Announcements

A2 due tomorrow night

A3 will be out this weekend

Today we begin next topic, processor architecture
• Corresponds to book chapter 4
• Suggested readings are on schedule page
Vulnerable Buffer Code

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

void call_echo() {
    echo();
}
Vulnerable Buffer Code

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

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

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

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

void call_echo()
{
    echo();
}

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

unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
Buffer Overflow Stack Example

Before call to gets

Stack Frame for call\_echo

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
</tr>
</tbody>
</table>

Stack Frame for echo

20 bytes unused

Stack Frame for call\_echo

| [3] | [2] | [1] | [0] |

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

```asm
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
```

call\_echo:

```asm
    callq 4006cf <echo>
    add $0x8, %rsp
    ...
```

buf ← %rsp
After call to gets

Stack Frame for call_echo

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
</tr>
<tr>
<td>00</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

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

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

call_echo:

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

buf ← %rsp

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

Overflowed buffer, but did not corrupt state
Buffer Overflow Stack Example #2

After call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 00 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
<tr>
<td>39 38 37 36</td>
</tr>
<tr>
<td>35 34 33 32</td>
</tr>
<tr>
<td>31 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

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

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

call_echo:
...

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

buf ← %rsp

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

Overflowed buffer, and corrupt return address
Buffer Overflow Stack Example #3

After call to gets

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

call_echo:

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

Overflowed buffer, corrupt return address, but program appears to still work!
**Buffer Overflow Stack Example #4**

*After call to gets*

<table>
<thead>
<tr>
<th>Stack Frame for <code>call_echo</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 00</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
<tr>
<td>39 38 37 36</td>
</tr>
<tr>
<td>35 34 33 32</td>
</tr>
<tr>
<td>31 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

```
buf ← %rsp
```

**register_tm_clones:**

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

“Returns” to unrelated code
Could be code controlled by attackers!
What to do about buffer overflow attacks

• Avoid overflow vulnerabilities
• Employ system-level protections
• Have compiler use “stack canaries”
1. Avoid Overflow Vulnerabilities in Code (!)

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

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

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
2. System-Level Protections can help

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

Any attempt to execute this code will fail.
3. Stack Canaries can help

- **Idea**
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function

- **GCC Implementation**
  - `-fstack-protector`
  - Now the default (disabled earlier)
3. Stack Canaries can help

- Idea
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function

- GCC Implementation
  - `-fstack-protector`
  - Now the default (disabled earlier)

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

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

Before call to `gets`

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

Stack Frame for `call_echo`

- Return Address (8 bytes)
- Canary (8 bytes)

```
buf ← %rsp
```

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

### After call to `gets`

<table>
<thead>
<tr>
<th>Stack Frame for <code>call_echo</code></th>
<th>Return Address (8 bytes)</th>
<th>Canary (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00 36 35 34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

### Input: 0123456

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

```
buf ← %rsp
```

### echo:

```
...  
movq 8(%rsp), %rax  # Retrieve from stack
xorq %fs:40, %rax  # Compare to canary
je .L6            # If same, OK
    call __stack_chk_fail  # FAIL
.L6:. . .
```
So far in 252...

C Program

Assembly Program

Instruction Set Architecture

Processor Microarchitecture

Circuits
So far in 252...

C Program

Assembly Program

Instruction Set Architecture

Processor Microarchitecture

Circuits

ret, call
movq, addq
jmp, jne
So far in 252...

C Program → Assembly Program

Instruction Set Architecture

Processor Microarchitecture → Circuits

C Program

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

Assembly Program

```
ret, call
movq, addq
jmp, jne
```
So far in 252...

C Program

Assembly Program

Instruction Set Architecture

Processor Microarchitecture

Circuits

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

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

ret, call
movq, addq
jmp, jne
So far in 252...

C Program

Assembly Program

Instruction Set Architecture

Processor Microarchitecture

Circuits

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

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

ret, call
movq, addq
jmp, jne

Logic gates
So far in 252...

