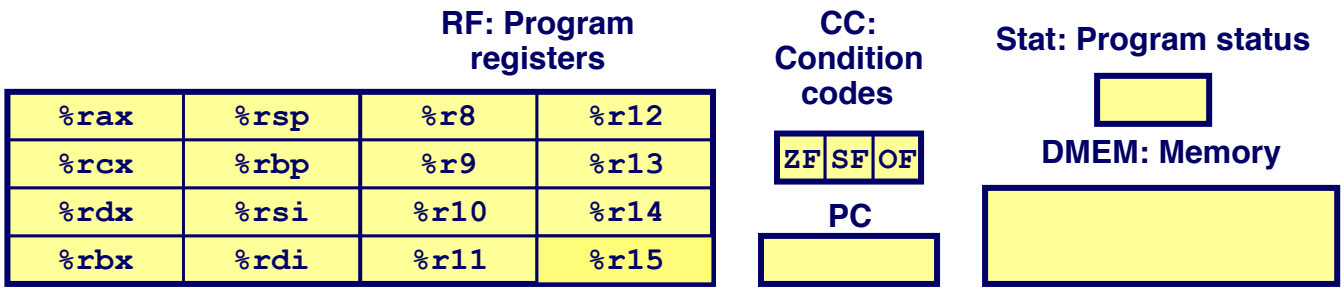
- ISA is the interface between assembly programs and microarchitecture
- Assembly view:
  - How to program the machine, based on instructions and **processor states** (registers, memory, condition codes, etc.)?
  - Instructions are executed sequentially.
- Microarchitecture view:
  - What hardware needs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

(Simplified) x86 Processor State



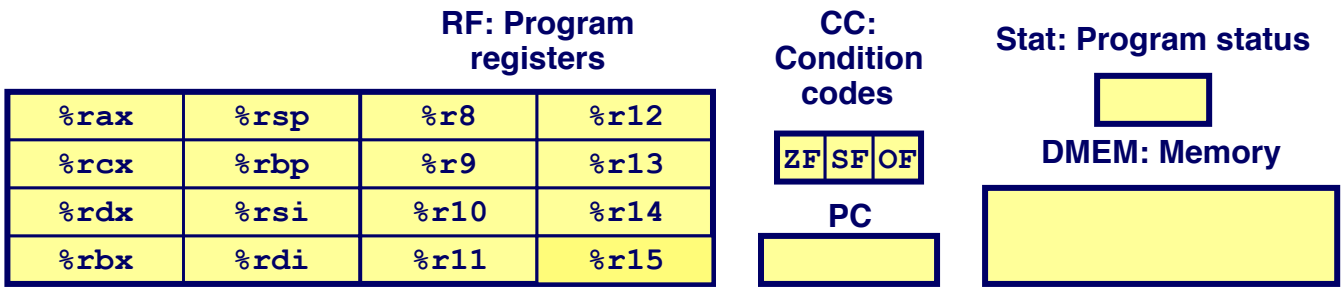
- Processor state is what's visible to assembly programs. Also known as architecture state.

(Simplified) x86 Processor State



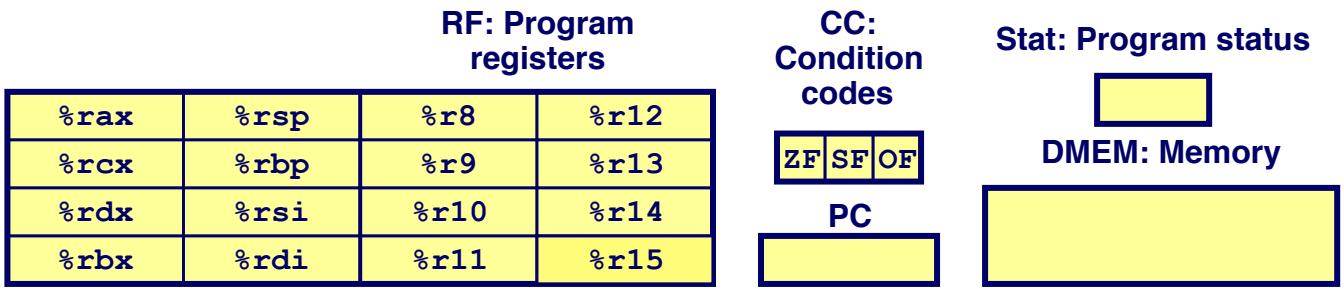
- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 16 registers.

(Simplified) x86 Processor State



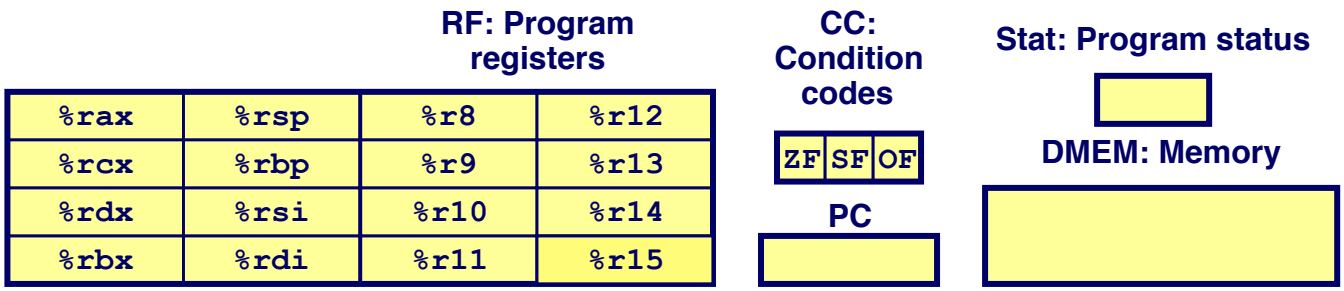
- Processor state is what's visible to assembly programs. Also known as architecture state.
- Program Registers: 16 registers.
- Condition Codes: Single-bit flags set by arithmetic or logical instructions (ZF, SF, OF)

(Simplified) x86 Processor State



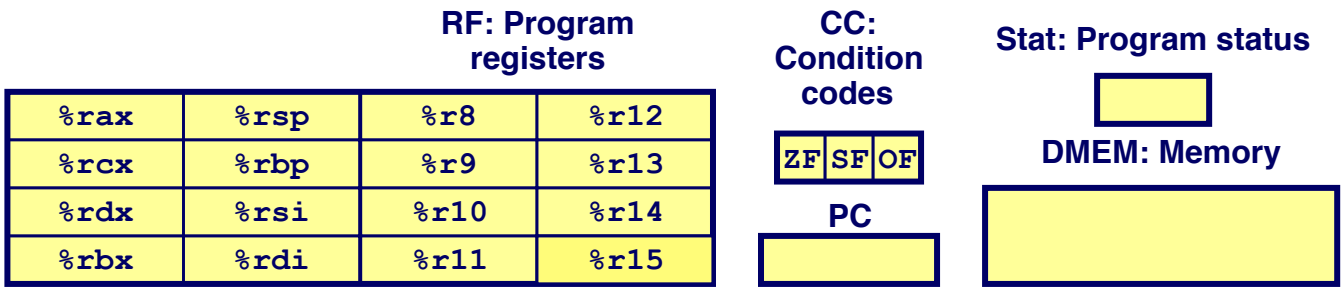
- Processor state is what's visible to assembly programs. Also known as architecture state.
- **Program Registers:** 16 registers.
- **Condition Codes:** Single-bit flags set by arithmetic or logical instructions (ZF, SF, OF)
- **Program Counter:** Indicates address of next instruction

(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?

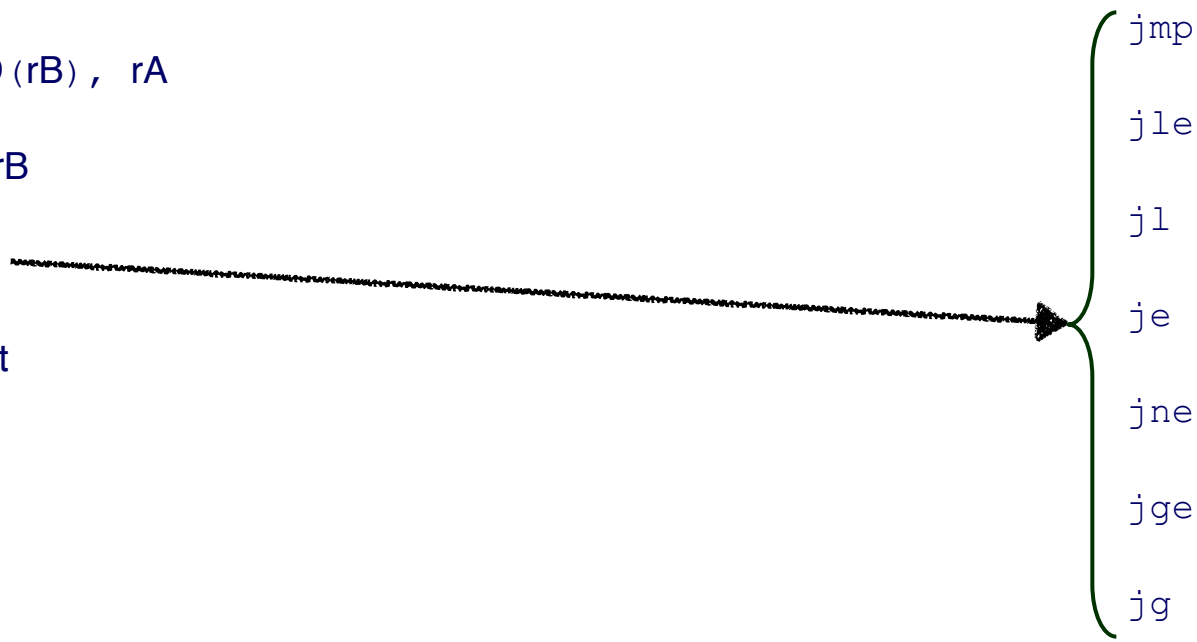
- Remember that instructions are stored in memory **as bits** (just like data)
- Each instruction is fetched (according to the address specified in the PC), decoded, and executed by the CPU
- The ISA defines the format of an instruction (syntax) and its meaning (semantics)
- Idea: encode the two major 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
```

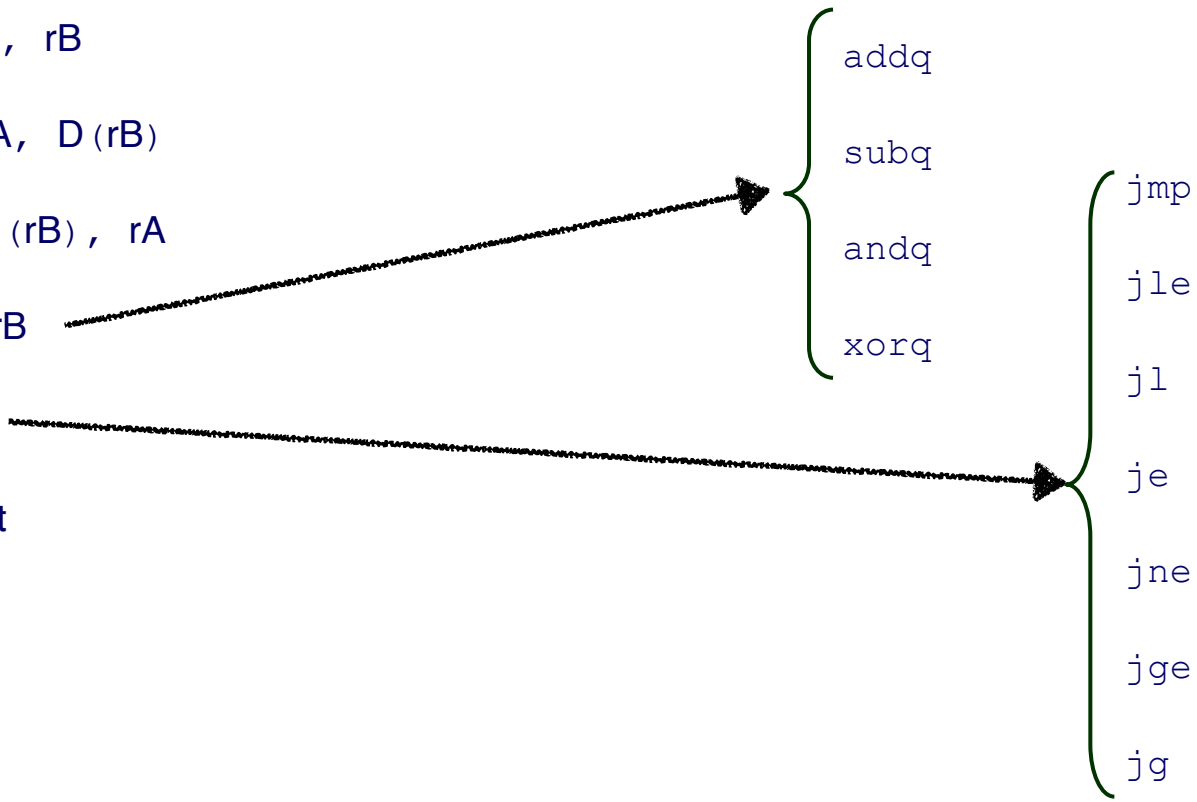
Y86-64 Instructions

```
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
```



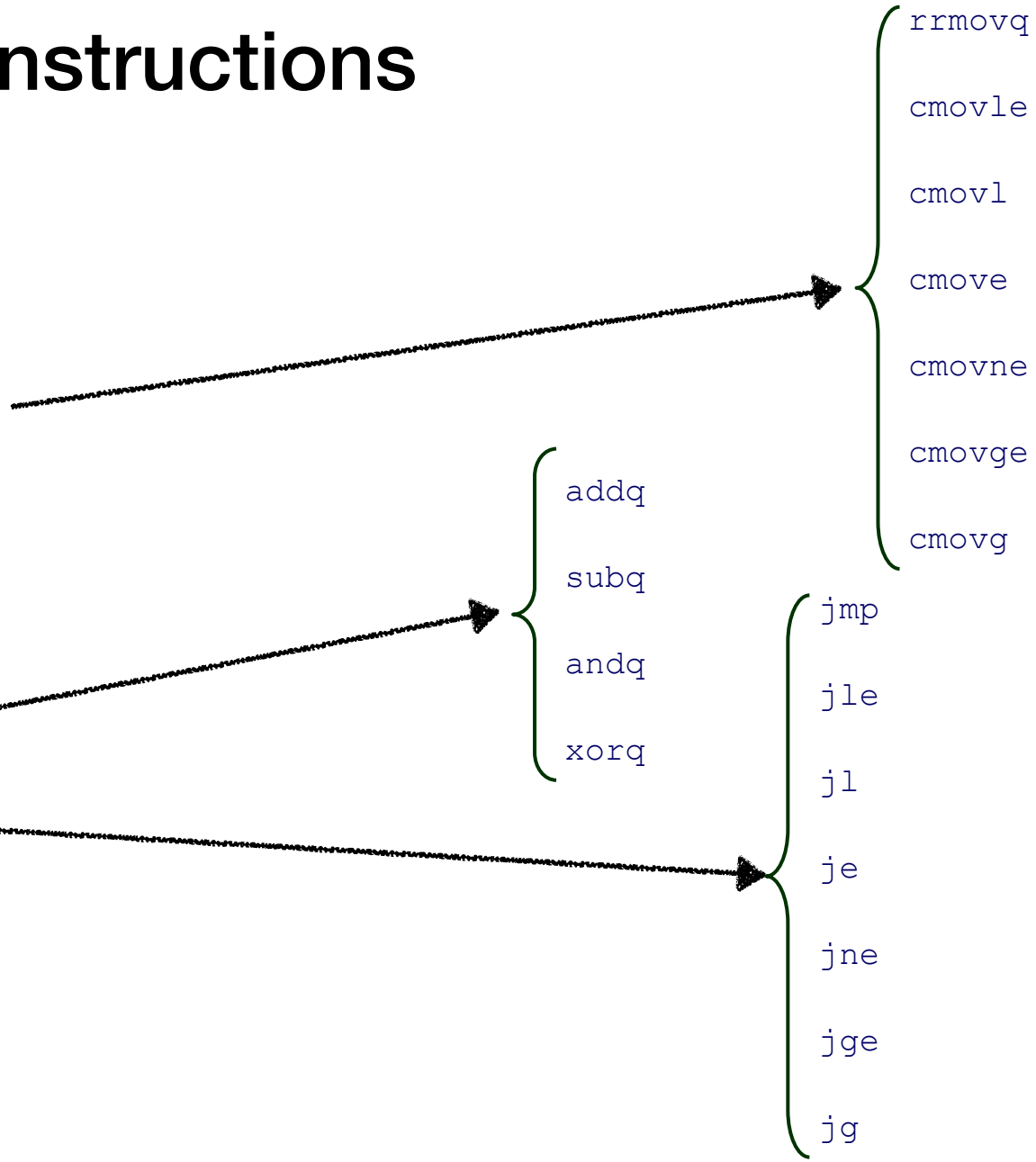
Y86-64 Instructions

```
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
```



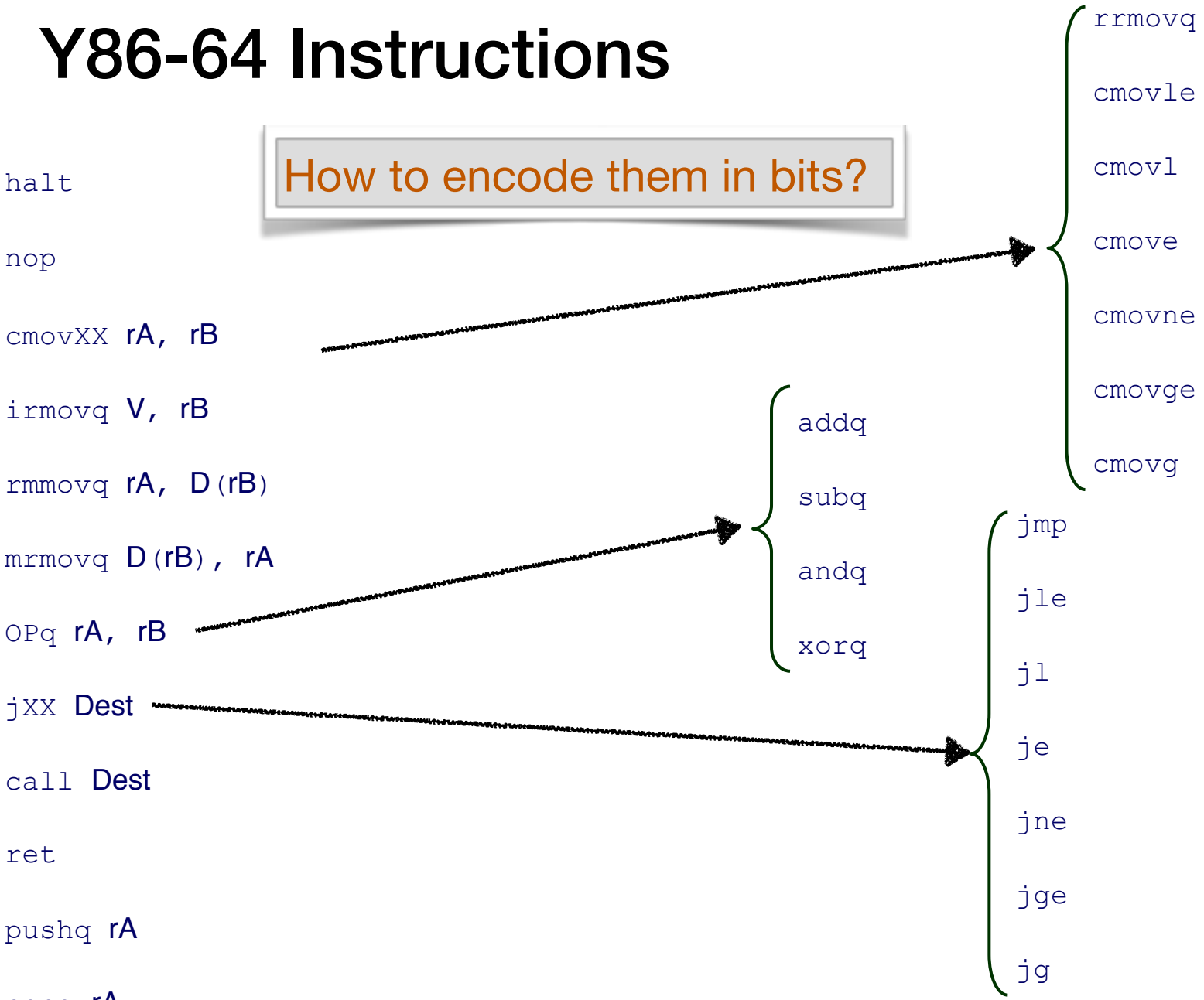
Y86-64 Instructions

```
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
```

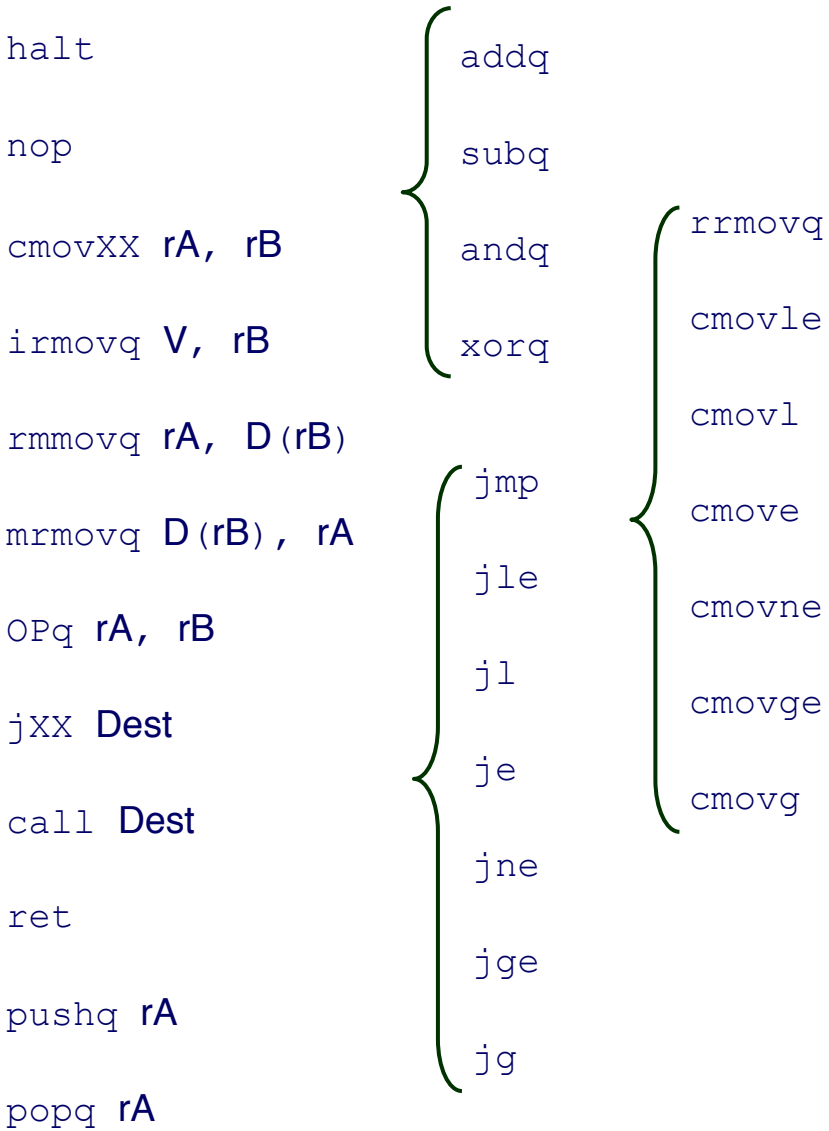


Y86-64 Instructions

How to encode them in bits?

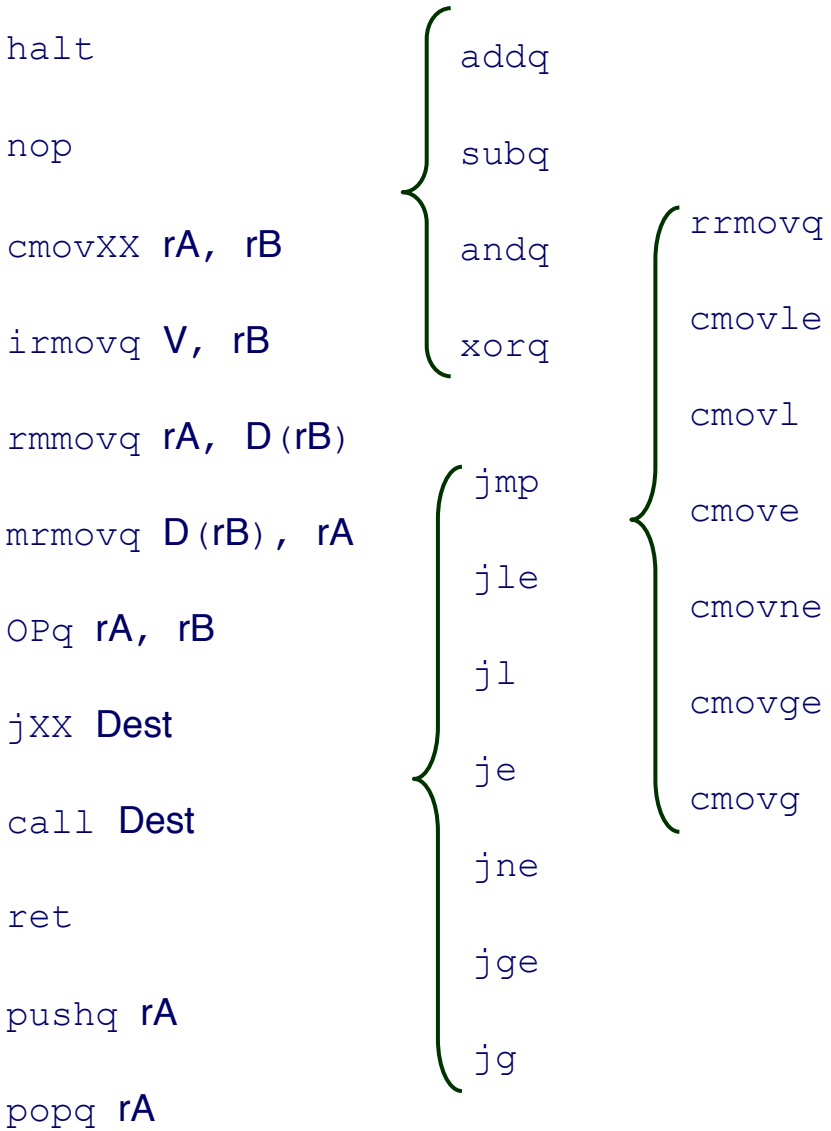


Encoding Operands



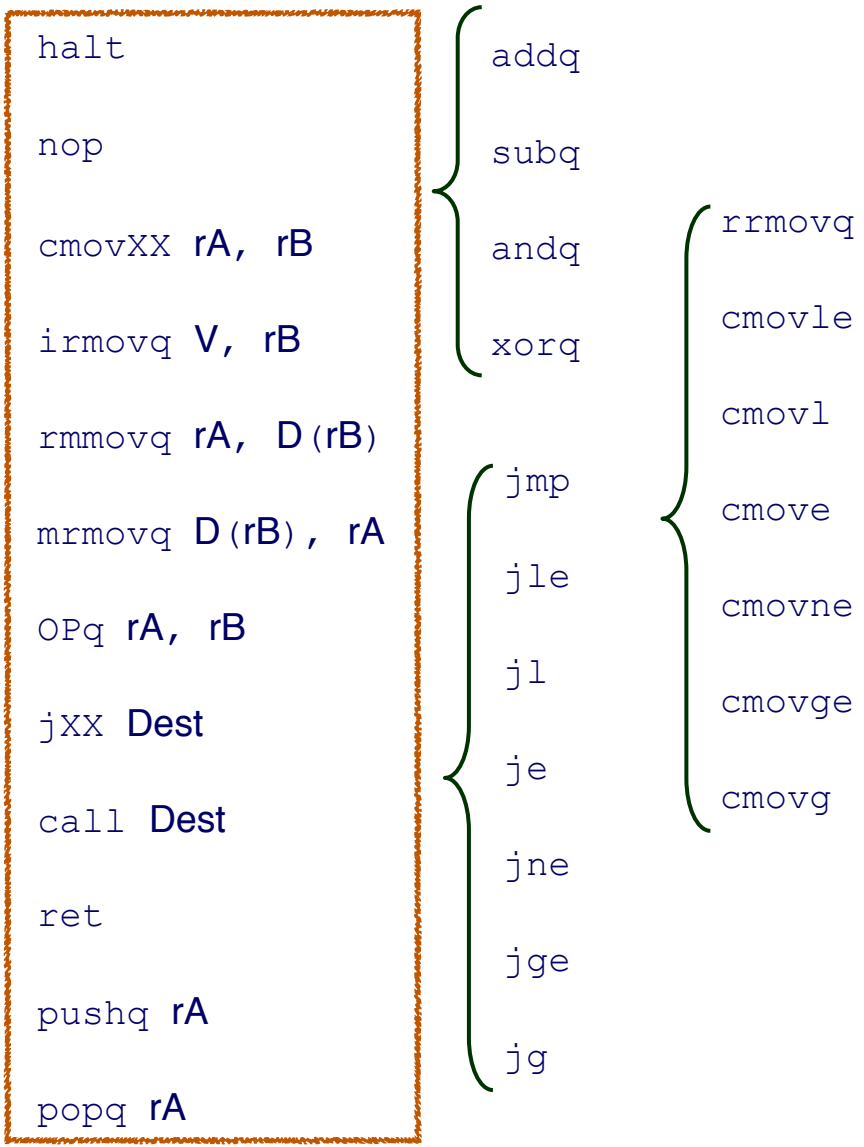
- 27 Instructions, so need 5 bits for encoding the operand

Encoding Operands



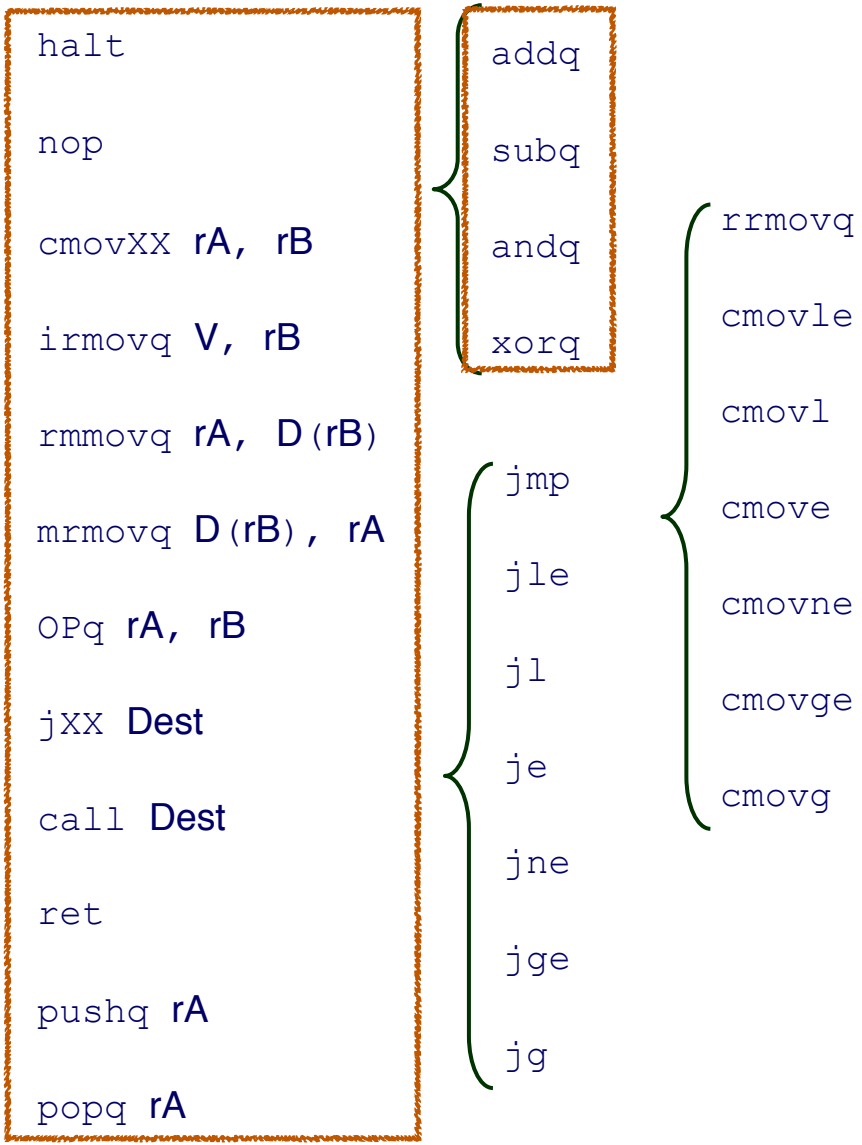
- 27 Instructions, so need 5 bits for encoding the operand
- Or: group similar instructions, use one opcode for them, and then use more bits to indicate specific instructions within a group.

Encoding Operands



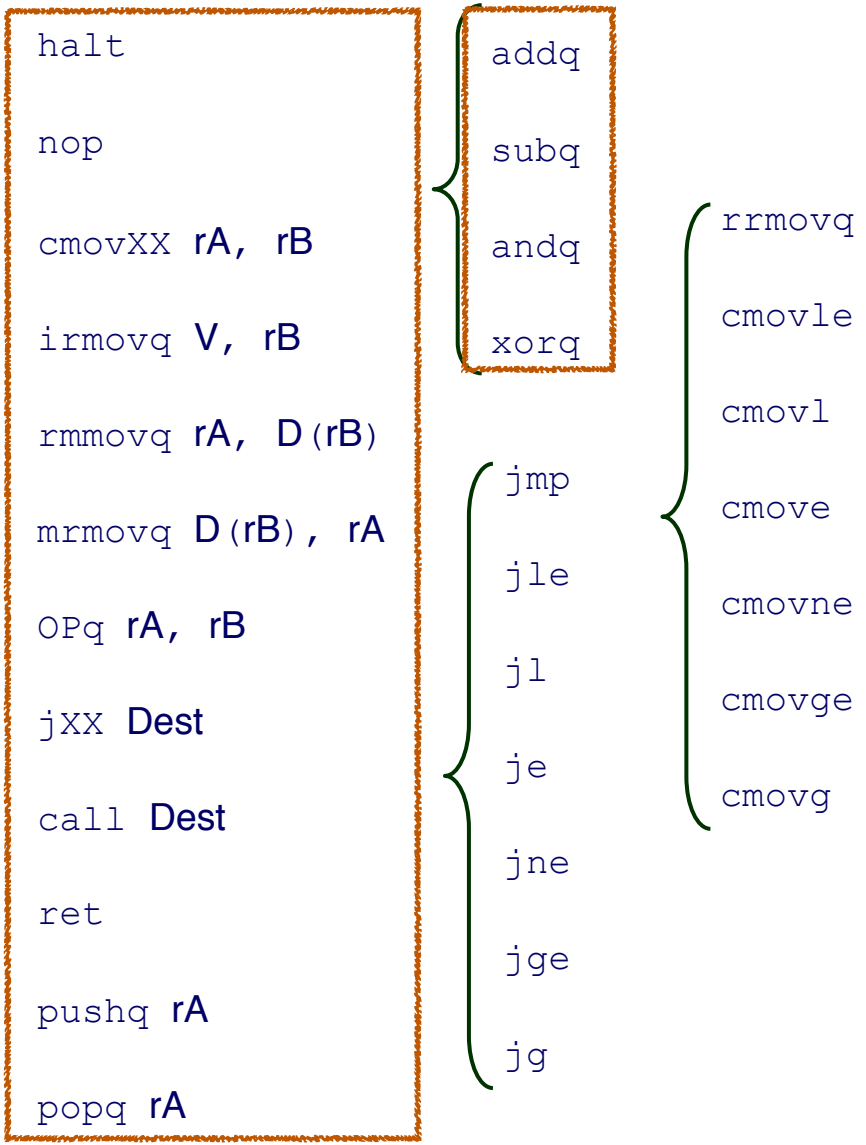
- 27 Instructions, so need 5 bits for encoding the operand
- Or: group similar instructions, use one opcode for them, and then use more bits to indicate specific instructions within a group.
- E.g., 12 categories, so 4 bits

Encoding Operands



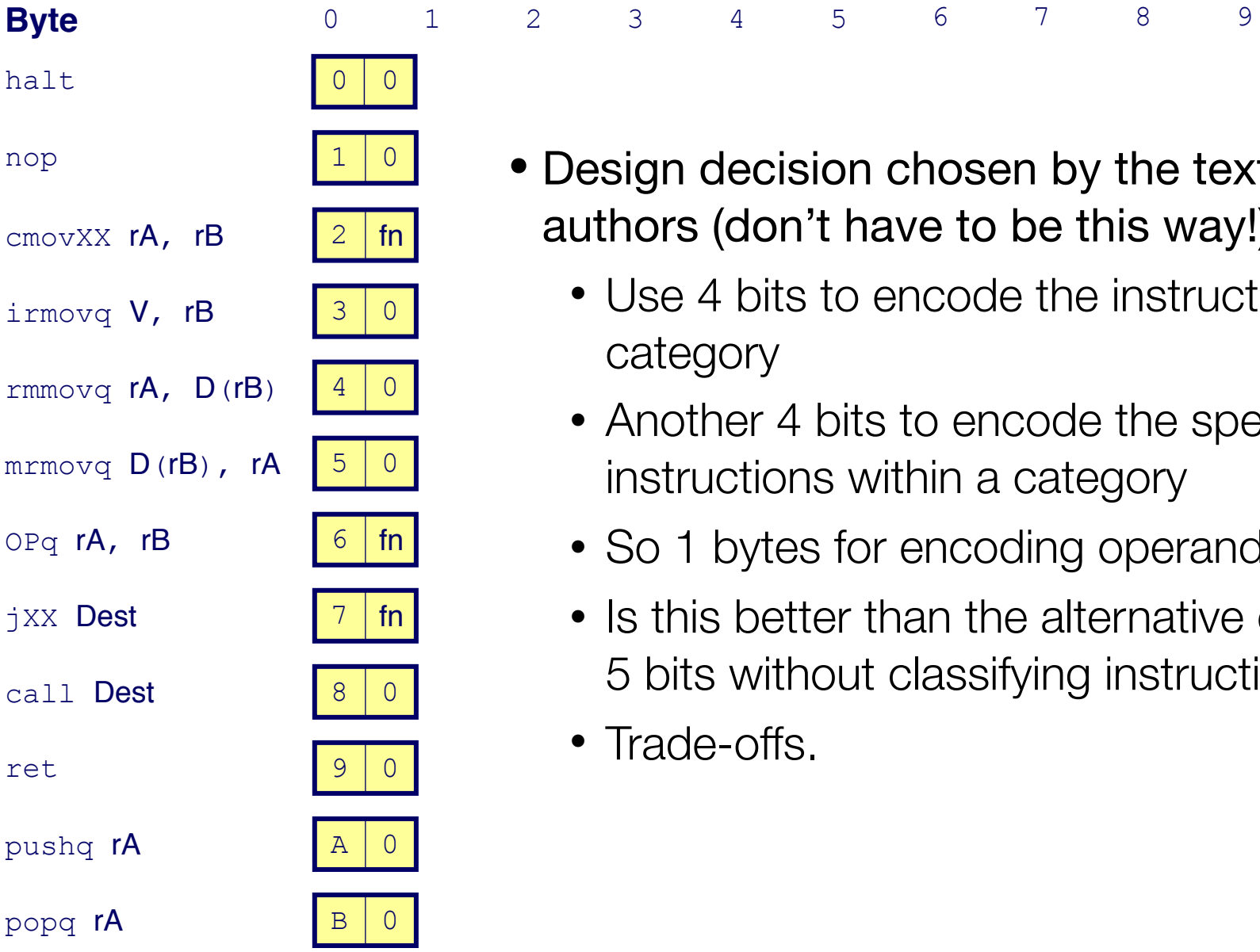
- 27 Instructions, so need 5 bits for encoding the operand
- Or: group similar instructions, use one opcode for them, and then use more bits to indicate specific instructions within a group.
- E.g., 12 categories, so 4 bits
- There are four instructions within the `OPq` category, so additional 2 bits.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

Encoding Operands



- 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



- 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.

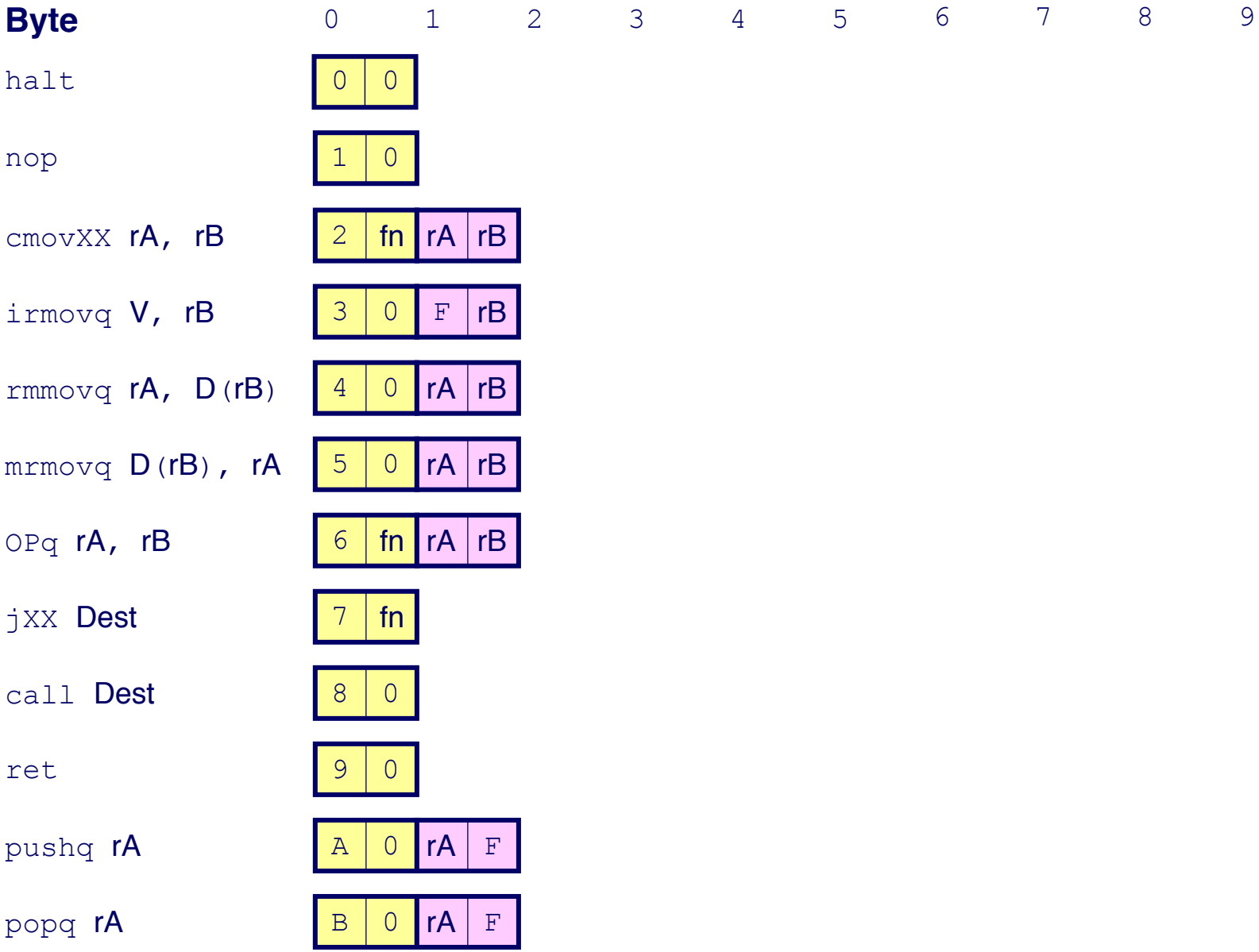
Encoding Registers

Each register has 4-bit ID

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

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

Encoding Registers



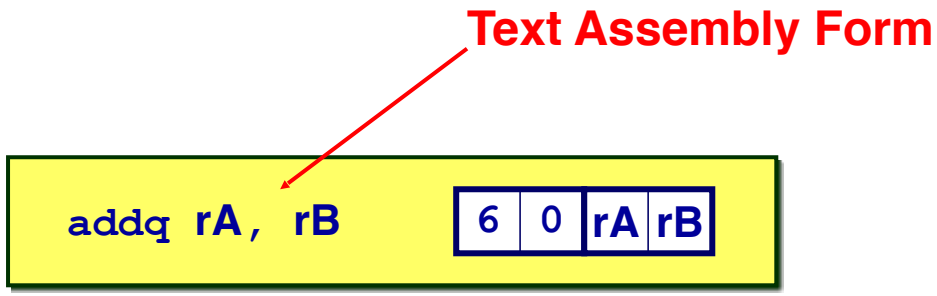
Instruction Example

Addition Instruction

- 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

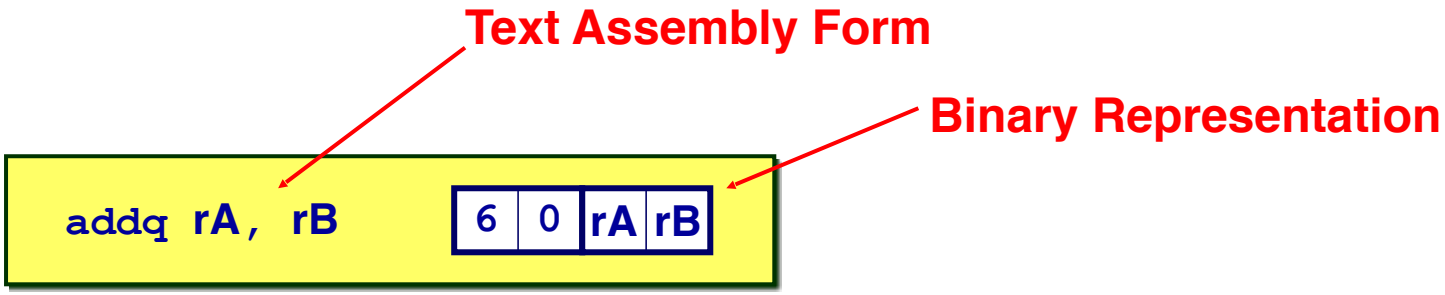
Addition Instruction