- **C Program**
- **Assembly Program**
- **Instruction Set Architecture**
- **Processor Microarchitecture**
- **Circuits**

**Syntax Elements**: int, float, if, else, +, -, >>

**Assembly Code**:
- `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:

C Program → Assembly Program → Instruction Set Architecture → Processor Microarchitecture → Circuits
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...

C Program

Assembly Program

Instruction Set Architecture

Processor Microarchitecture

Circuits

• 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)
• 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
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>addq</td>
<td>rrmovq</td>
</tr>
<tr>
<td>nop</td>
<td>subq</td>
<td>cmovle</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>andq</td>
<td>cmovl</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>xorq</td>
<td>cmove</td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>jmp</td>
<td>cmovne</td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>jle</td>
<td>cmovge</td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>jl</td>
<td>cmovg</td>
</tr>
<tr>
<td>jXX Dest</td>
<td>je</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>jne</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>jge</td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>jg</td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 \texttt{OPq} category, so additional 2 bits. Similarly, 3 more bits for \texttt{jXX} and \texttt{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???

**In-Order:**
- 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

** SIMD Instructions:**
- addq
- subq
- andq
- xorq
- rrmovq
- cmovle
- cmovl
- cmove
- cmovne
- cmovge
- cmovg
- jmp
- jle
- jl
- je
- jne
- jge
- jg
### Encoding Operands

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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”

<table>
<thead>
<tr>
<th>Register</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0</td>
</tr>
<tr>
<td>%rcx</td>
<td>1</td>
</tr>
<tr>
<td>%rdx</td>
<td>2</td>
</tr>
<tr>
<td>%rbx</td>
<td>3</td>
</tr>
<tr>
<td>%rsp</td>
<td>4</td>
</tr>
<tr>
<td>%rbp</td>
<td>5</td>
</tr>
<tr>
<td>%rsi</td>
<td>6</td>
</tr>
<tr>
<td>%rdi</td>
<td>7</td>
</tr>
<tr>
<td>%r8</td>
<td>8</td>
</tr>
<tr>
<td>%r9</td>
<td>9</td>
</tr>
<tr>
<td>%r10</td>
<td>A</td>
</tr>
<tr>
<td>%r11</td>
<td>B</td>
</tr>
<tr>
<td>%r12</td>
<td>C</td>
</tr>
<tr>
<td>%r13</td>
<td>D</td>
</tr>
<tr>
<td>%r14</td>
<td>E</td>
</tr>
</tbody>
</table>

No Register: F
Encoding Registers

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>A</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>A</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

- 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

\texttt{addq rA, rB} \quad \begin{array}{c} \text{6} \text{ 0} \text{ rA} \text{ rB} \end{array}

Subtract (rA from rB)

\texttt{subq rA, rB} \quad \begin{array}{c} \text{6} \text{ 1} \text{ rA} \text{ rB} \end{array}

And

\texttt{andq rA, rB} \quad \begin{array}{c} \text{6} \text{ 2} \text{ rA} \text{ rB} \end{array}

Exclusive-Or

\texttt{xorq rA, rB} \quad \begin{array}{c} \text{6} \text{ 3} \text{ rA} \text{ rB} \end{array}

- 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

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Move Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

irmovq $0xabcd, %rdx
# Move Instructions

<table>
<thead>
<tr>
<th><strong>Byte</strong></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>halt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>nop</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>cmovXX rA, rB</strong></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>irmovq V, rB</strong></td>
<td>3</td>
<td>0</td>
<td>f</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>rmmovq rA, D(rB)</strong></td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>mrmovq D(rB), rA</strong></td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OPq rA, rB</strong></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>jXX Dest</strong></td>
<td>7</td>
<td></td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>call Dest</strong></td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pushq rA</strong></td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>popq rA</strong></td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Move Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Move Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Move Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>fn</td>
<td>rA rB</td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td>F rB V</td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td>rA rB D</td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>0</td>
<td>rA rB</td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>fn</td>
<td>rA rB</td>
</tr>
<tr>
<td>jXX Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>fn</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mrmovq -12(%rbp),%rcx
# Move Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Move Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The instruction length limits the immediate value and displacement.
Move Instruction Examples