- 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



- 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

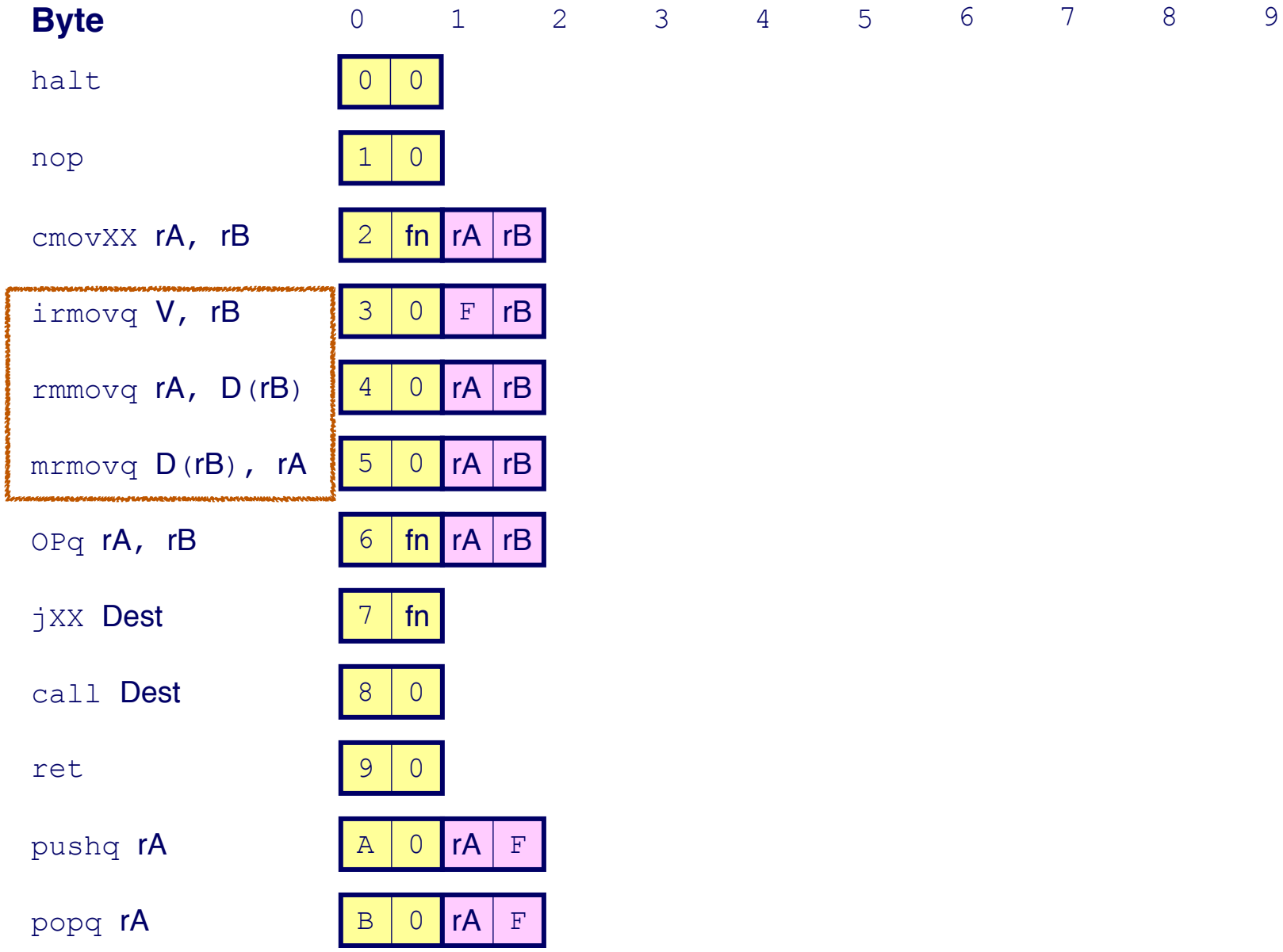
Subtract (rA from rB)

And

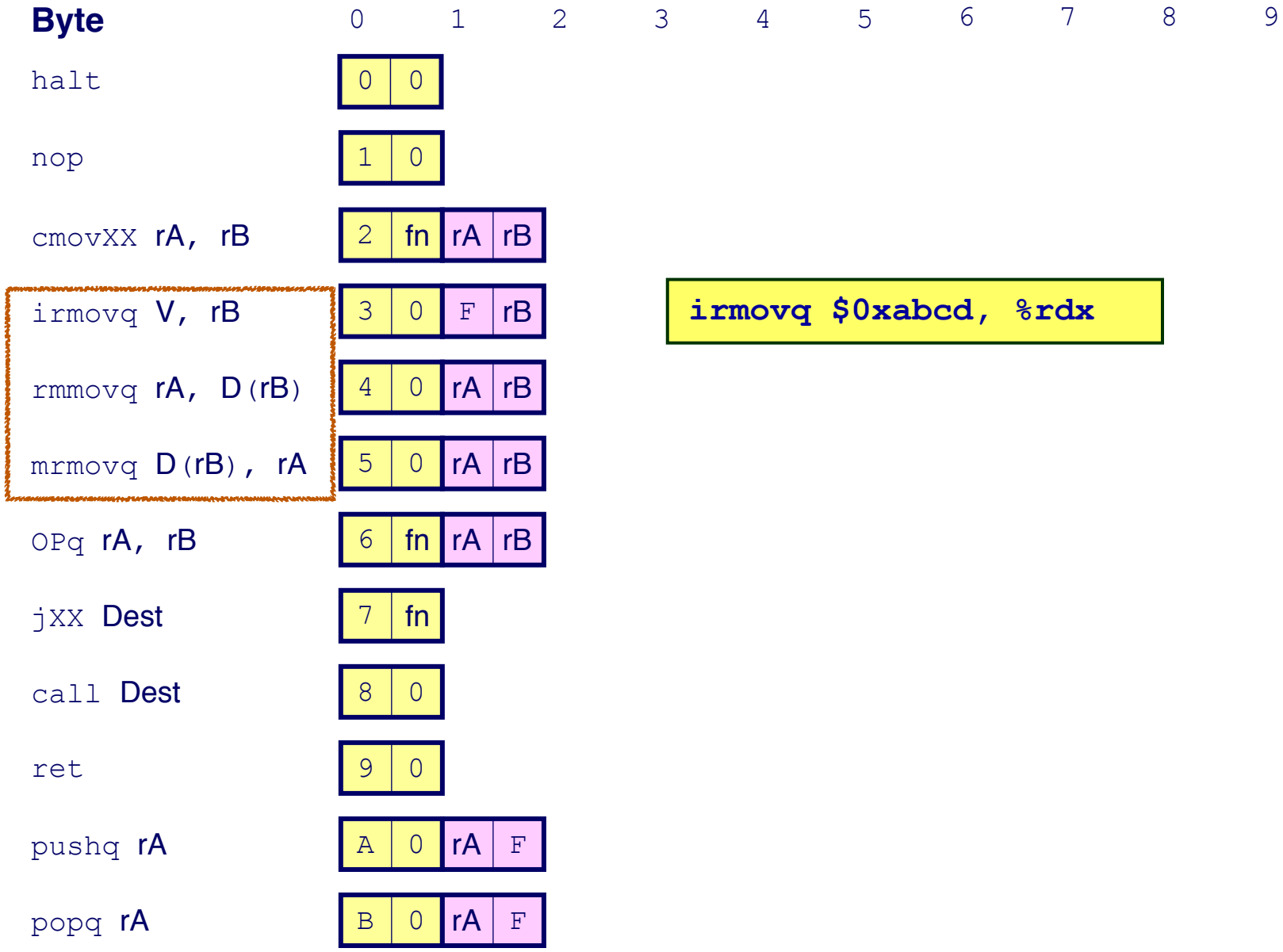
Exclusive-Or

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

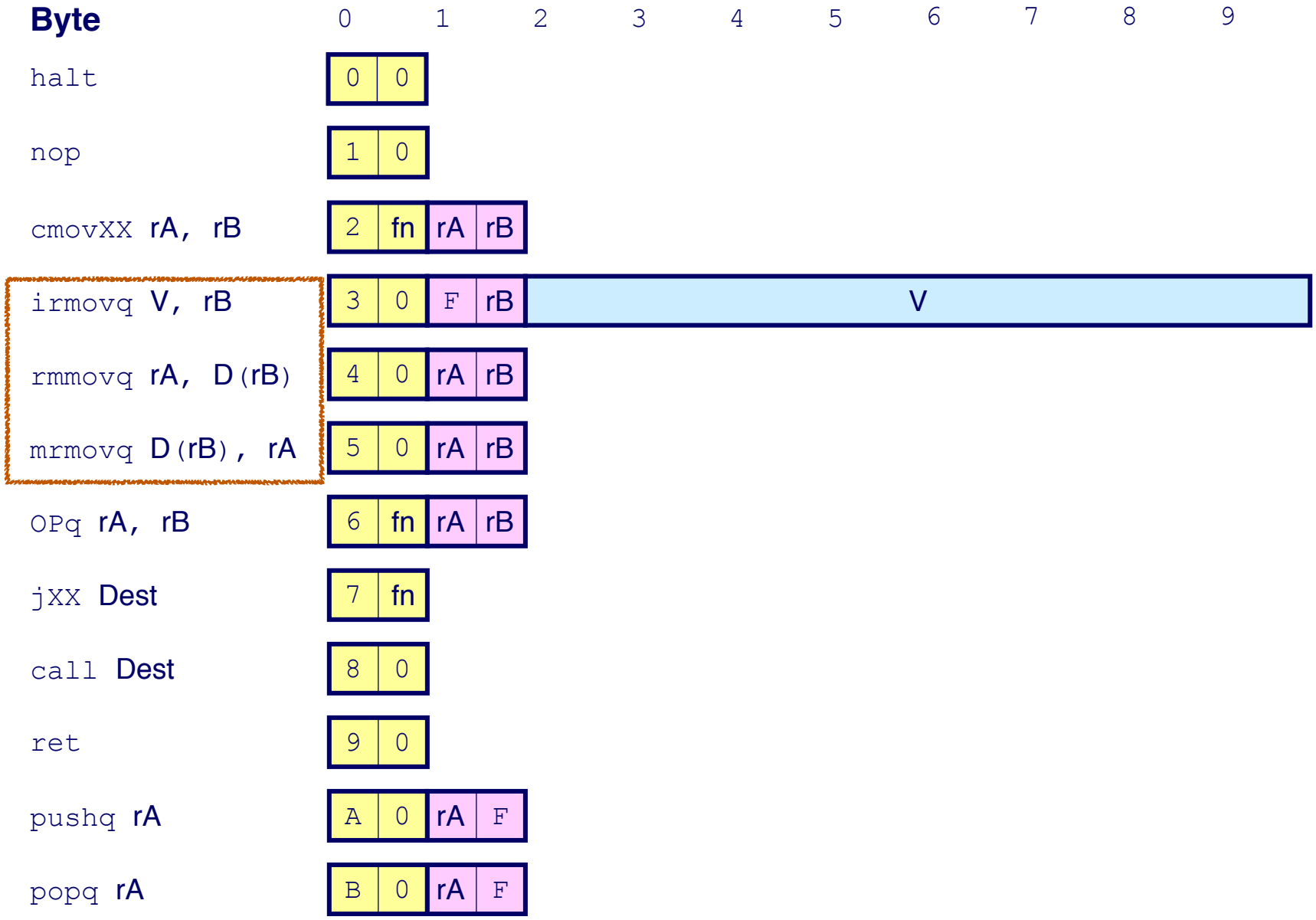
Move Instructions



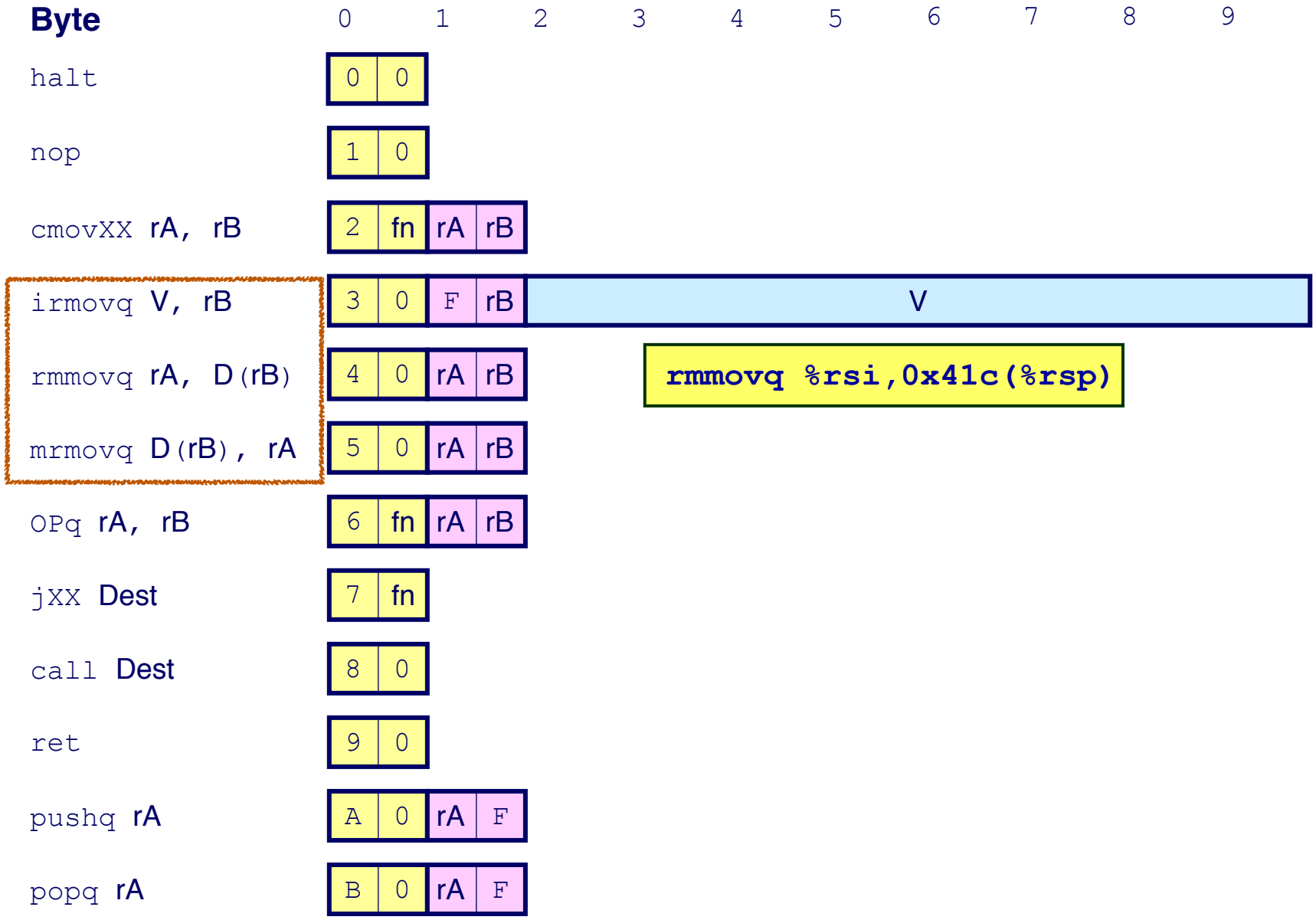
Move Instructions



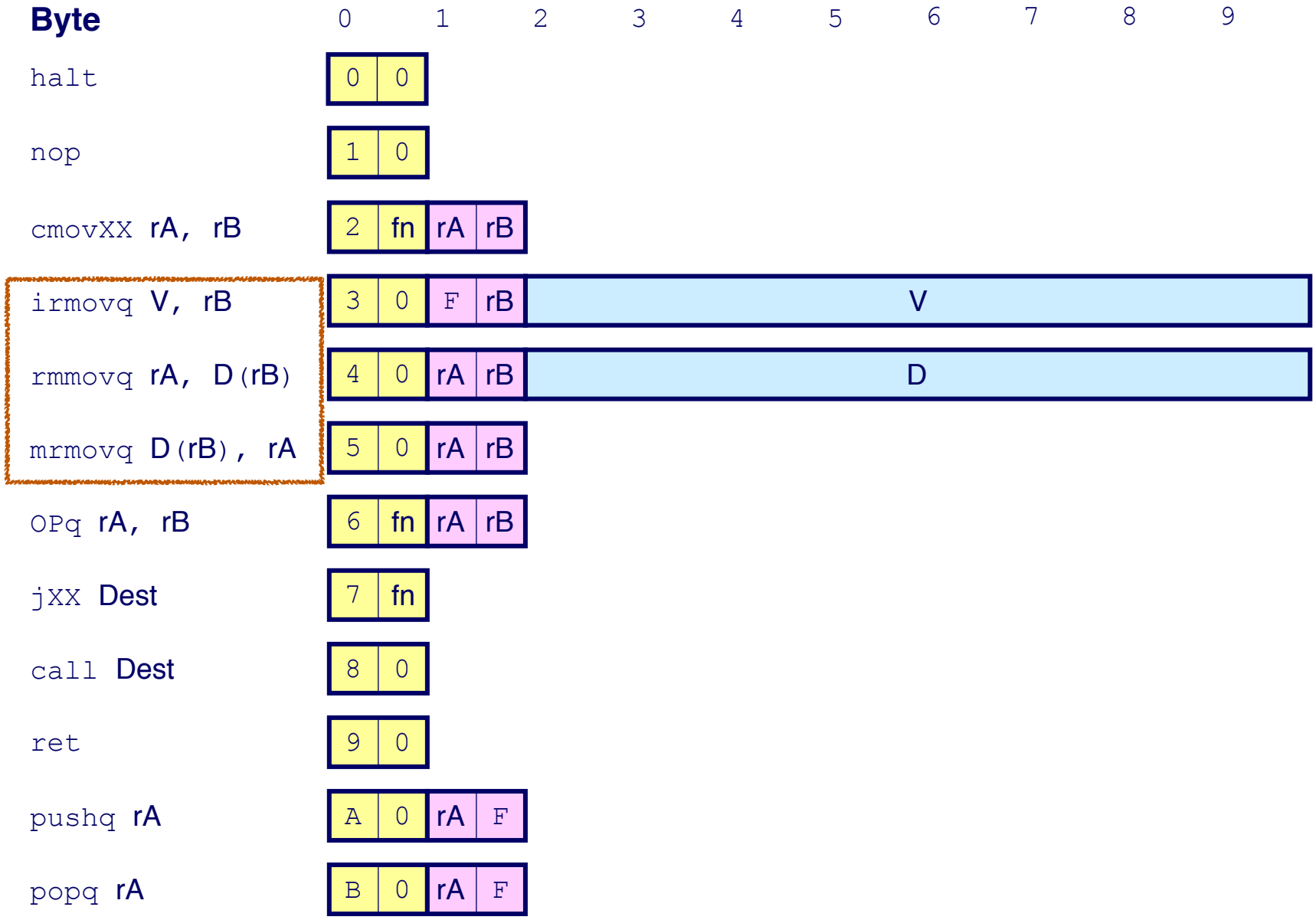
Move Instructions



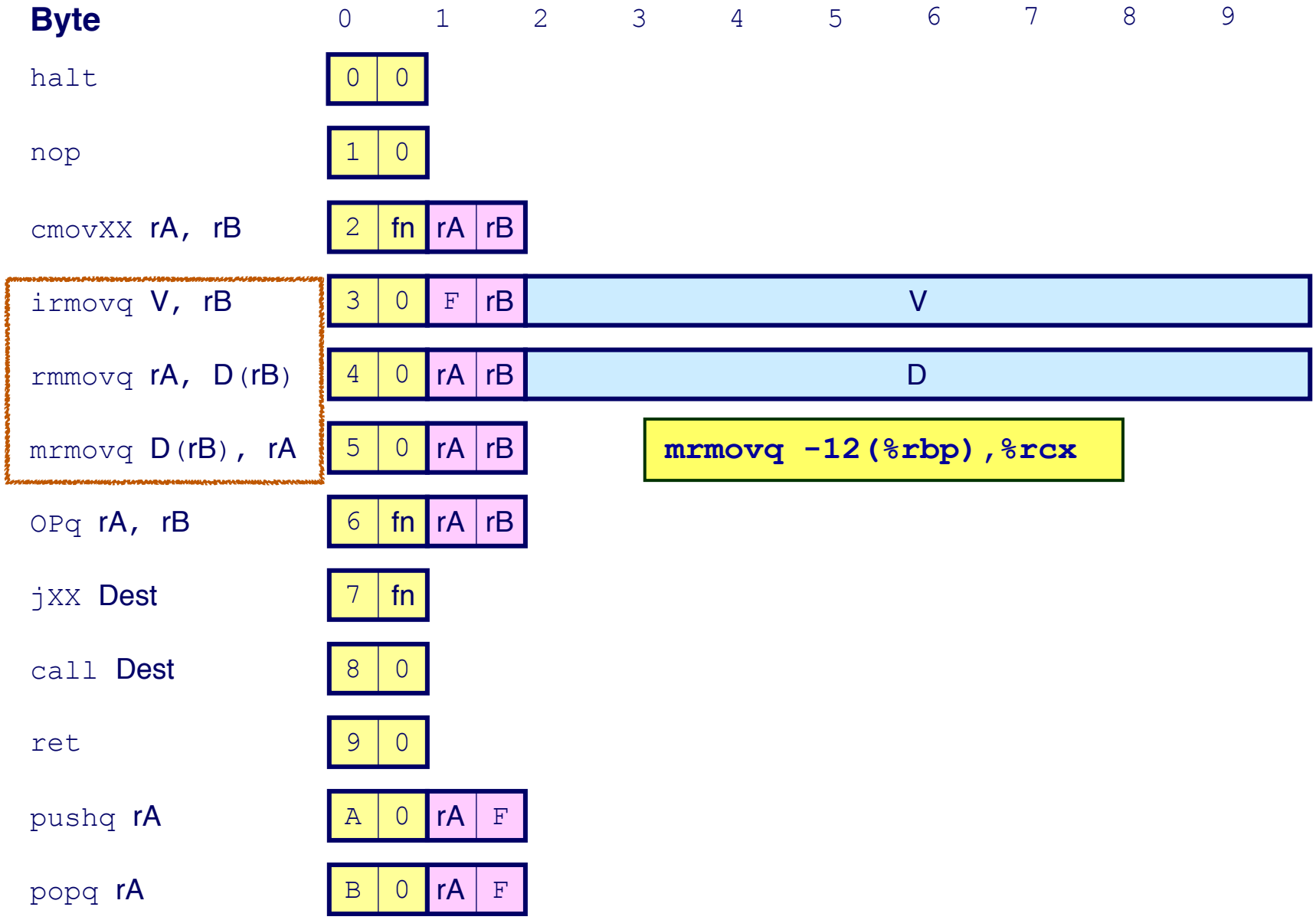
Move Instructions



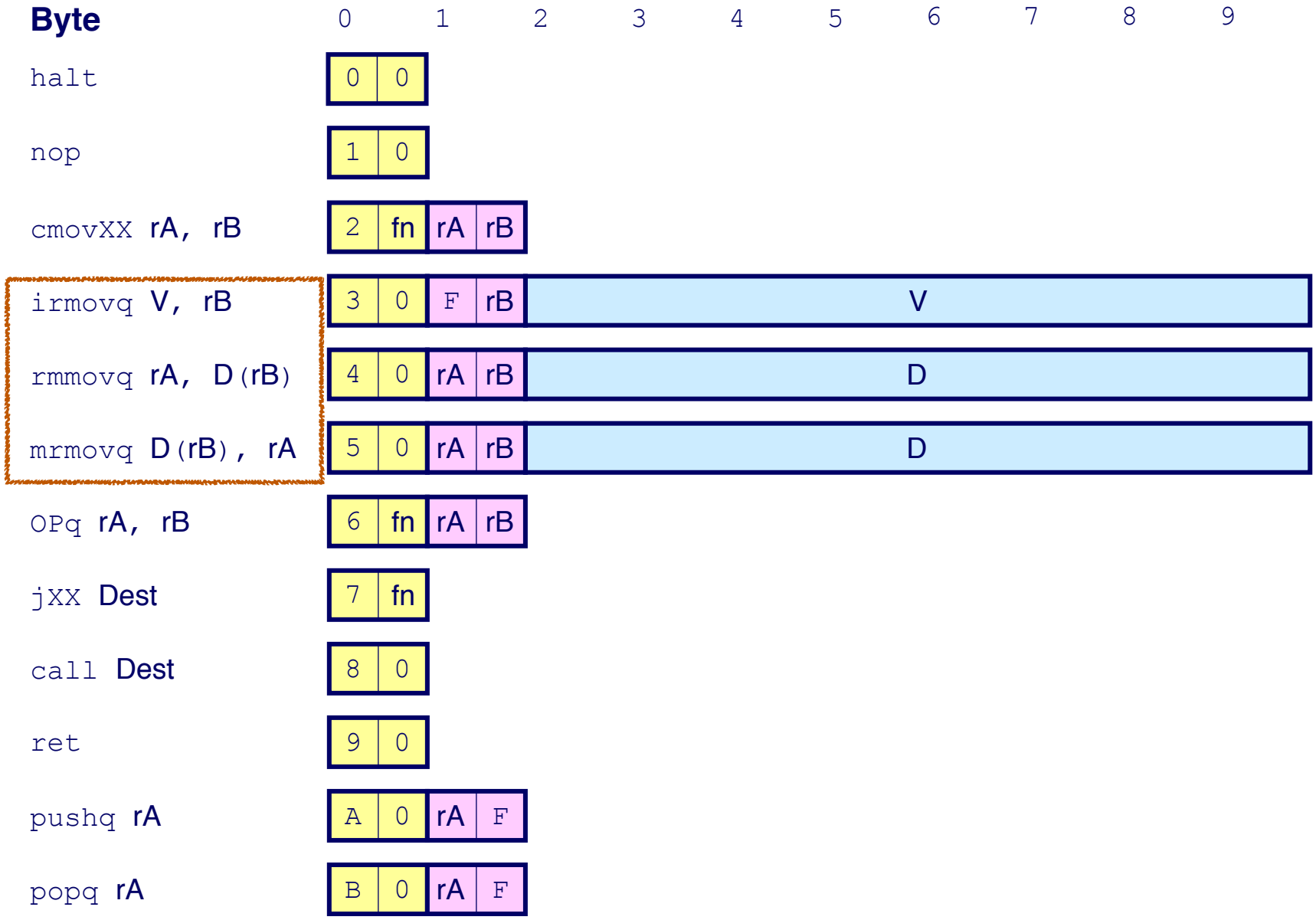
Move Instructions



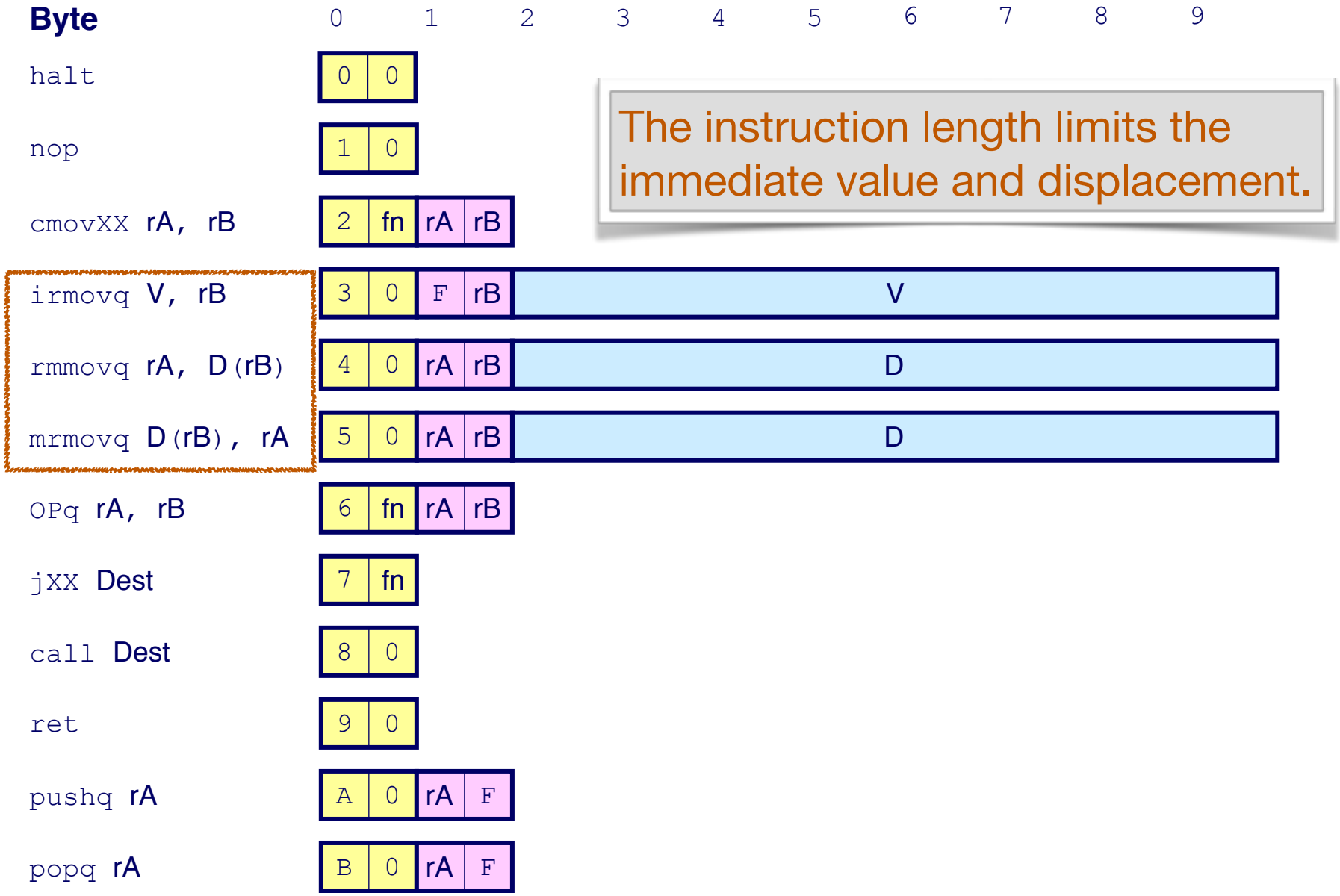
Move Instructions



Move Instructions



Move Instructions



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`

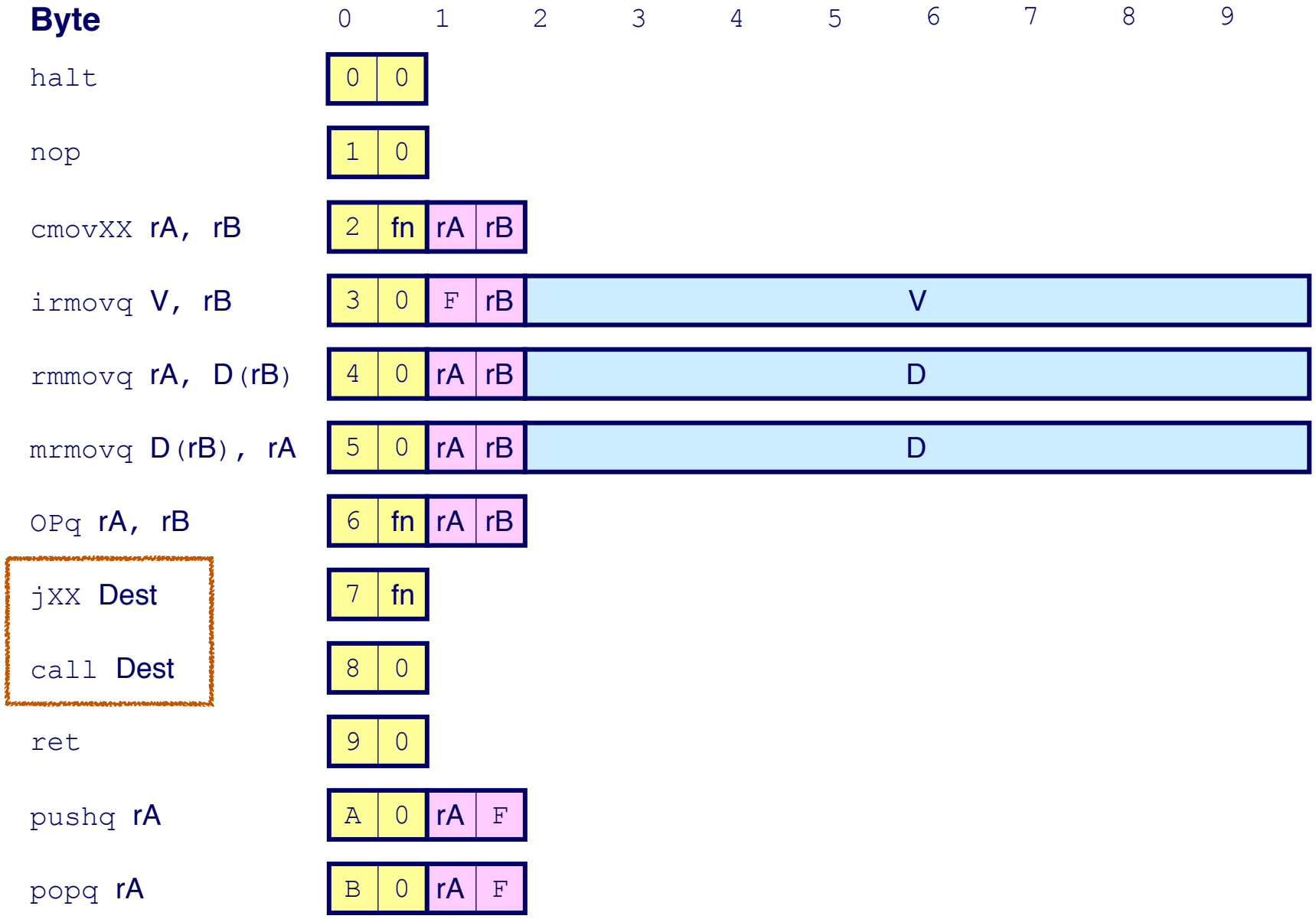
```
mrmovq -12(%rbp),%rcx
```

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

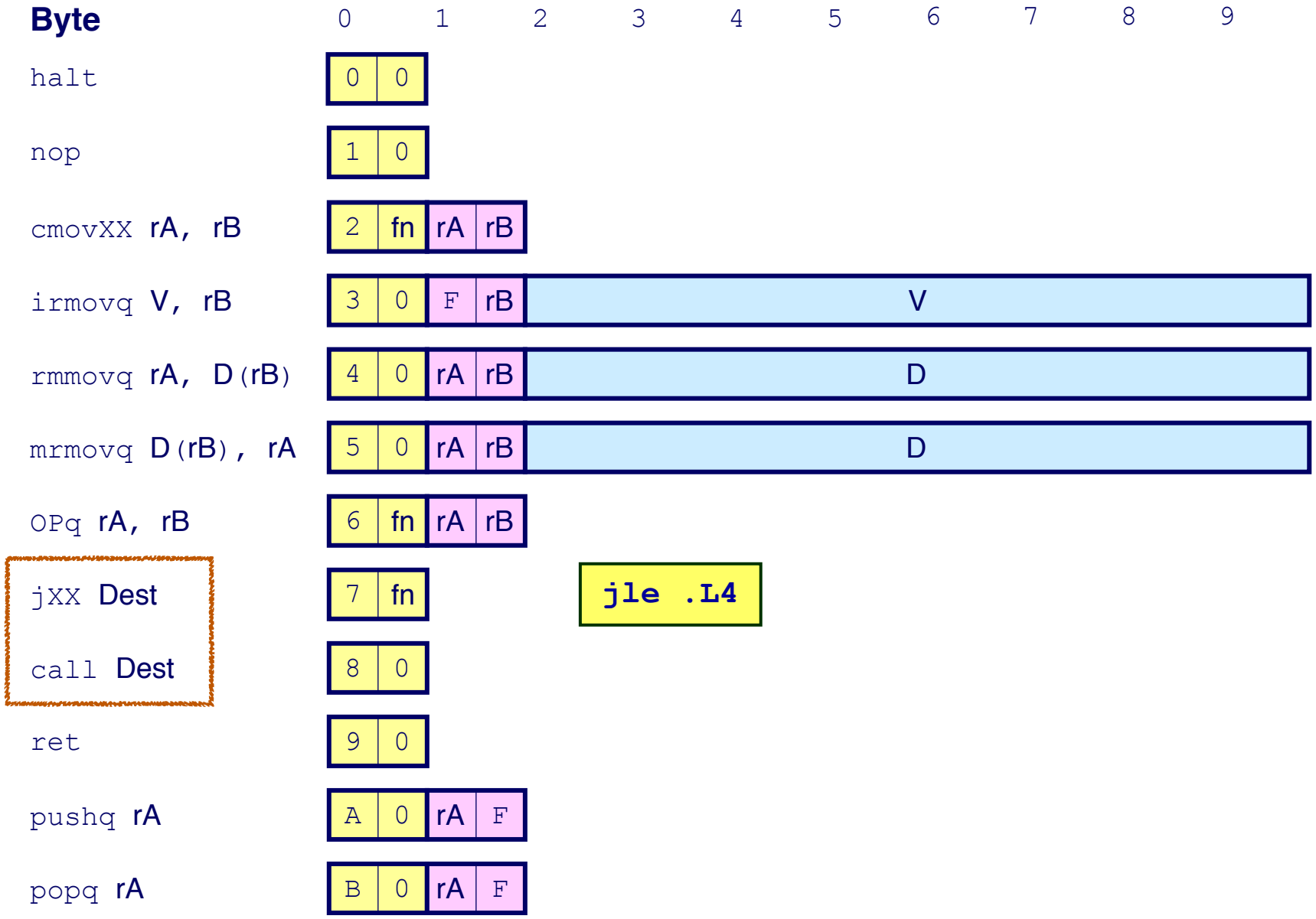
```
rmmovq %rsi,0x41c(%rsp)
```

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

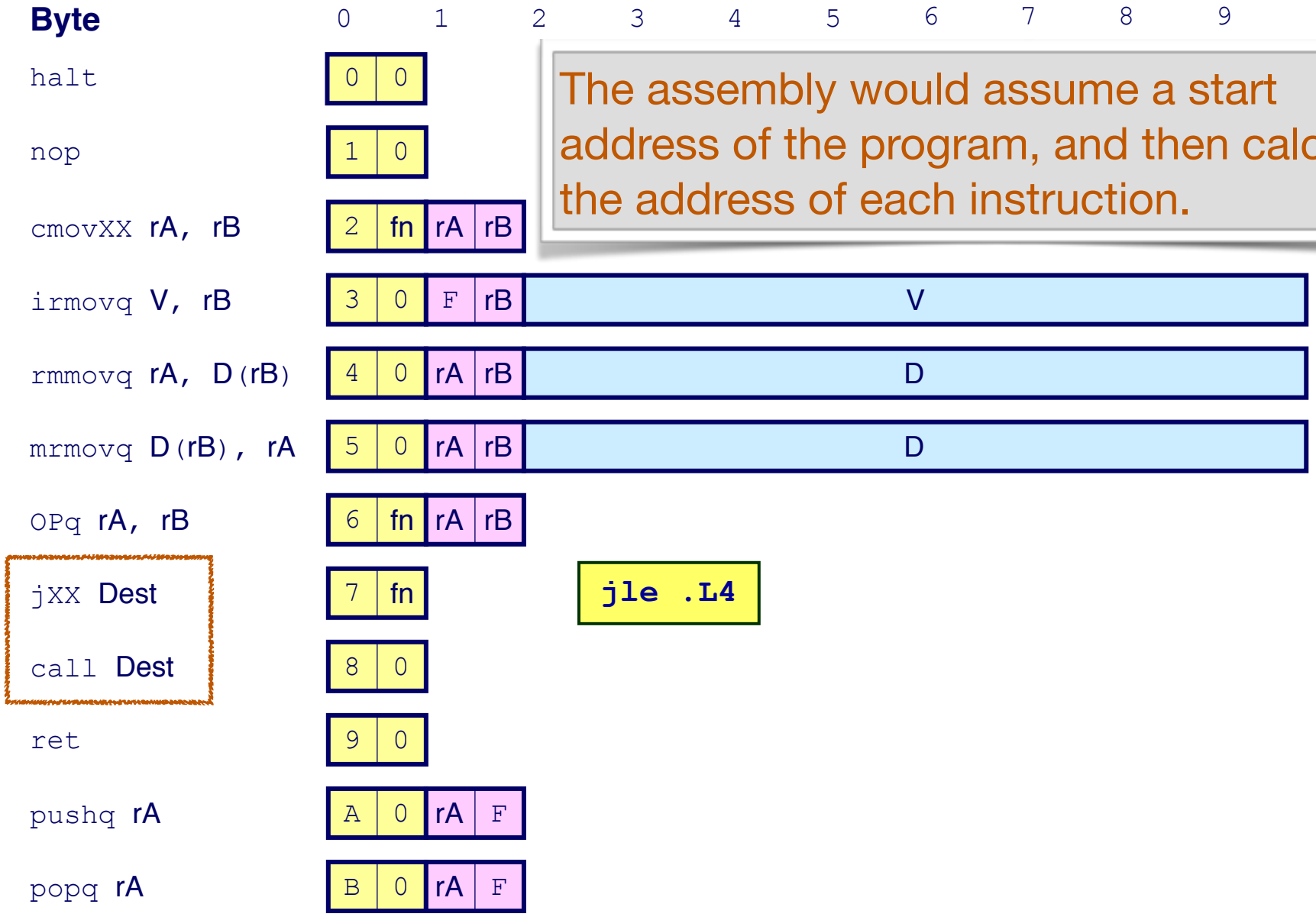
Jump/Call Instructions



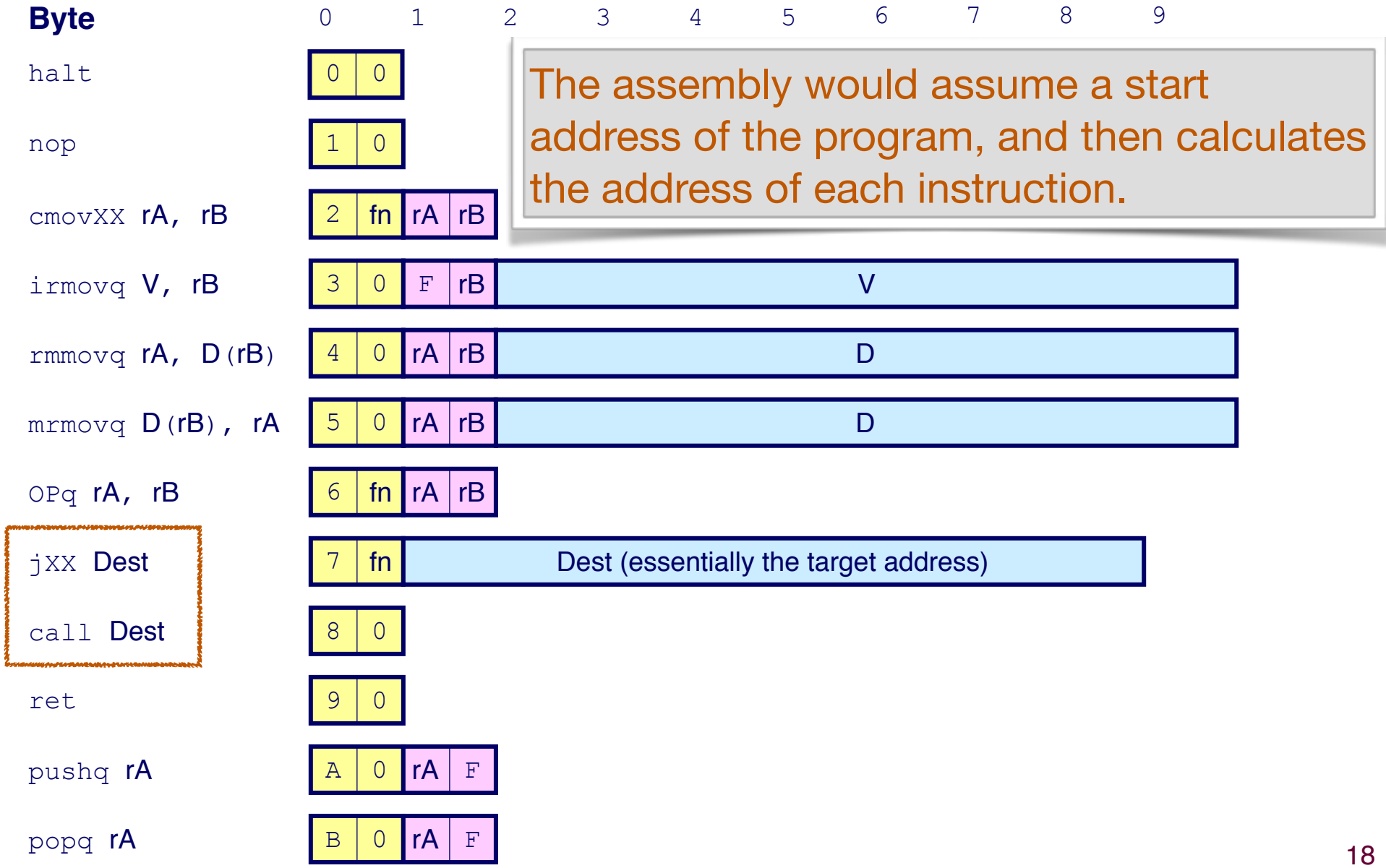
Jump/Call Instructions



Jump/Call Instructions

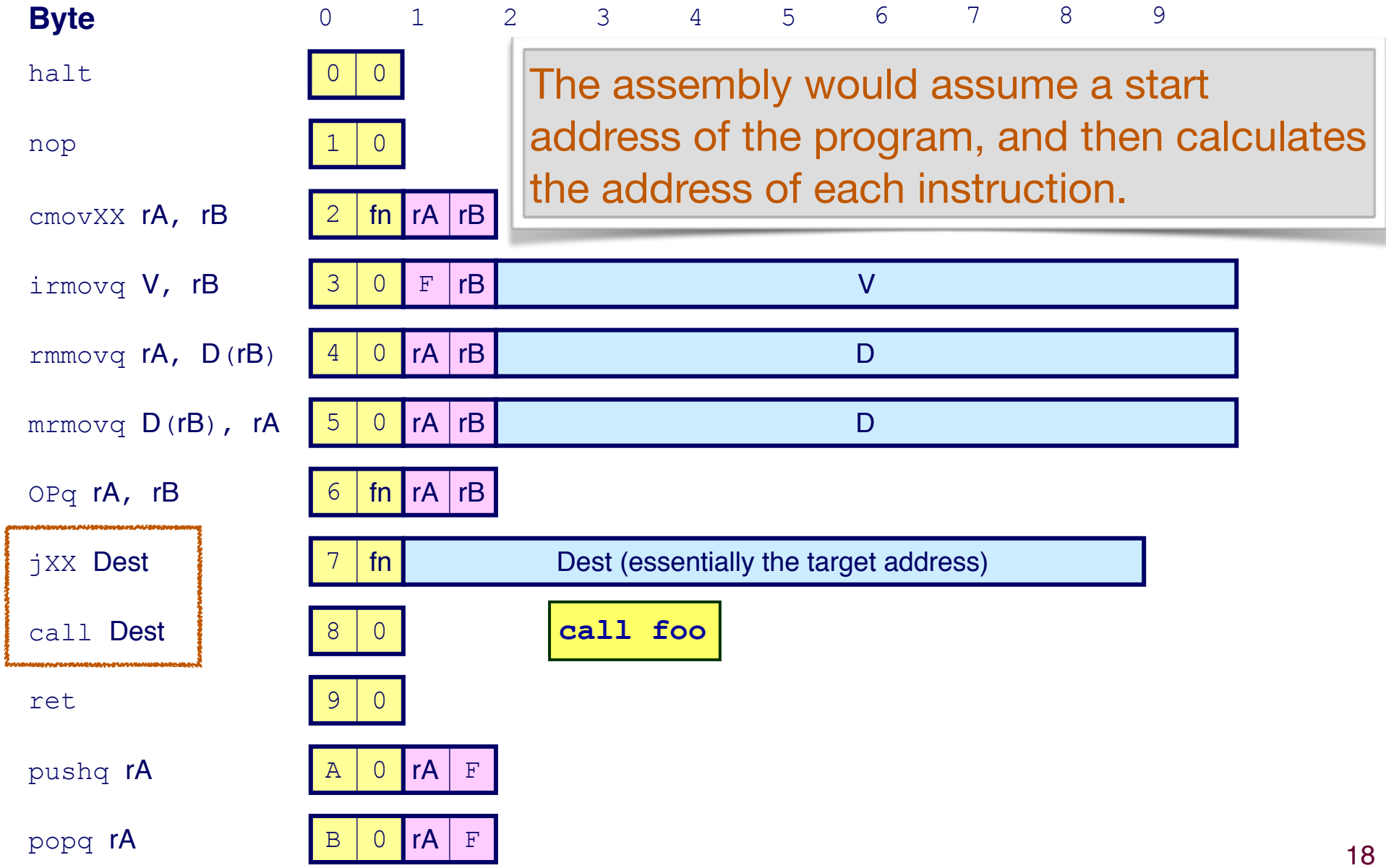


Jump/Call Instructions



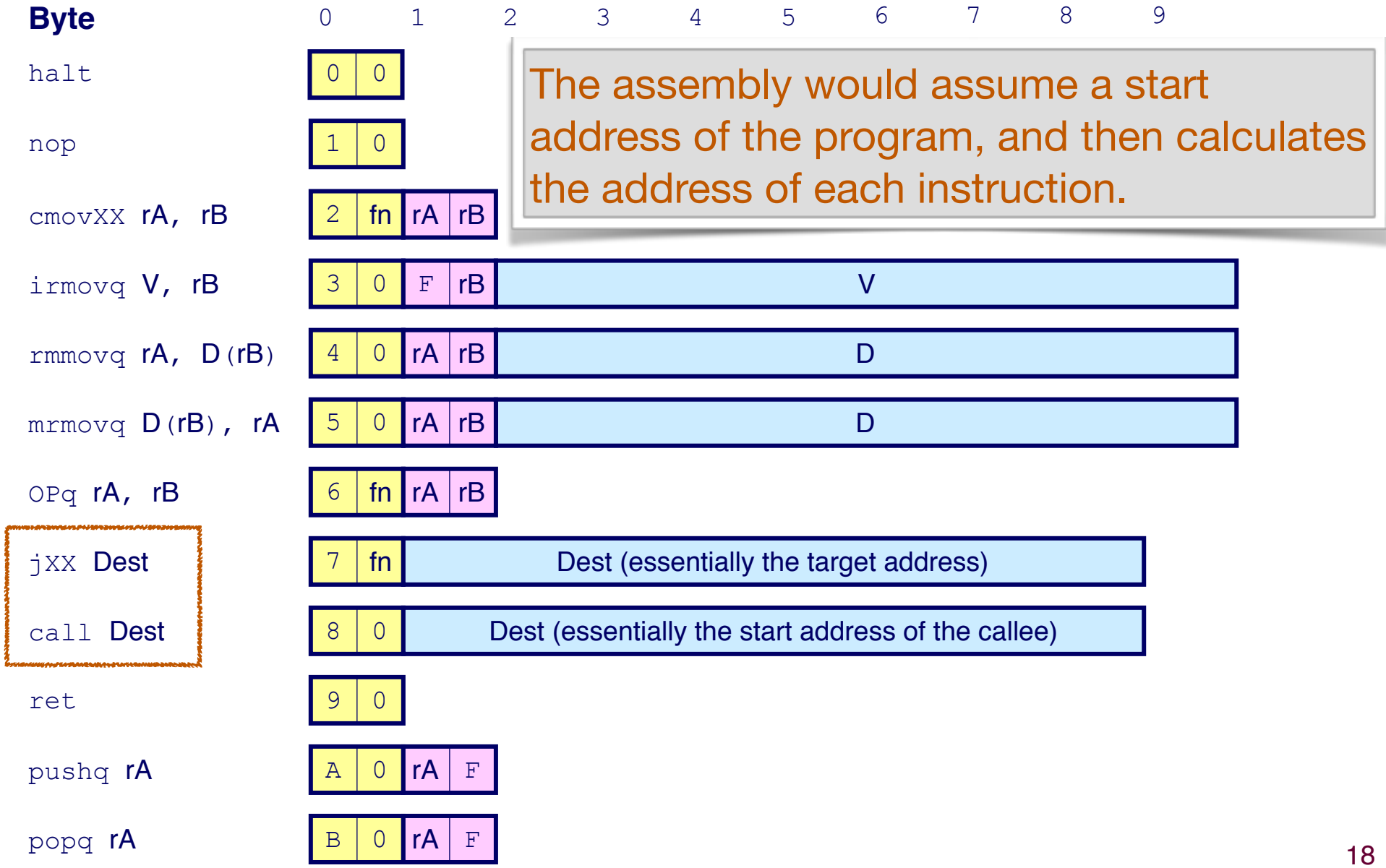
The assembly would assume a start address of the program, and then calculates the address of each instruction.

Jump/Call Instructions



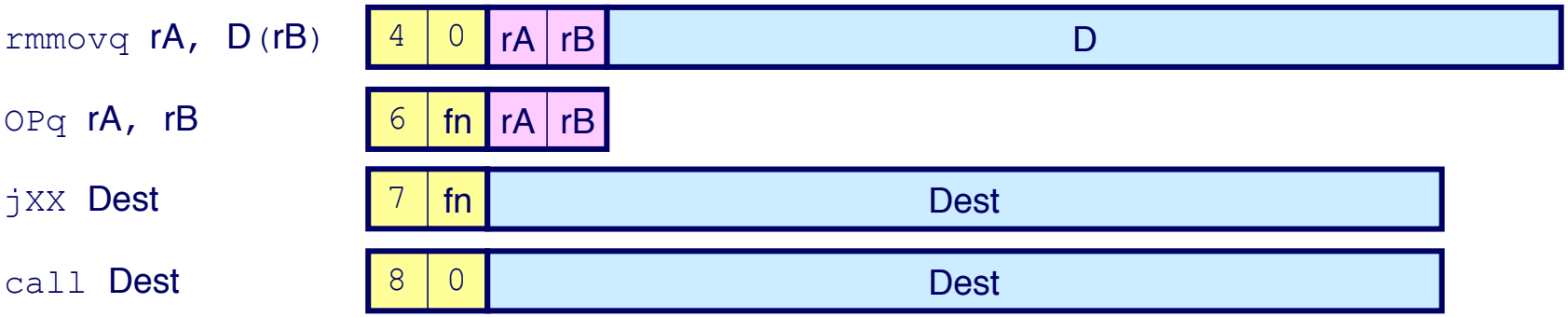
The assembly would assume a start address of the program, and then calculates the address of each instruction.

Jump/Call Instructions

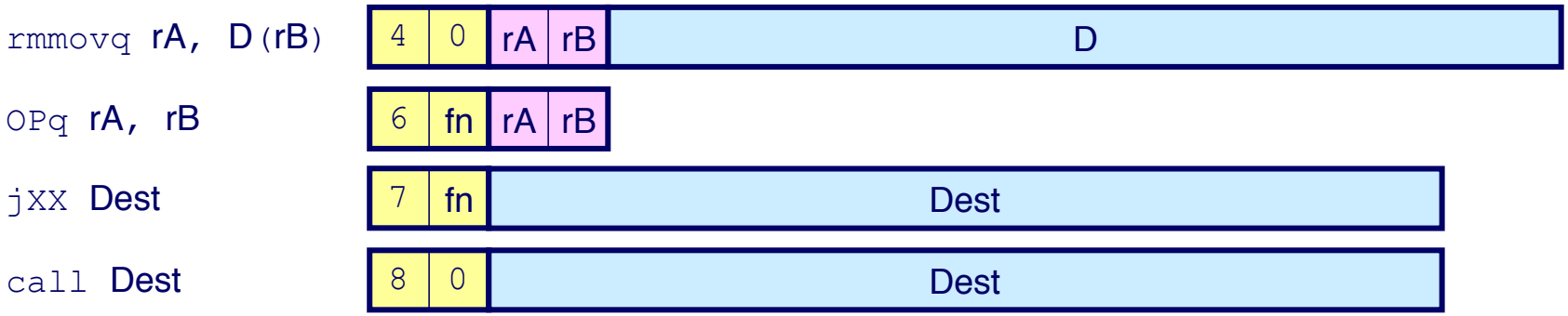


The assembly would assume a start address of the program, and then calculates the address of each instruction.

How Does An Assembler Work?

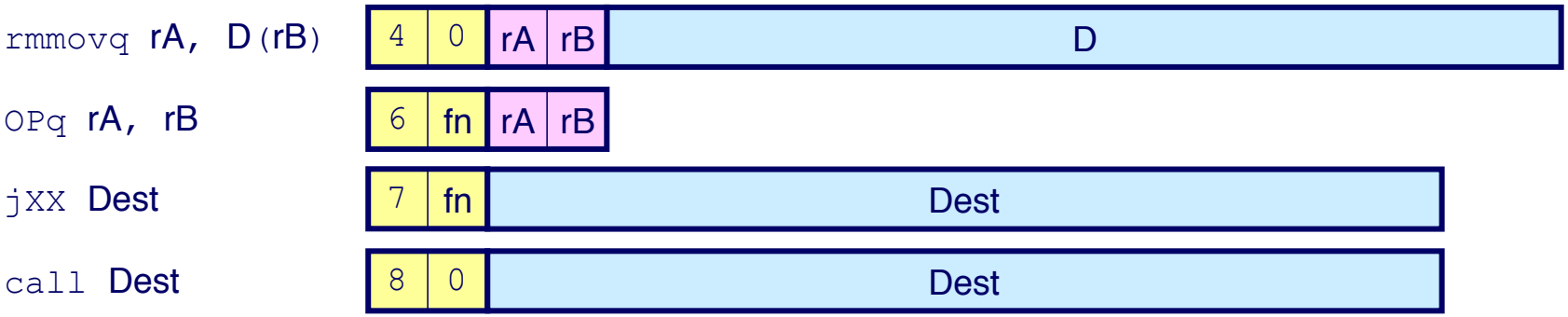


How Does An Assembler Work?



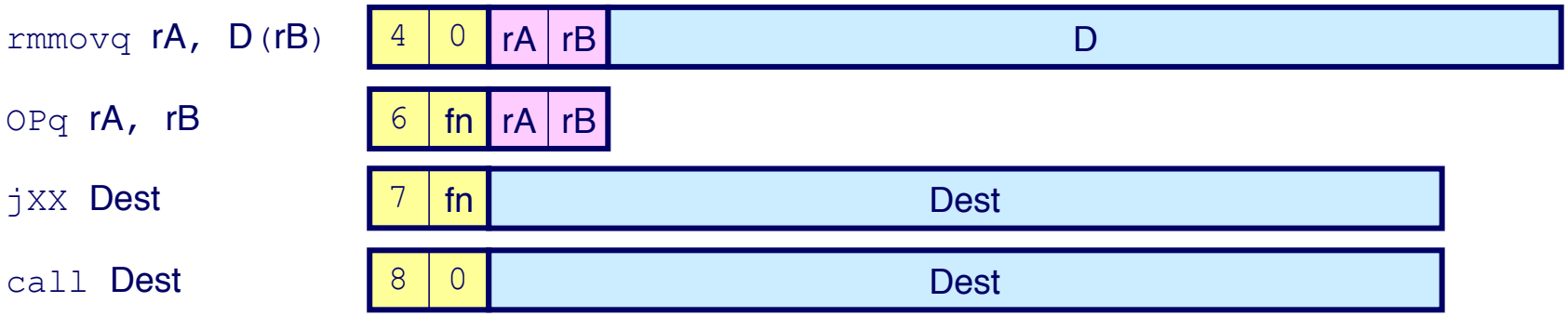

```
0x100    <foo>   rmmovq %rsi,0x41c(%rsp)
                 ... ...
                 ret
                 addq %rax,%rsi
                 call <foo>
                 jmp .L0
                 ... ...
         .L0     irmovq $0xabcd, %rdx
```

How Does An Assembler Work?



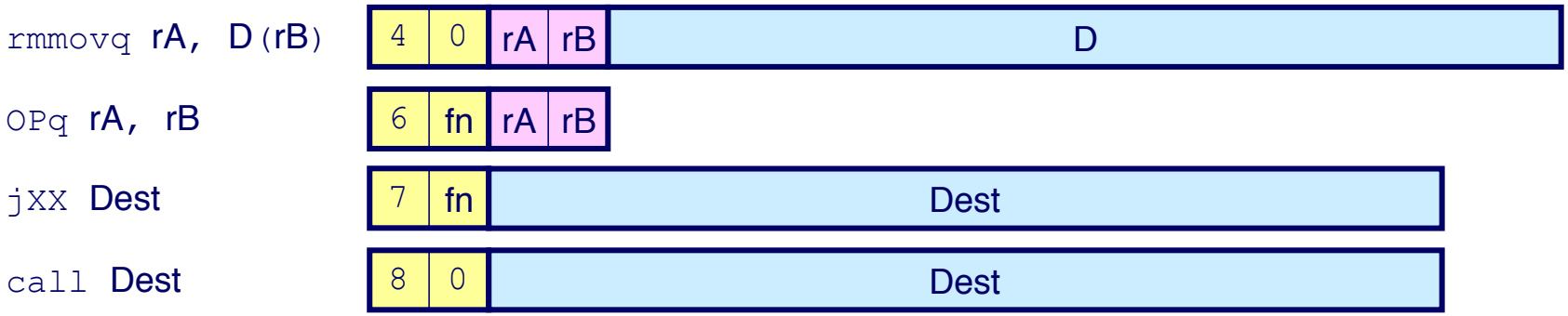