Y86-64

iramovq $0xabcd, %rdx

**Encoding:** 30 82 cd ab 00 00 00 00 00 00

rrmovq %rsp, %rbx

**Encoding:** 20 43

rmrovq -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

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Jump/Call Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td>jle .L4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Jump/Call Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td>jle .L4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The assembly would assume a start address of the program, and then calculates the address of each instruction.
## Jump/Call Instructions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovq V, rB</td>
<td>3</td>
<td>0</td>
<td>F</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>rmmovq rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mrmovq D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPq rA, rB</td>
<td>6</td>
<td>fn</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td>Dest (essentially the target address)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushq rA</td>
<td>A</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>popq rA</td>
<td>B</td>
<td>0</td>
<td>rA</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
The assembly would assume a start address of the program, and then calculates the address of each instruction.
How Does An Assembler Work?

- `rmmovq rA, D(rB)`
  - OP: 4
  - FN: 0
  - Destination: D

- `OPq rA, rB`
  - OP: 6
  - FN: 0
  - Source: rA, rB

- `jXX Dest`
  - OP: 7
  - FN: 0
  - Destination: Dest

- `call Dest`
  - OP: 8
  - FN: 0
  - Destination: Dest
How Does An Assembler Work?

```
rmmovq rA, D(rB)
rmmovq %rsi, 0x41c(%rsp)
addq %rax, %rsi
jmp .L0
ret
```

```
OPq rA, rB
```

```
jXX Dest
```

```
call Dest
```

```
0x100 <foo>
```

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

```
... ...
```

```
ret
```

```
addq %rax, %rsi
```

```
call <foo>
call <foo>
call <foo>
call <foo>
call <foo>
```

```
jmp .L0
```

```
... ...
```

```
.L0
```

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

```
... ...
```
How Does An Assembler Work?

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

0x100 <foo> rmmovq $rsi,0x41c(%rsp) 40 64 1c 04 00 00 00 00 00 00

... ...
ret
addq %rax,%rsi
call <foo>
jmp .L0
... ...
.L0 irmovq $0xabcd, %rdx
```
How Does An Assembler Work?

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

OPq rA, rB 6 fn rA rB

ejXX Dest 7 fn Dest

call Dest 8 0 Dest

0x100 <foo>  rmmovq %rsi,0x41c(%rsp) 40 64 1c 04 00 00 00 00 00 00 00 00

... ...

ret 90

addq %rax,%rsi

call <foo>

jmp .L0

... ...

.L0  irmovq $0xabcd, %rdx
How Does An Assembler Work?

rmmovq rA, D(rB)

OPq rA, rB

jXX Dest

call Dest

0x100 <foo> rmmovq %rsi,0x41c(%rsp) 40 64 1c 04 00 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?

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

0x100 <foo> rmmovq %rsi,0x41c(%rsp) 4 0 64 1c 04 00 00 00 00 00 00 00 00

... ...
ret 90
addq %rax,%rsi 60 06
call <foo> 80 00 01 00 00 00 00 00 00 00 00
jmp .L0 ...
... ...
.L0 irmovq $0xabcd, %rdx
```
How Does An Assembler Work?