```
0x100    <foo>   rmmovq %rsi,0x41c(%rsp)    40 64 1c 04 00 00 00 00 00 00
                 ... ...
                 ret
                 addq %rax,%rsi
                 call <foo>
                 jmp .L0
                 ... ...
         .L0     irmovq $0xabcd, %rdx
```

How Does An Assembler Work?



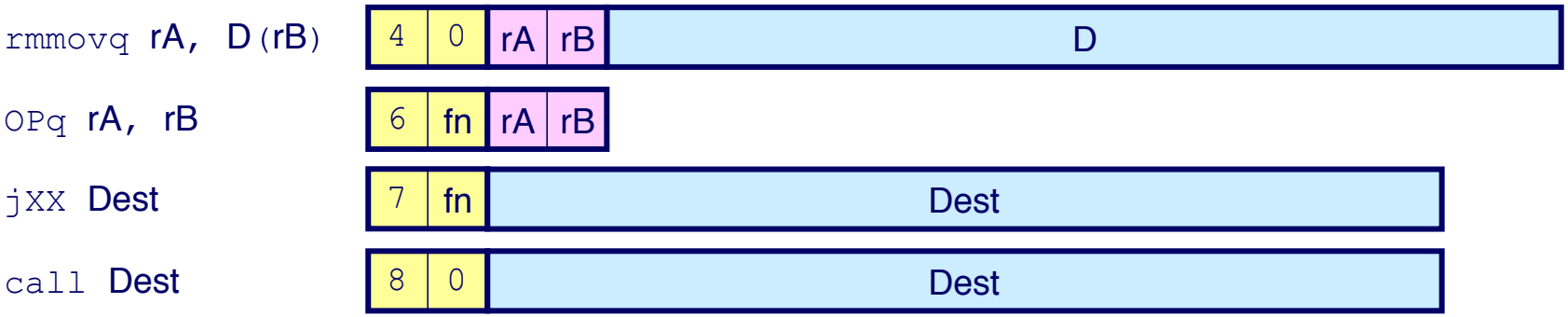

```
0x100    <foo>   rmmovq %rsi,0x41c(%rsp)    40 64 1c 04 00 00 00 00 00 00
                 ... ...
                 ret                       90
                 addq %rax,%rsi
                 call <foo>
                 jmp .L0
                 ... ...
             .L0 irmovq $0xabcd, %rdx
```

How Does An Assembler Work?



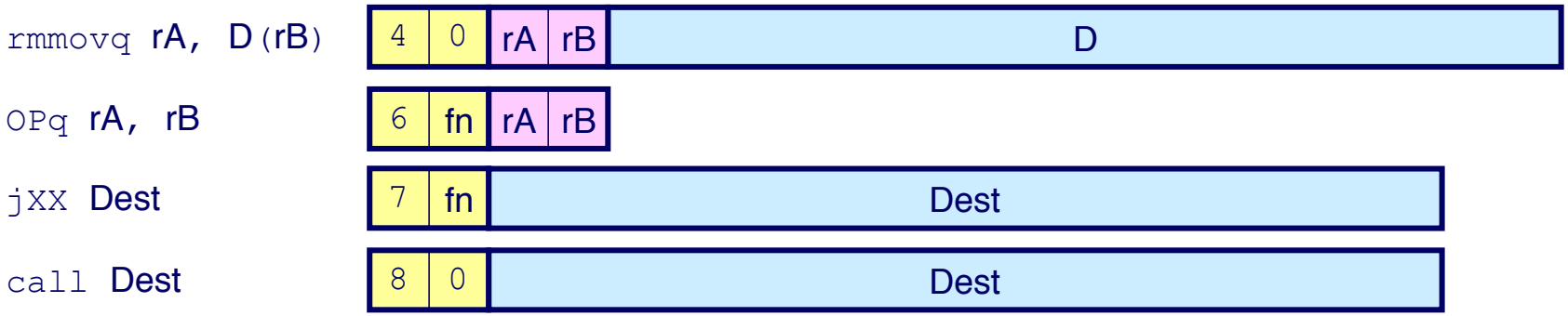

```
0x100    <foo>   rmmovq %rsi,0x41c(%rsp)    40 64 1c 04 00 00 00 00 00 00
                 ... ...
                 ret                       90
                 addq %rax,%rsi            60 06
                 call <foo>
                 jmp .L0
                 ... ...
         .L0     irmovq $0xabcd,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

How Does An Assembler Work?



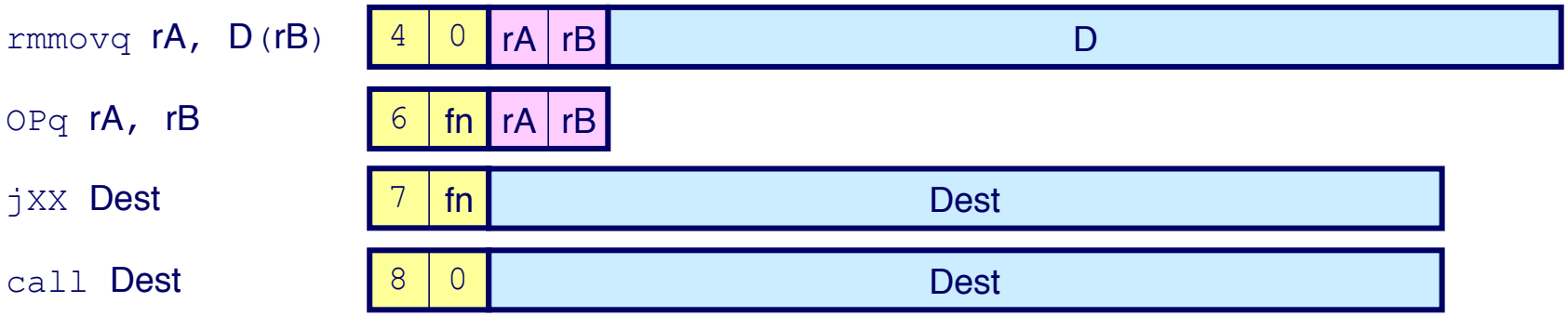

```
0x100    <foo>   rmmovq %rsi,0x41c(%rsp)     40 64 1c 04 00 00 00 00 00 00
                 ... ...
                 ret                        90
                 addq %rax,%rsi             60 06
                 call <foo>                 80 00 01 00 00 00 00 00 00
                 jmp .L0
   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

How Does An Assembler Work?



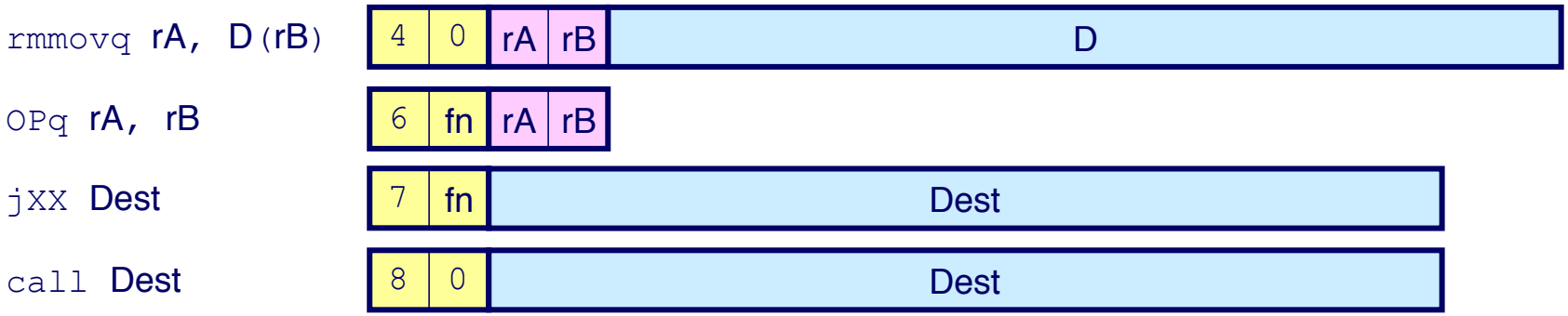

```
0x100    <foo>   rmmovq %rsi,0x41c(%rsp)    40 64 1c 04 00 00 00 00 00 00
                 ... ...
                 ret                       90
                 addq %rax,%rsi            60 06
                 call <foo>                80 00 01 00 00 00 00 00 00
                 jmp .L0        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

How Does An Assembler Work?




```
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
```

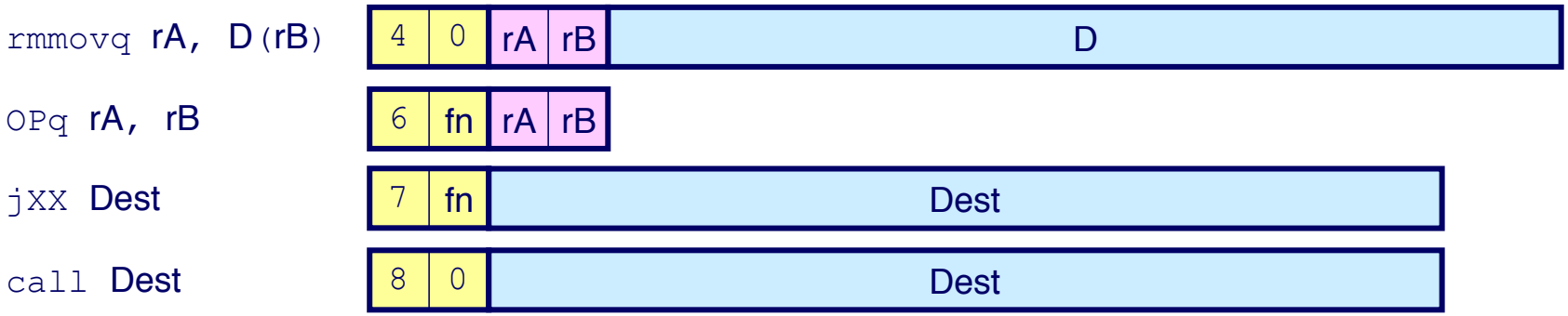
How Does An Assembler Work?




```
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

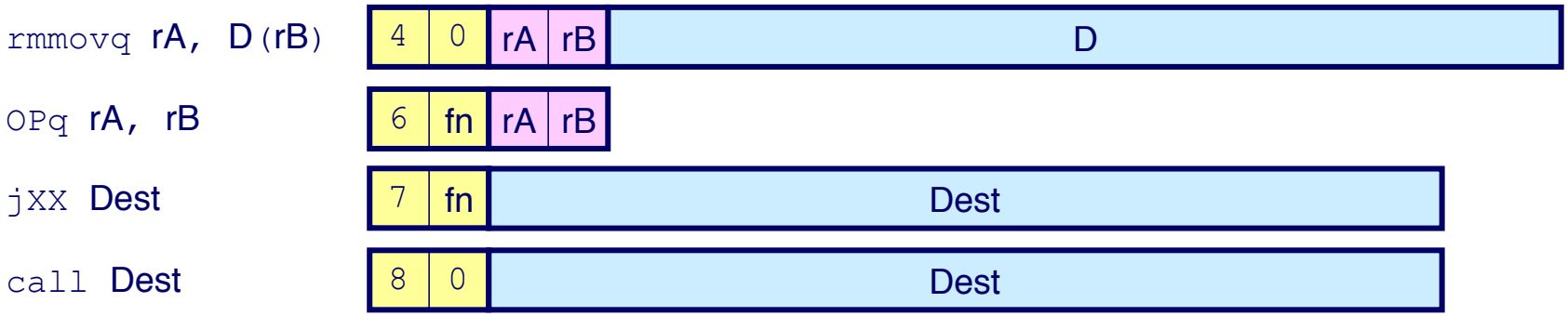
How Does An Assembler Work?




```
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?

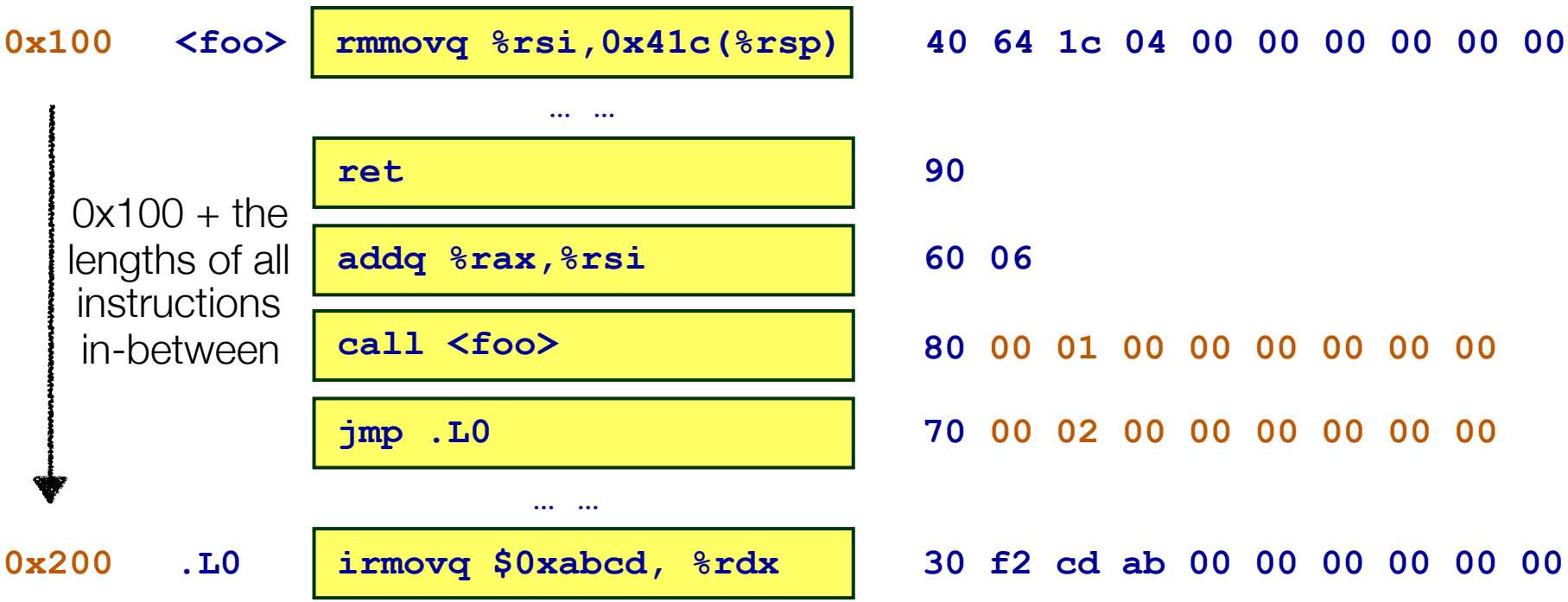



```
0x100    <foo>   rmmovq %rsi,0x41c(%rsp)     40 64 1c 04 00 00 00 00 00 00
                 ... ...
                 ret                        90
                 addq %rax,%rsi             60 06
                 call <foo>                 80 00 01 00 00 00 00 00 00
                 jmp .L0    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



Jump Instructions

Jump Unconditionally

Jump When Less or Equal

Jump When Less

Jump When Equal

Jump When Not Equal

Jump When Greater or Equal

Jump When Greater

Subroutine Call and Return

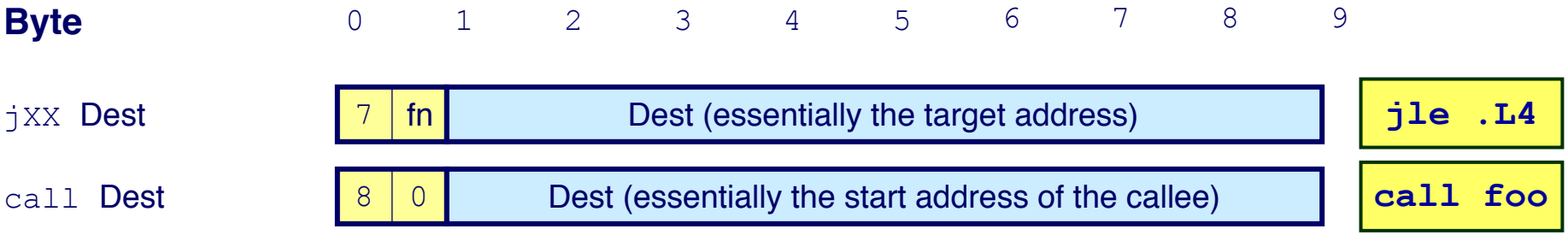
`call` Dest

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

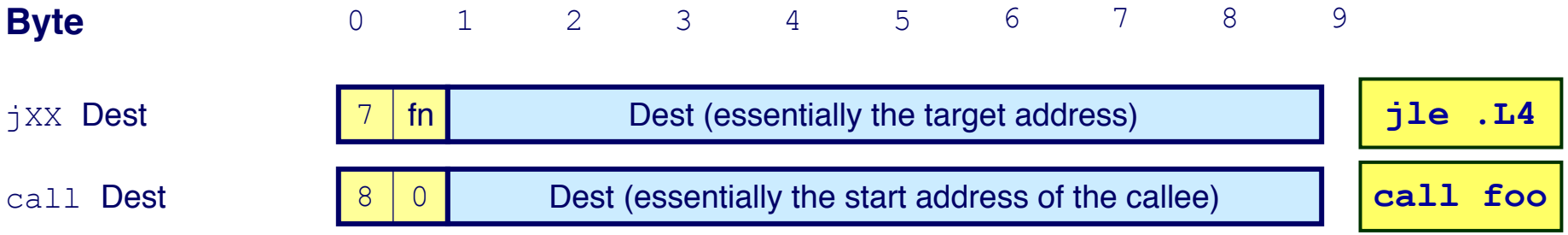
`ret`

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

One More Complication…

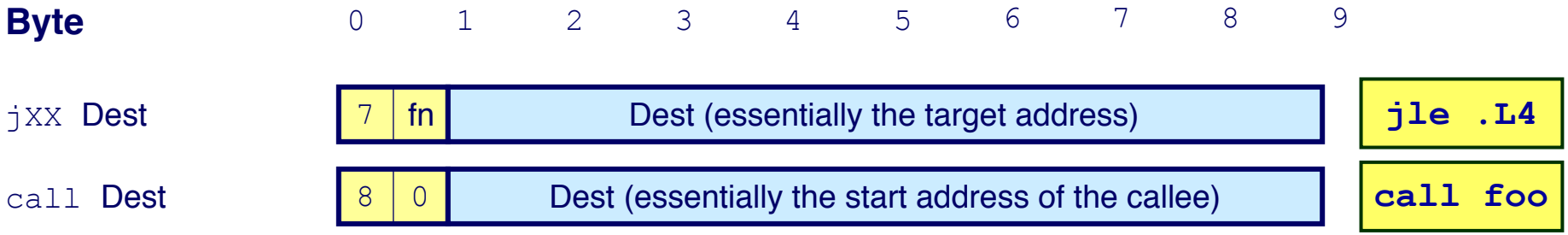


One More Complication…



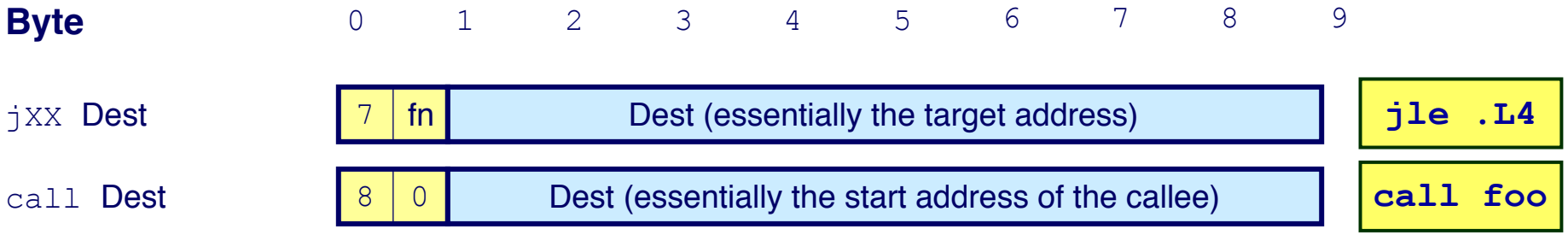
- The instruction length limits how far you can jump/call functions. What if the jump target has a very long address that can't fit in 8 bytes?
  - Or if we can use only say 4 bytes for the target address?

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?
  - Or if we can use only say 4 bytes for the target address?
- One alternative: use a super long instruction encoding format.
  - Simple to encode, but space inefficient (waste bits for jumps to short addr.)

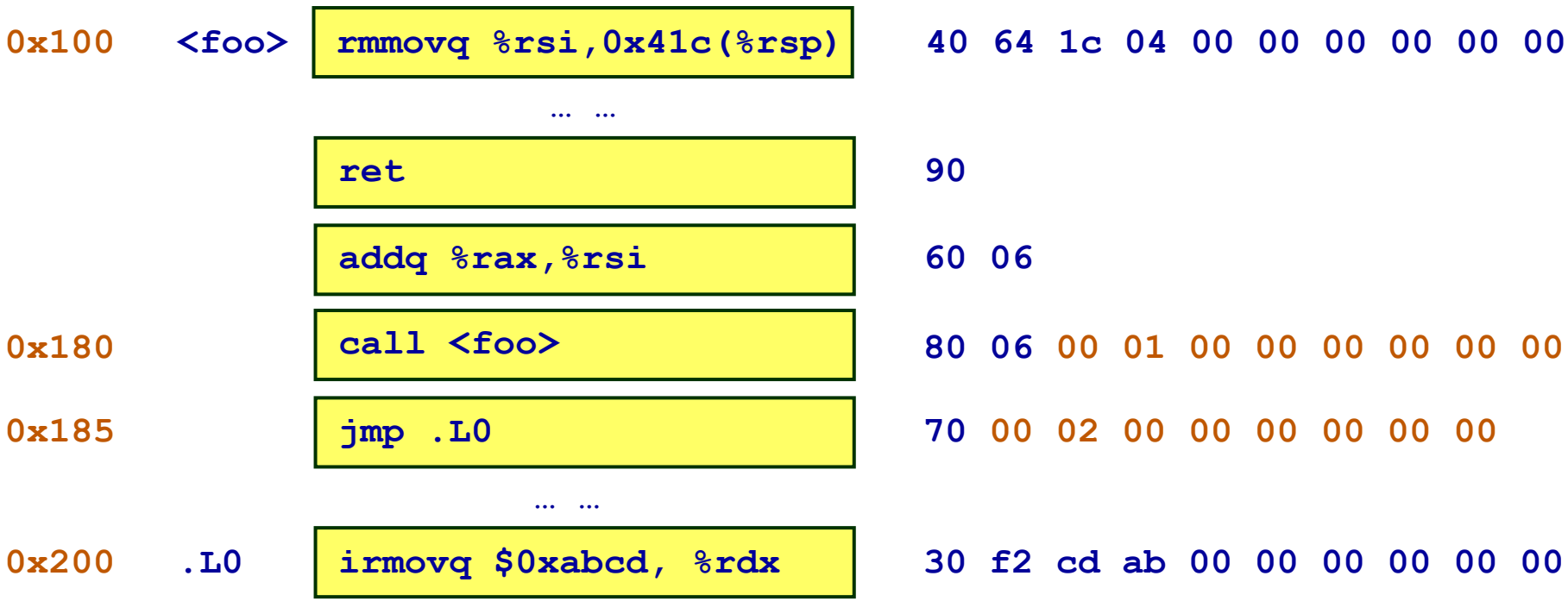
One More Complication…



- 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?
  - Or if we can use only say 4 bytes for the target address?
- 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

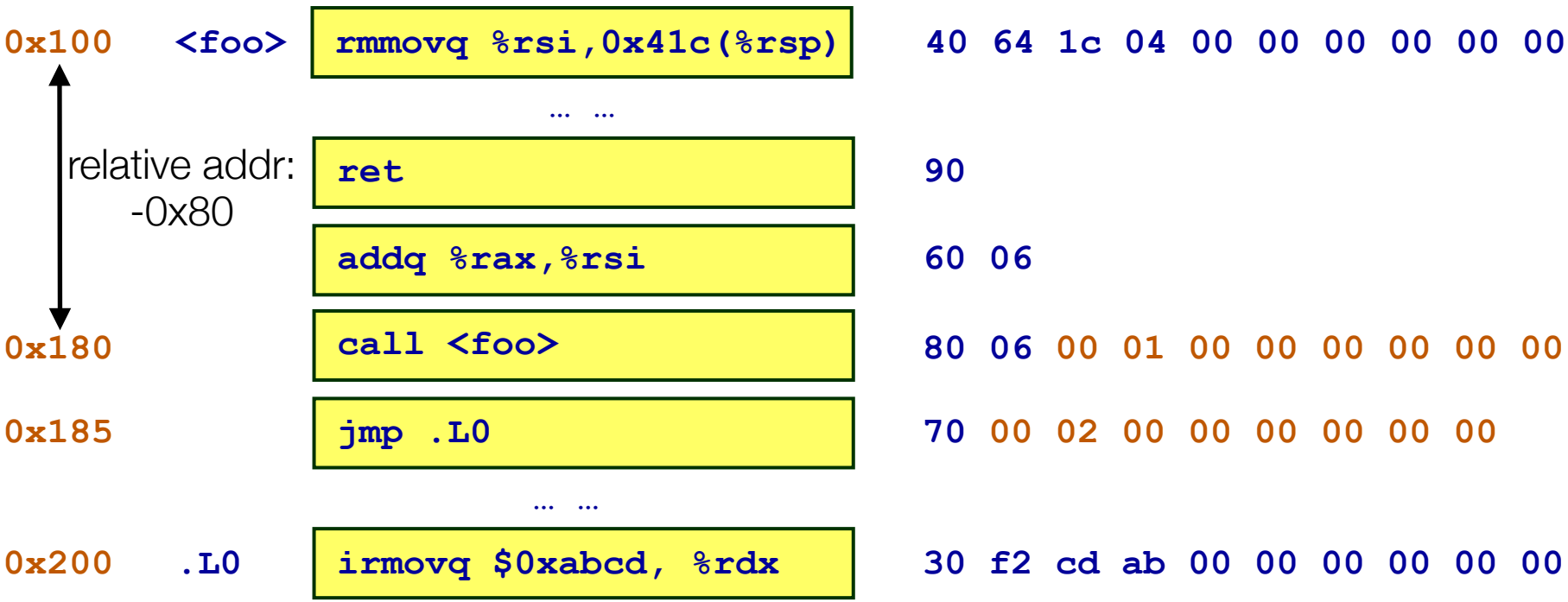
Using Relative Addresses for Jumps

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



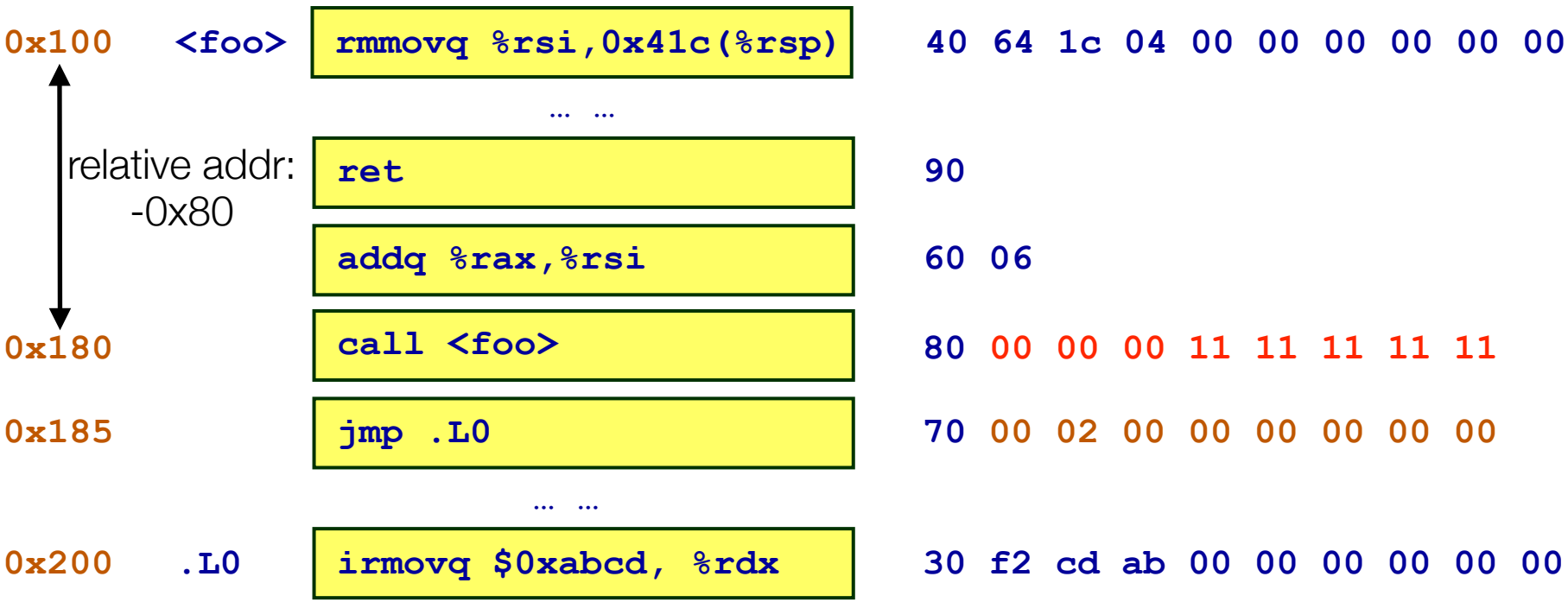
Using Relative Addresses for Jumps

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



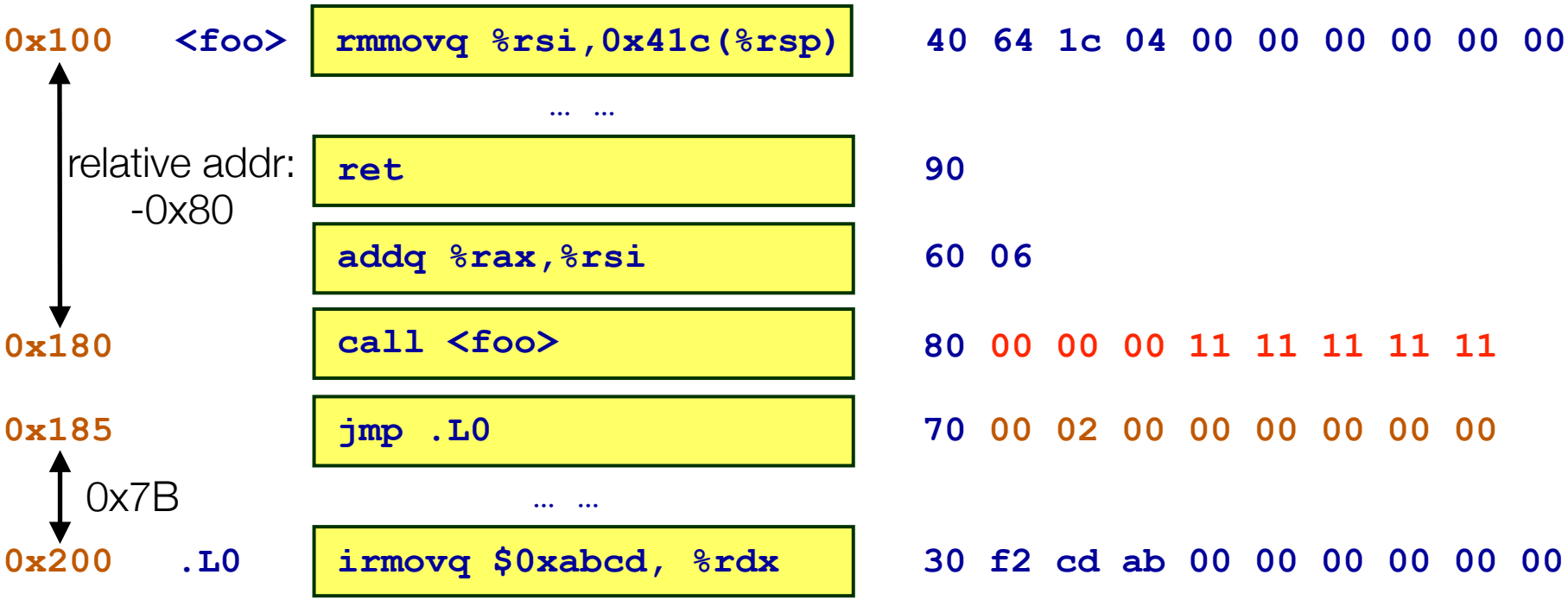
Using Relative Addresses for Jumps

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



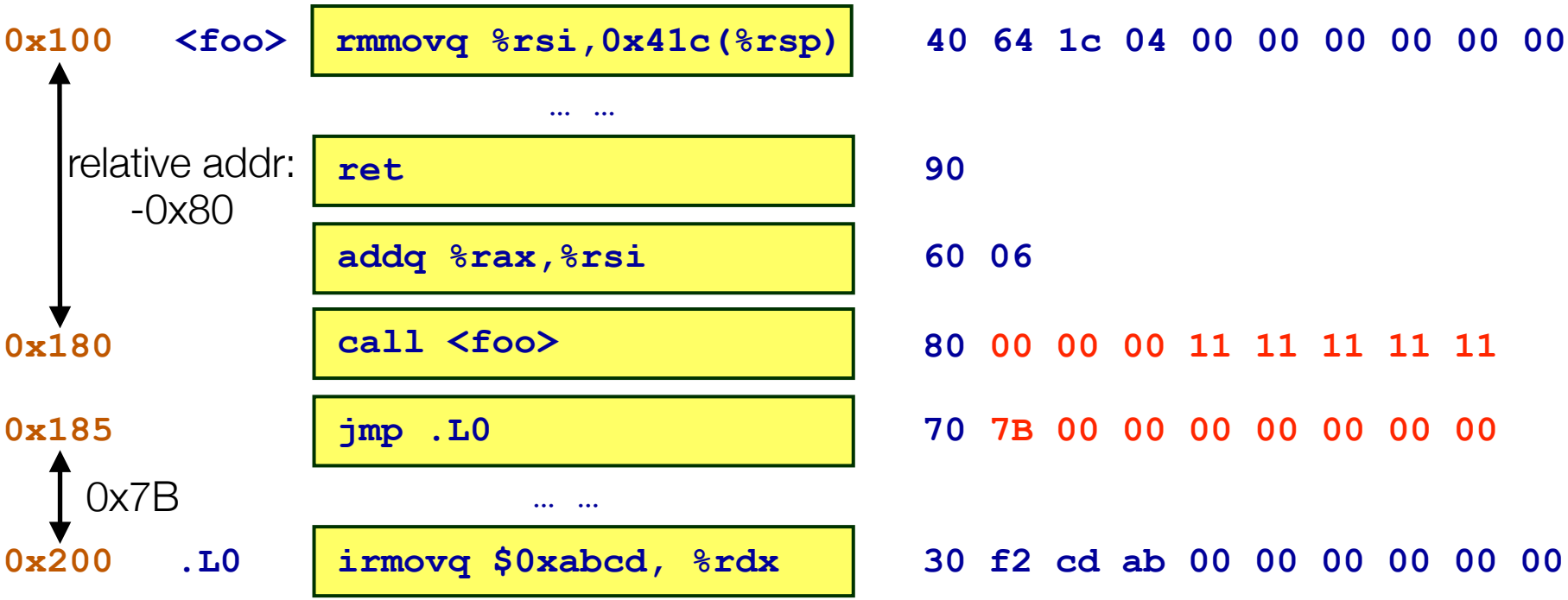
Using Relative Addresses for Jumps

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



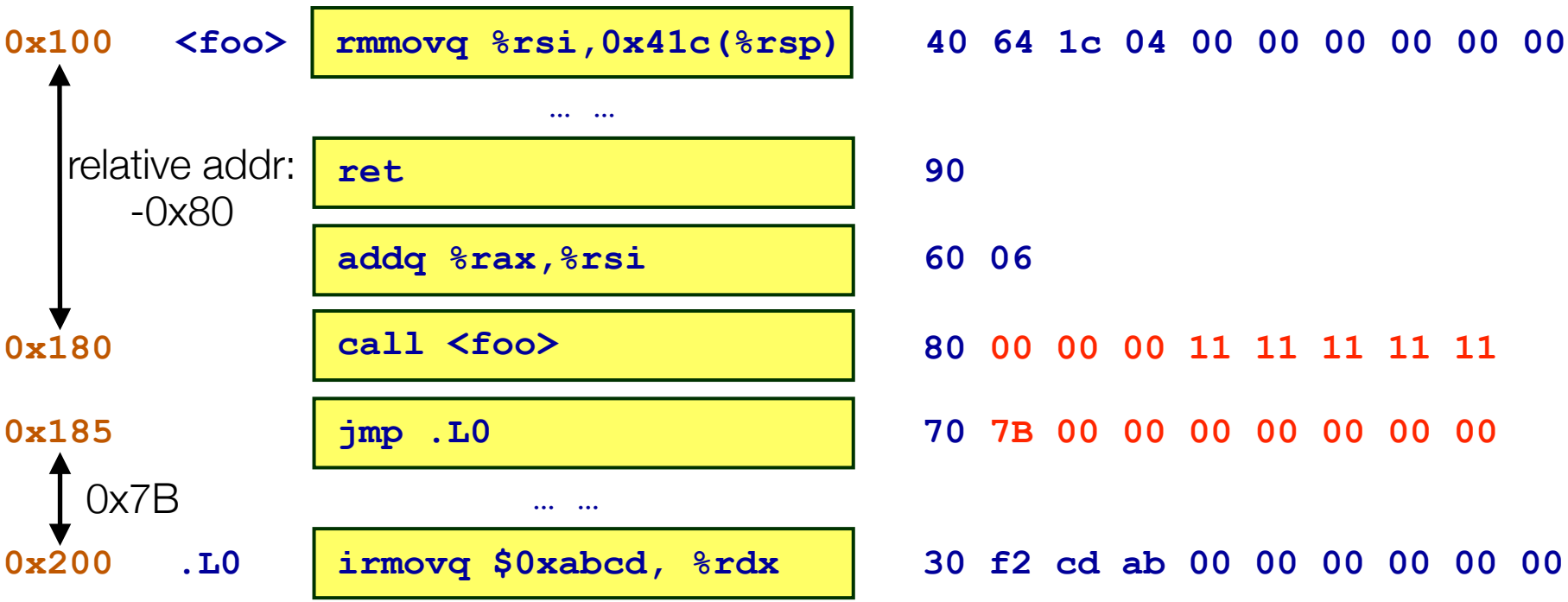
Using Relative Addresses for Jumps

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



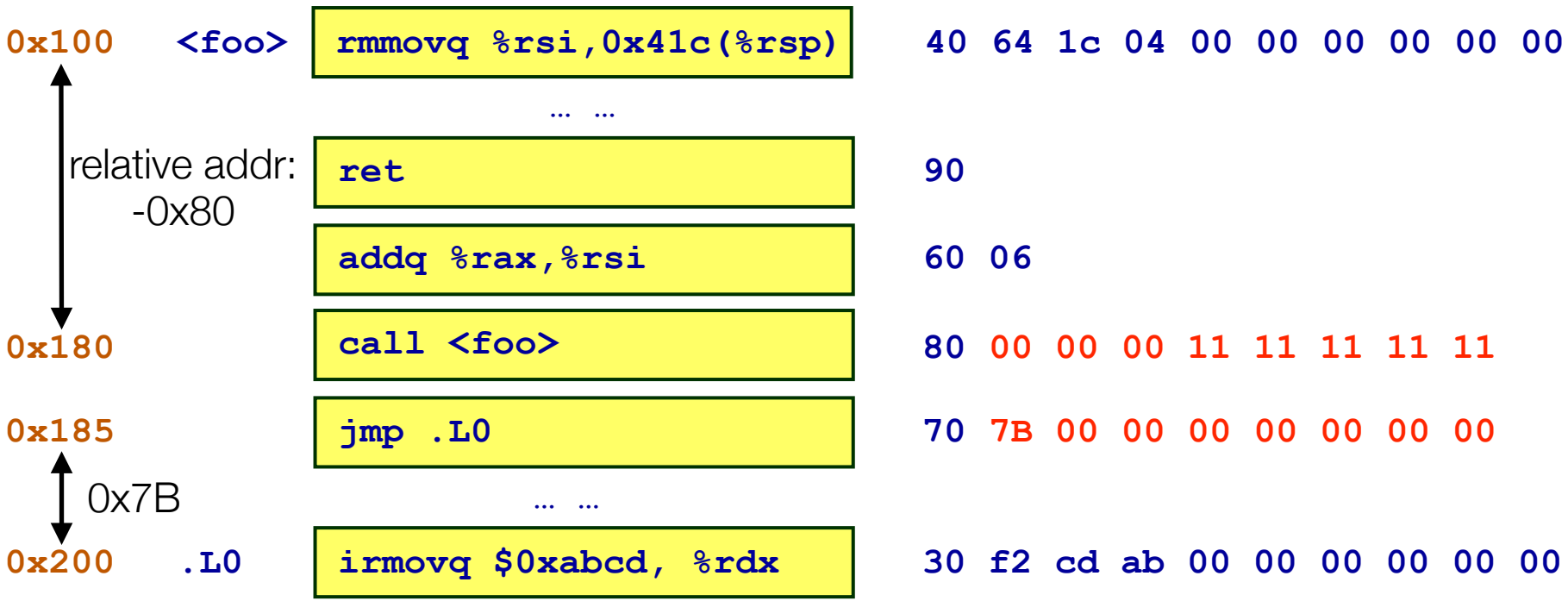
Using Relative Addresses for Jumps

- 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?



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

- 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

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?

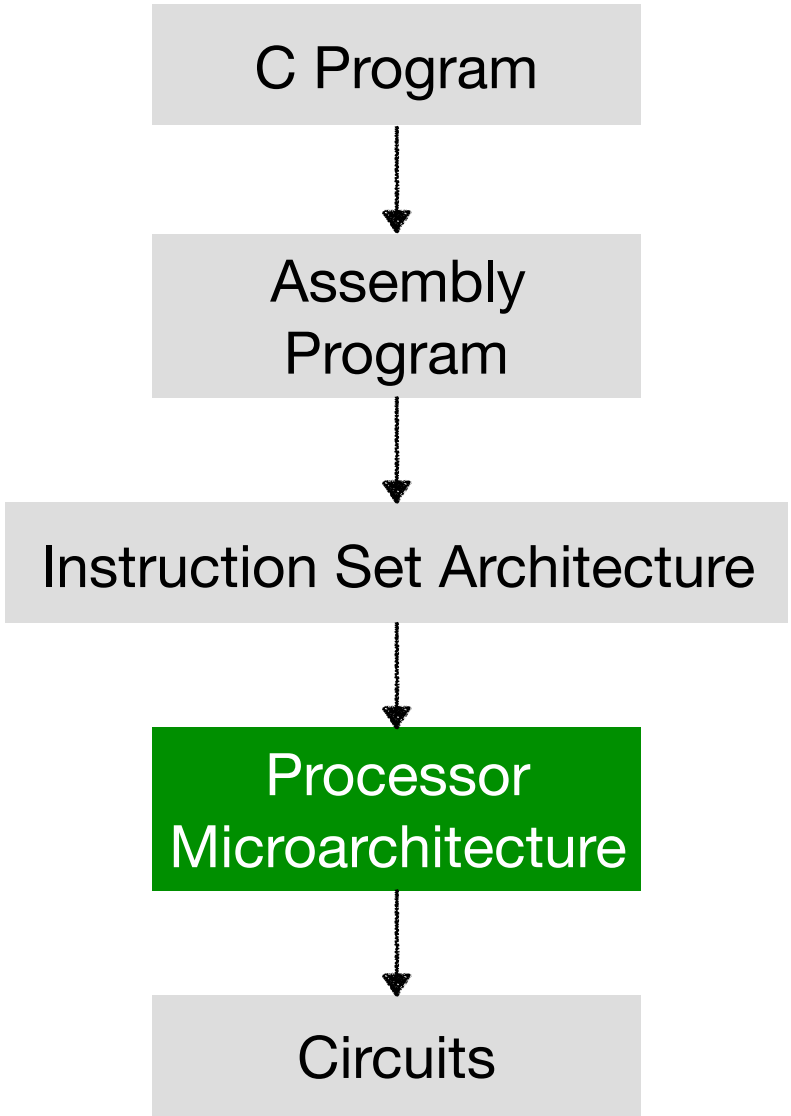
Variable Length Instructions

- 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…

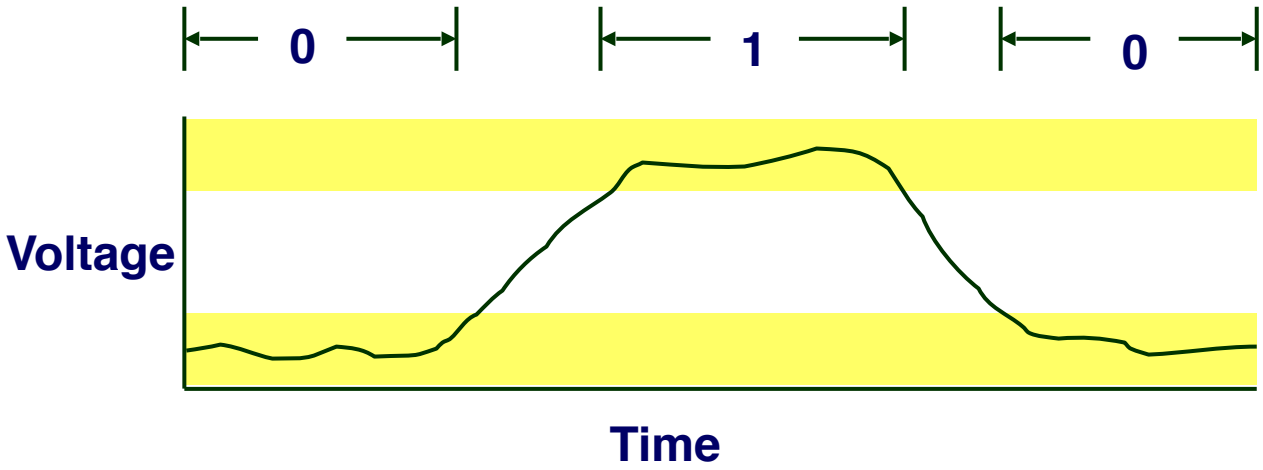

C Program
Assembly Program
Instruction Set Architecture
Processor Microarchitecture
Circuits

Today: Circuits Basics

- Basics
- Circuits for computations
- Circuits for storing data

Overview of Circuit-Level Design

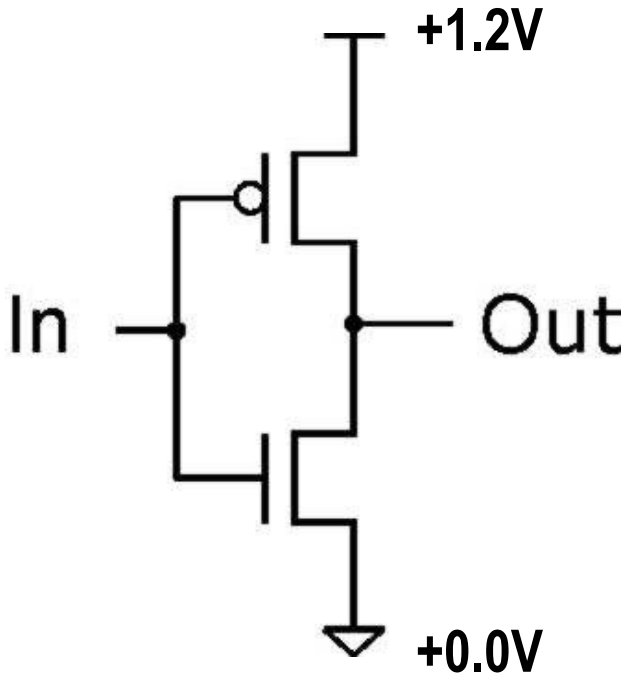
- Fundamental Hardware Requirements
  - Communication: How to get values from one place to another. Mainly three electrical **wires**.
  - Computation: **transistors**. Combinational logic.
  - Storage: **transistors**. Sequential logic.
- Circuit design is often abstracted as logic design



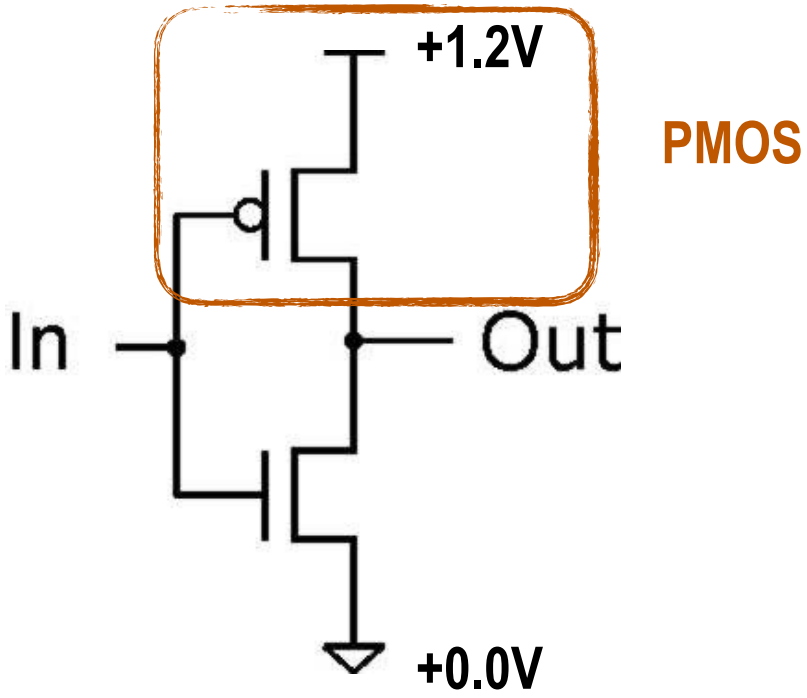
Today: Circuits Basics

- Transistors
- Circuits for computations
- Circuits for storing data

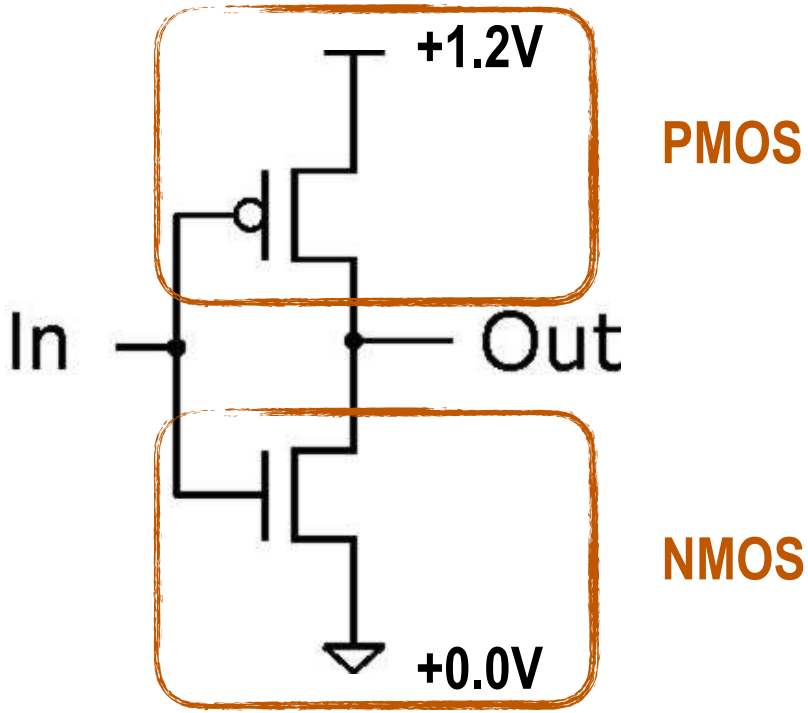
Inverter (NOT Gate)



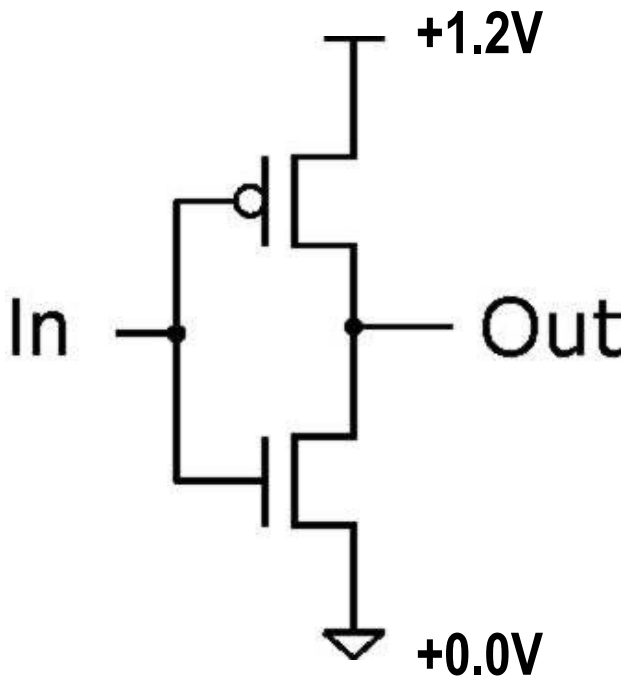
Inverter (NOT Gate)

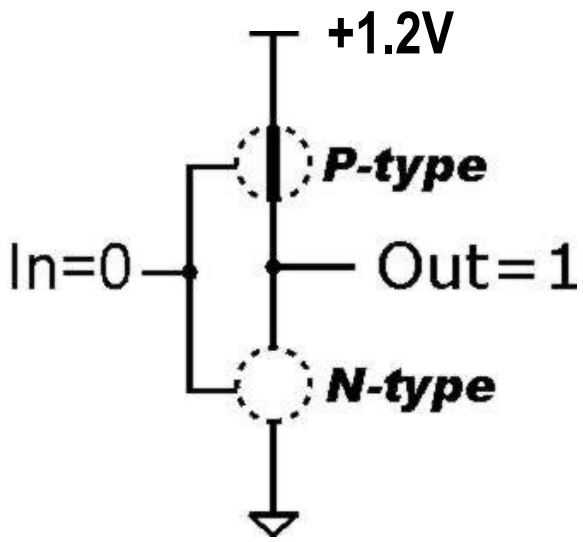


Inverter (NOT Gate)

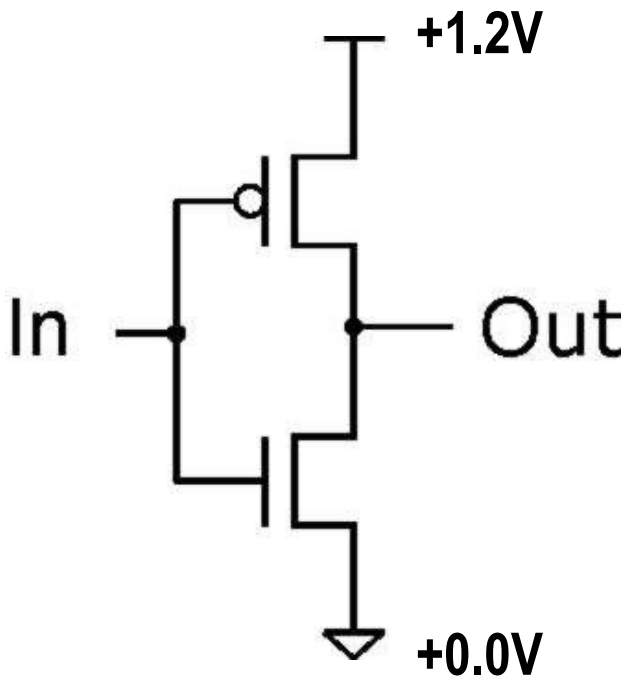


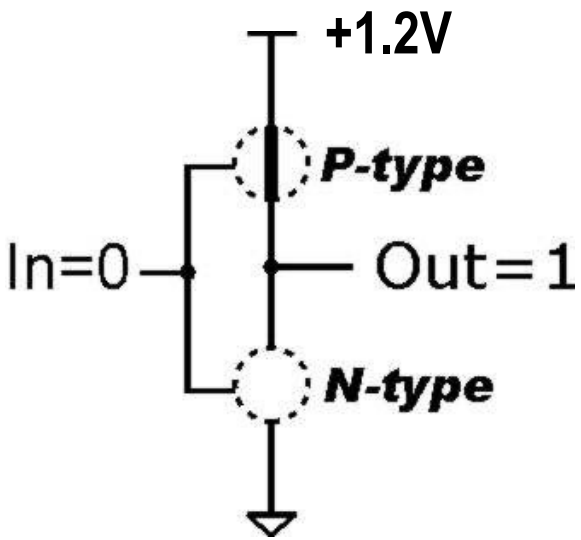
Inverter (NOT Gate)

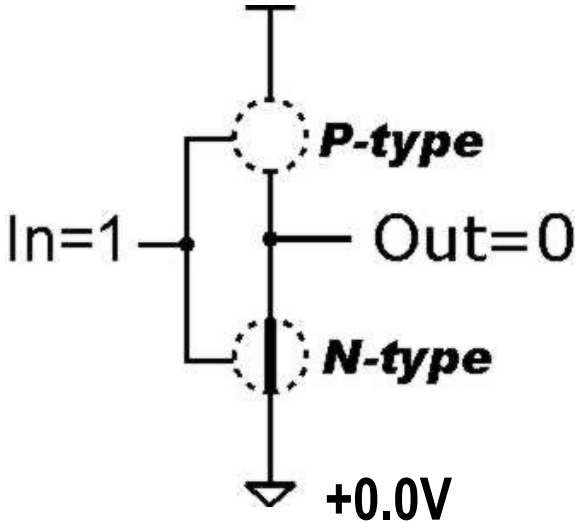




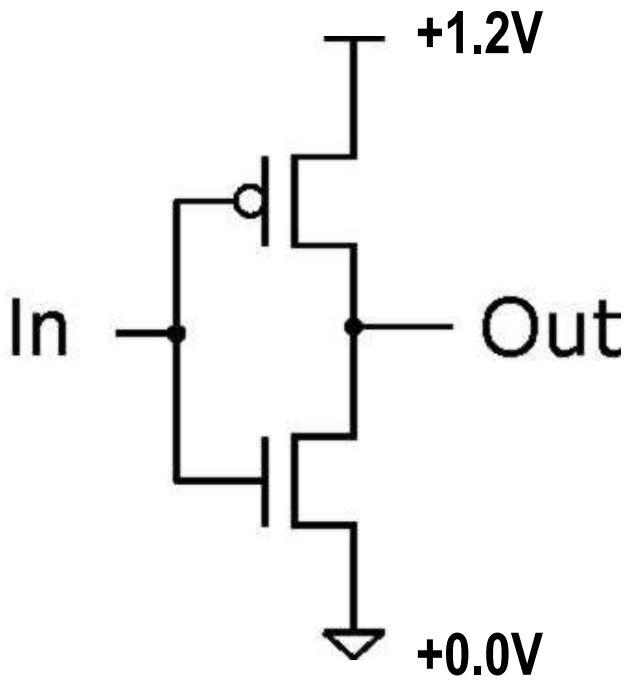
Inverter (NOT Gate)



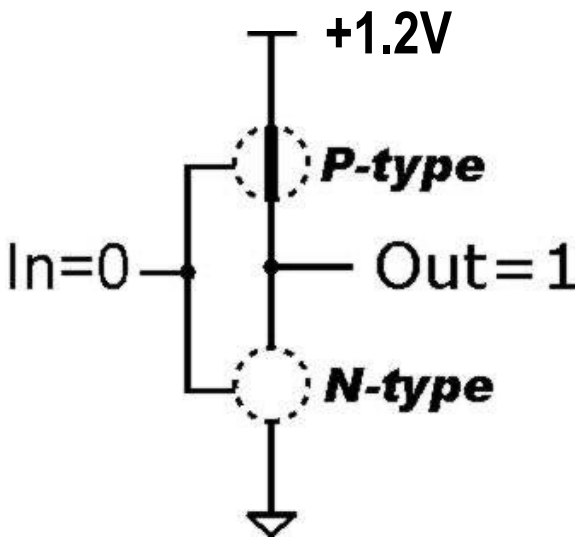


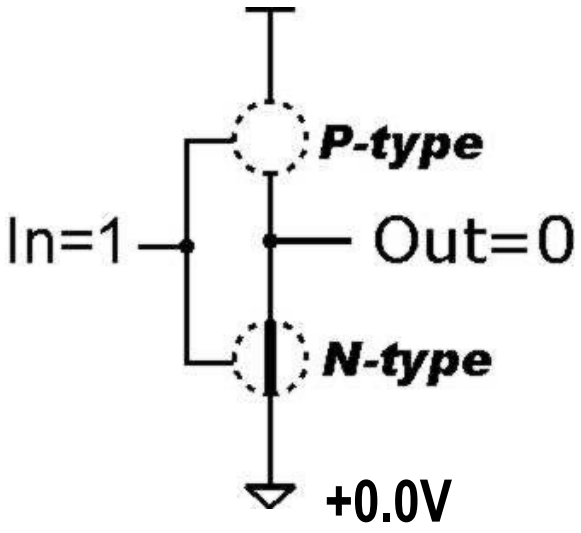


Inverter (NOT Gate)

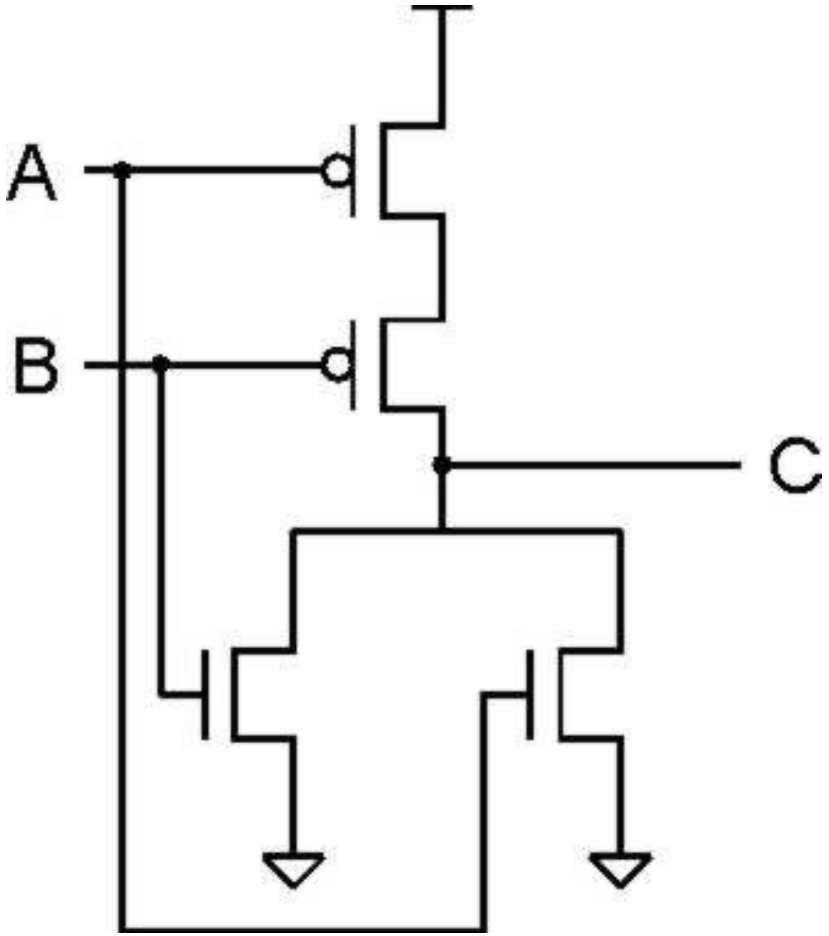


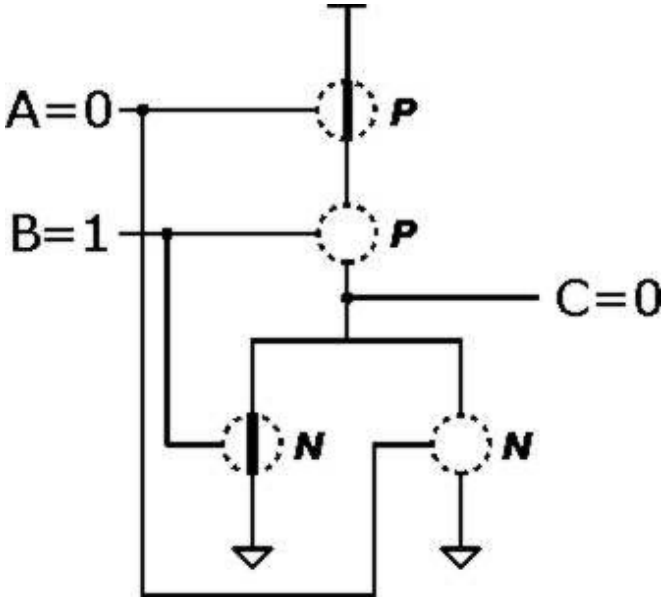
In	Out
0	1
1	0





NOR Gate (NOT + OR)

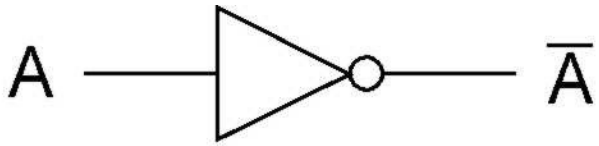




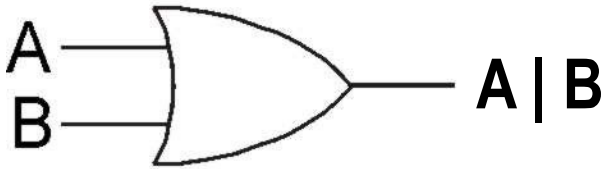
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

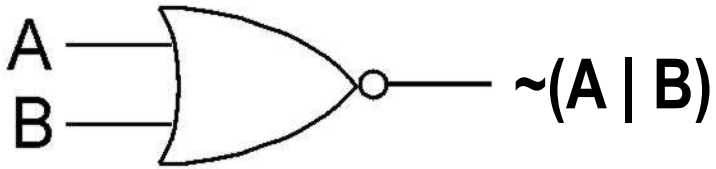
Basic Logic Gates



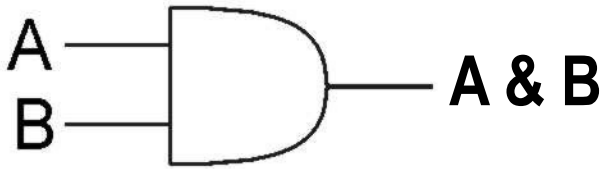
NOT



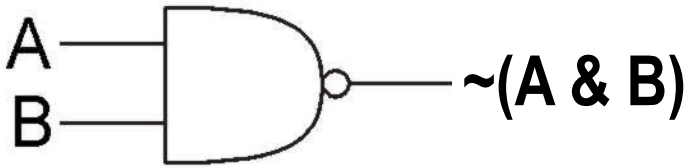
OR



NOR

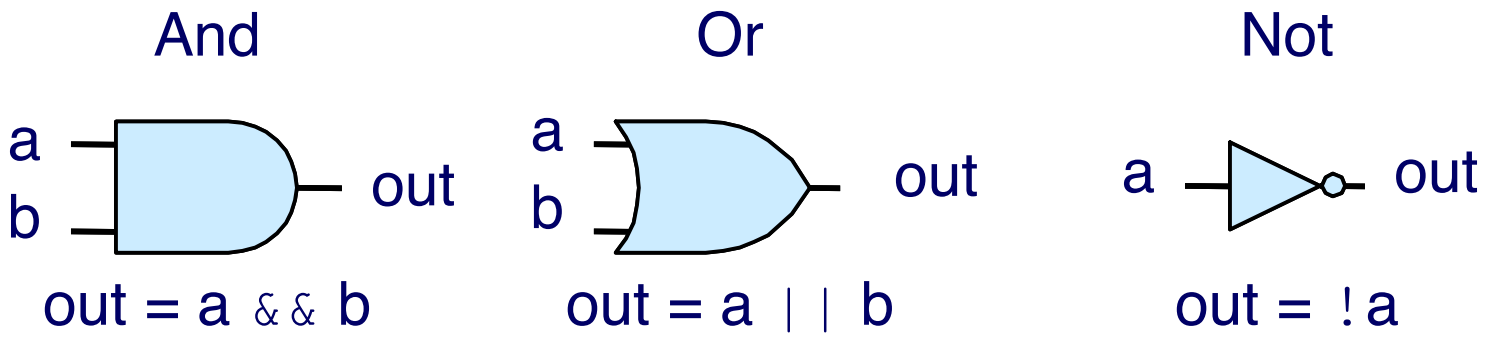


AND

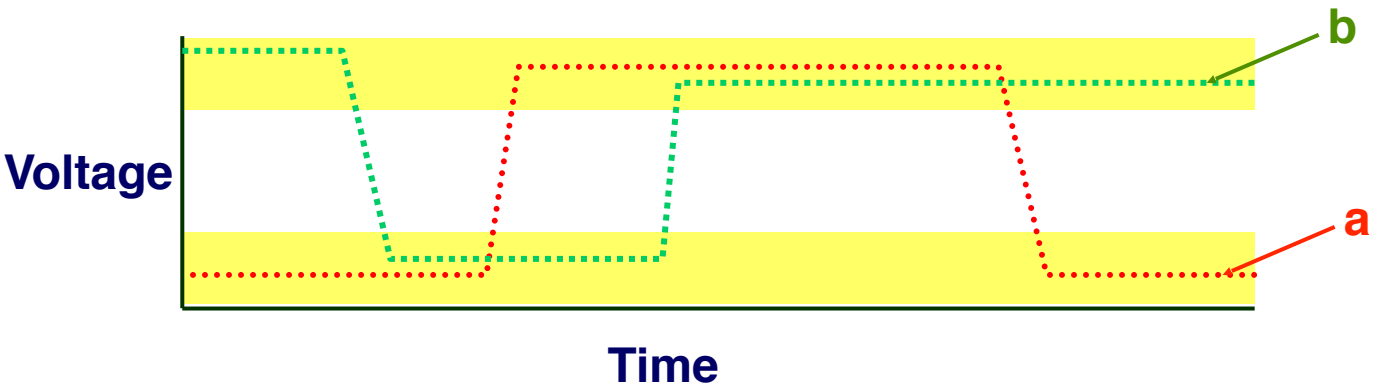


NAND

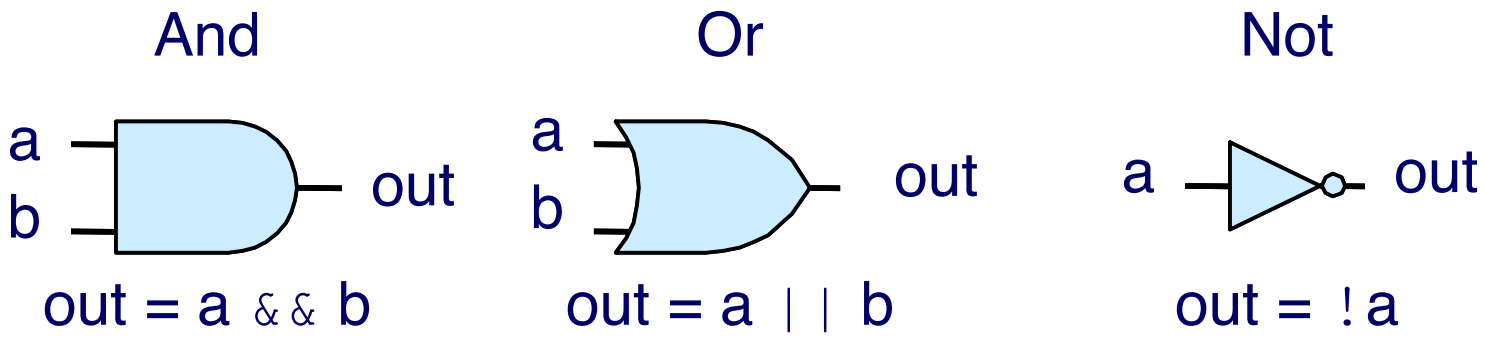
Computing with Logic Gates



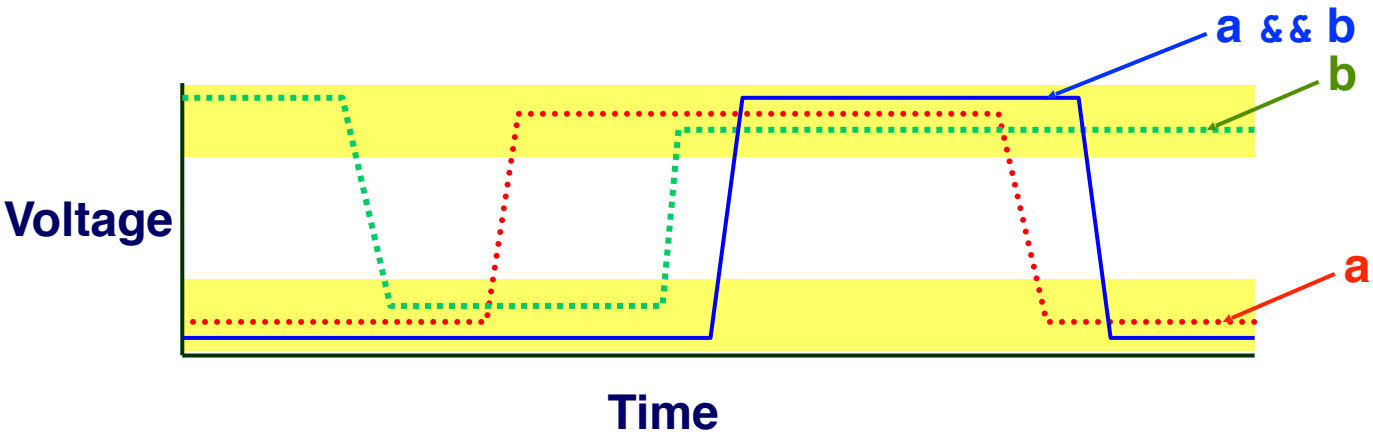
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay
- Different gates have different delays (b/c different transistor combinations)



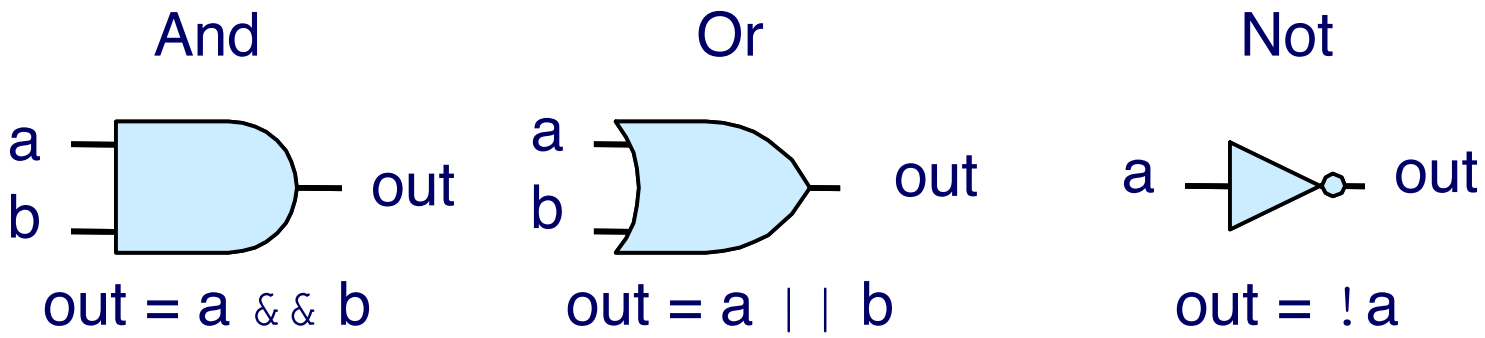
Computing with Logic Gates



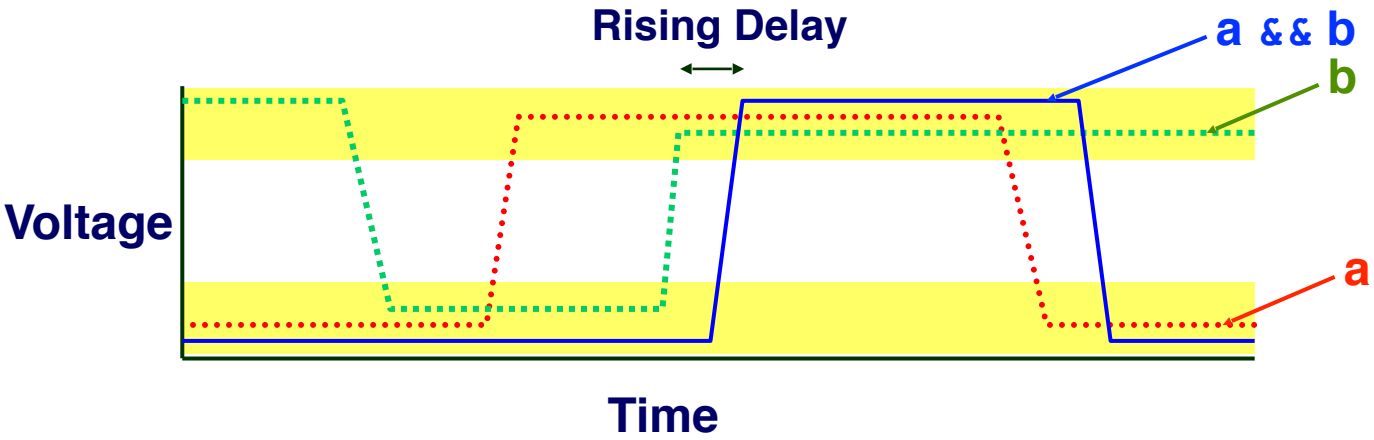
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay
- Different gates have different delays (b/c different transistor combinations)



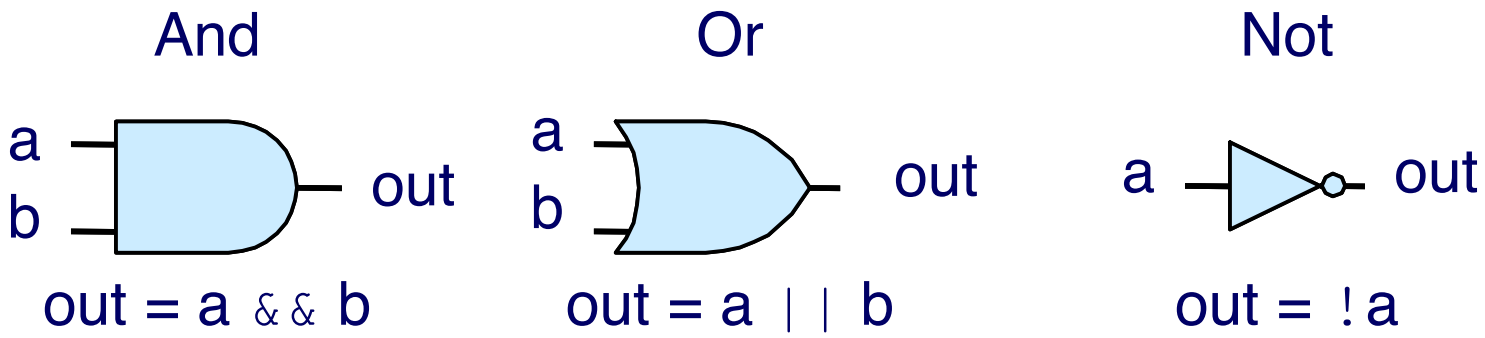
Computing with Logic Gates



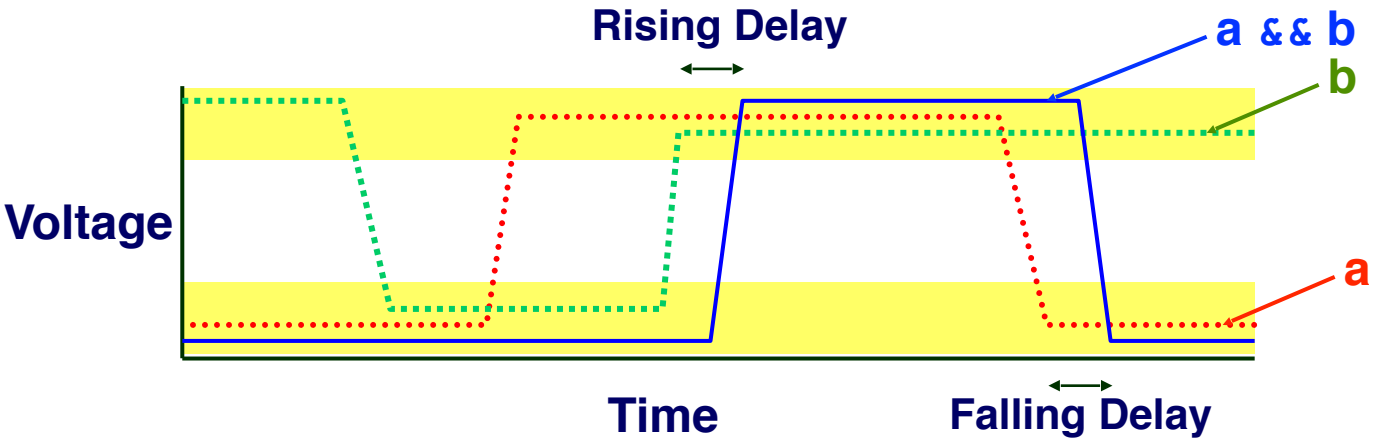
- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay
- Different gates have different delays (b/c different transistor combinations)



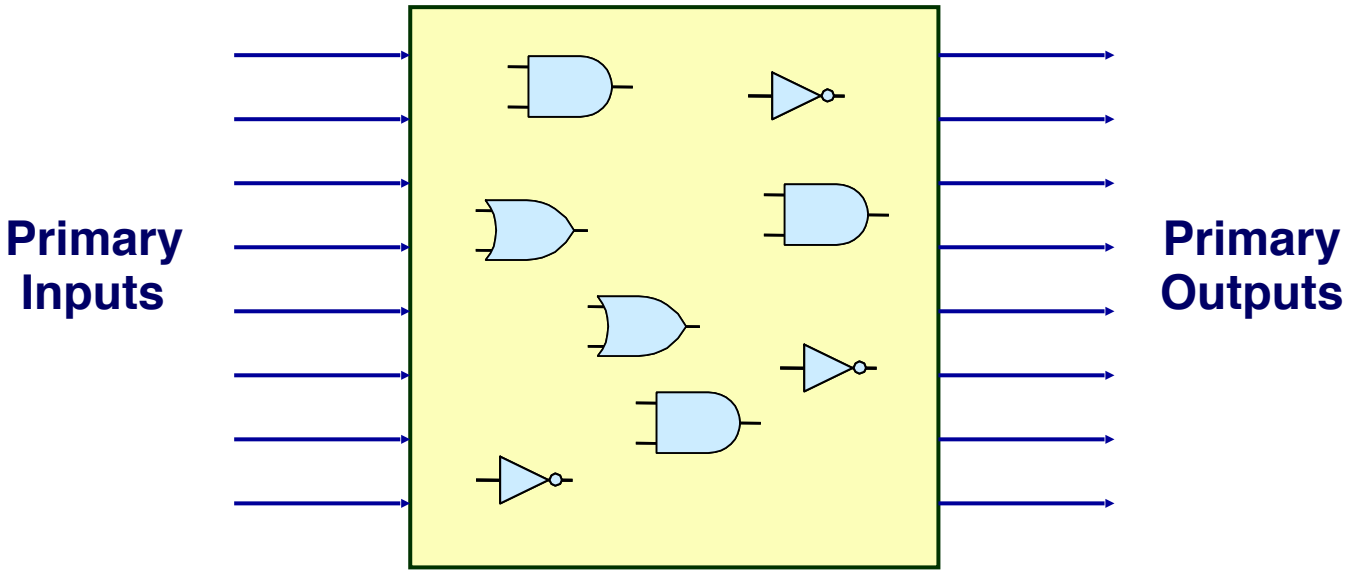
Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs with some small delay
- Different gates have different delays (b/c different transistor combinations)



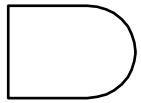
Combinational Circuits



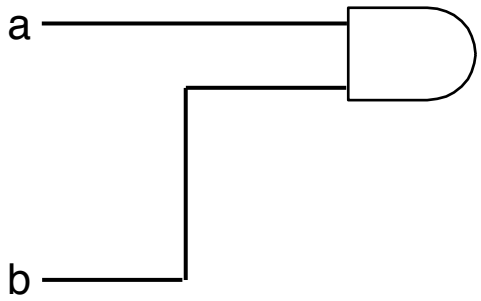
- A Network of Logic Gates
  - Continuously responds to changes on primary inputs
  - Primary outputs become (**after some delay**) Boolean functions of primary inputs

Bit Equality

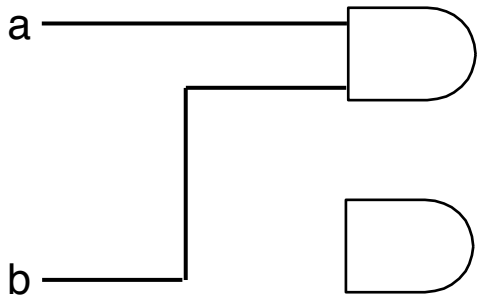
Bit Equality



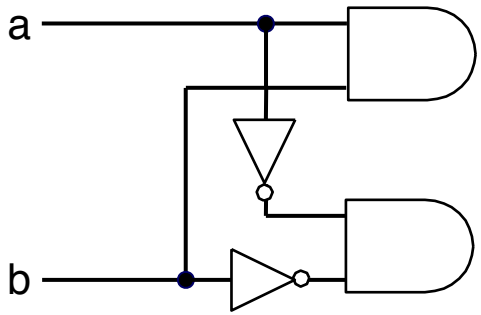
Bit Equality



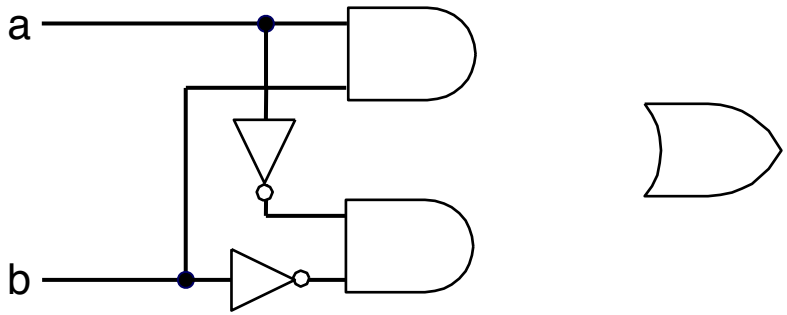
Bit Equality



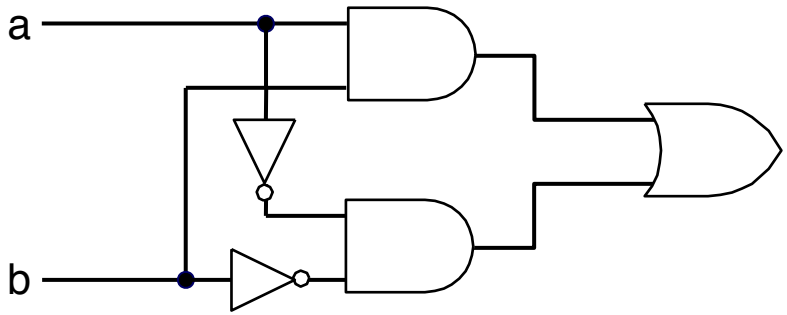
Bit Equality



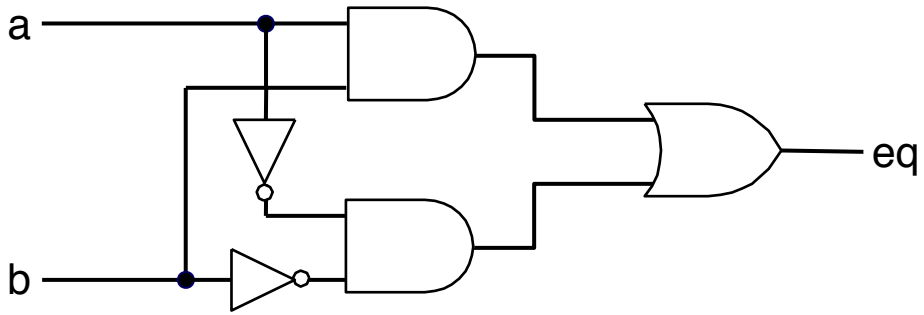
Bit Equality



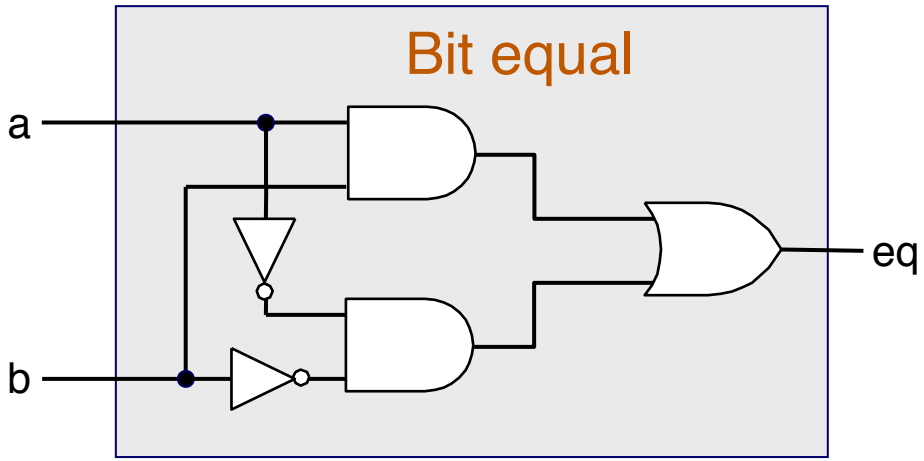
Bit Equality



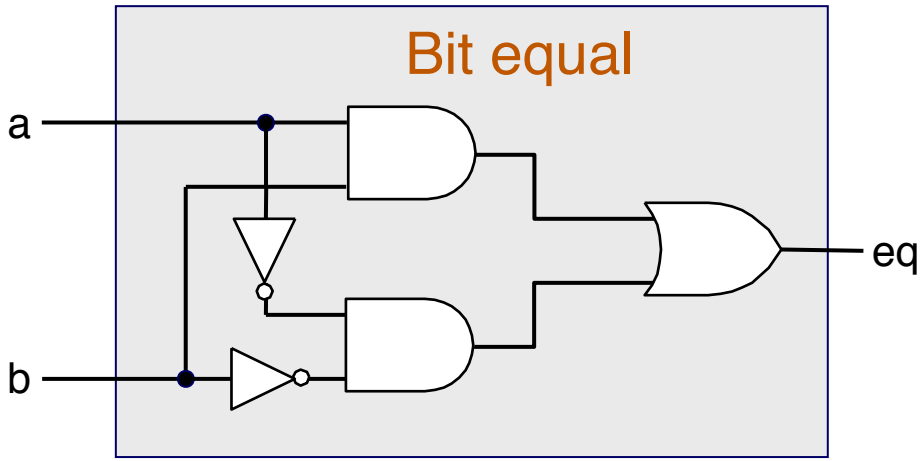
Bit Equality



Bit Equality

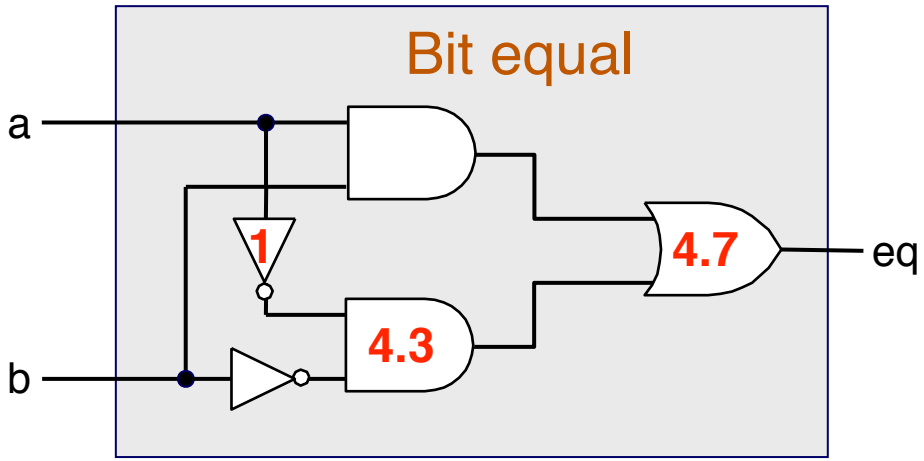


Delay of Bit Equal Circuit



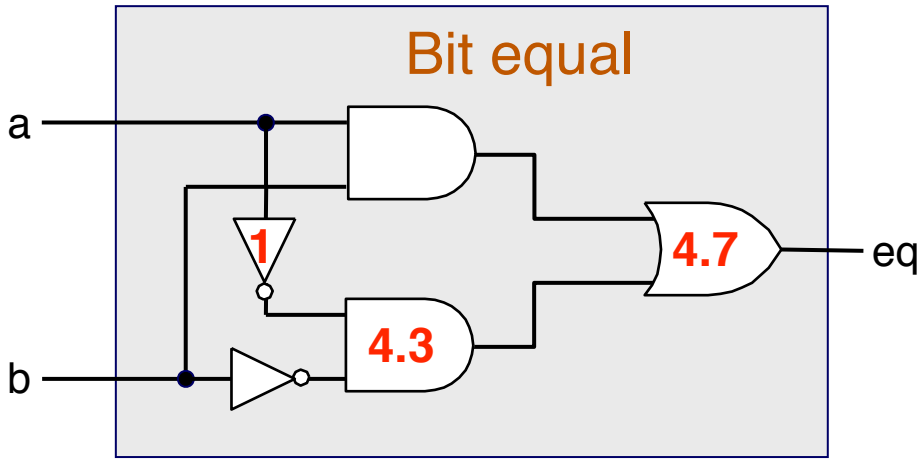
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

Delay of Bit Equal Circuit



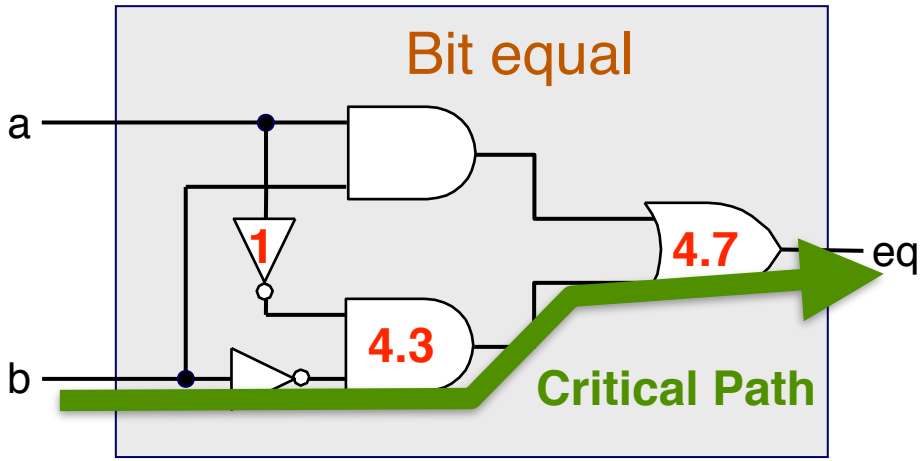
- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7

Delay of Bit Equal Circuit



- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its "critical path"
  - The path between an input and the output that the maximum delay
  - Estimating the critical path delay is called static timing analysis

Delay of Bit Equal Circuit



- What's the delay of this bit equal circuit?
  - Assuming 1-input NOT takes 1 unit of time, 2-input AND takes 4.3, and 2-input OR takes 4.7
- The delay of a circuit is determined by its "critical path"
  - The path between an input and the output that the maximum delay
  - Estimating the critical path delay is called static timing analysis