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

OPq  rA, rB  6  fn  rA  rB

jXX  Dest  7  fn  Dest

call  Dest  8  0  Dest

0x100  <foo>  rmmovq  %rsi,0x41c(%rsp)  40 64 1c 04 00 00 00 00 00 00 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?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rmmovq rA, D(rB)</code></td>
<td>0x40</td>
<td>Moves register <code>rA</code> to memory location <code>D(rB)</code></td>
</tr>
<tr>
<td><code>OPq rA, rB</code></td>
<td>0x60</td>
<td>Performs operation <code>OPq</code> on registers <code>rA</code> and <code>rB</code></td>
</tr>
<tr>
<td><code>jXX Dest</code></td>
<td>0x70</td>
<td>Jumps to <code>Dest</code></td>
</tr>
<tr>
<td><code>call Dest</code></td>
<td>0x80</td>
<td>Calls function <code>Dest</code></td>
</tr>
</tbody>
</table>

**Example Assembly Code:**

```assembly
0x100 <foo>
```
```
rmmovq `%rsi`, 0x41c(%rsp)
```
```
... ...
```
```
ret
```
```
addq `%rax`, `%rsi`
```
```
call <foo>
```
```
jmp .L0
```
```
... ...
```
```
.L0 irmovq $0xabcd, %rdx
```
```
... ...
```
```
... ...
```
```
```
How Does An Assembler Work?

rmmovq rA, D(rB)

OPq rA, rB

jXX Dest

call Dest

0x100 <foo>

rmmovq %rsi,0x41c(%rsp) 40 64 1c 04 00 00 00 00 00 00 00 00

... ... 90

ret

addq %rax,%rsi 60 06

call <foo> 80 00 01 00 00 00 00 00 00 00

jmp .L0 70 ??????????

... ... 30 f2 cd ab 00 00 00 00 00 00 00 00
How Does An Assembler Work?

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

OPq rA, rB  6  fn  rA  rB

jXX Dest  7  fn  Dest

call Dest  8  0  Dest

0x100  <foo>  rmmovq  %rsi,0x41c(%rsp)  40 64 1c 04 00 00 00 00 00 00 00

... ... 0x100 + the lengths of all instructions in-between

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 00
How Does An Assembler Work?

- `rmmovq rA, D(rB)`

- `OPq rA, rB`

- `jXX Dest`

- `call Dest`

---

- `0x100 <foo>`
  - `rmmovq %rsi,0x41c(%rsp)`

- `0x100 + the lengths of all instructions in-between`
  - `ret`
  - `addq %rax,%rsi`
  - `call <foo>`
  - `jmp .L0`

- `0x200 .L0`
  - `irmovq $0xabcd, %rdx`
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

```
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 00
       jmp .L0          70 00 02 00 00 00 00 00 00 00
       ... ...
0x200  .L0     irmovq $0xabcd, %rdx  30 f2 cd ab 00 00 00 00 00 00
```
## Jump Instructions

### Jump Unconditionally

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp Dest</td>
<td>7 0</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Less or Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jle Dest</td>
<td>7 1</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Less

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jl Dest</td>
<td>7 2</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>je Dest</td>
<td>7 3</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Not Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jne Dest</td>
<td>7 4</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Greater or Equal

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jge Dest</td>
<td>7 5</td>
<td>Dest</td>
</tr>
</tbody>
</table>

### Jump When Greater

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jg Dest</td>
<td>7 6</td>
<td>Dest</td>
</tr>
</tbody>
</table>
# Subroutine Call and Return

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>call Dest</td>
<td>8 0</td>
<td>Push address of next instruction onto stack</td>
</tr>
<tr>
<td></td>
<td>Dest</td>
<td>Start executing instructions at Dest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Like x86-64</td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td>Pop value from stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use as address for next instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Like x86-64</td>
</tr>
</tbody>
</table>
One More Complication...

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>jxx</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dest (essentially the target address)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jle .L4</td>
</tr>
<tr>
<td>call</td>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dest (essentially the start address of the callee)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>call foo</td>
</tr>
</tbody>
</table>
### One More Complication...

<table>
<thead>
<tr>
<th>Byte</th>
<th>jxx Dest</th>
<th>Dest (essentially the target address)</th>
<th>jle .L4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte</th>
<th>call Dest</th>
<th>Dest (essentially the start address of the callee)</th>
<th>call foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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…

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>jxx</td>
<td>7</td>
<td>fn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dest (essentially the target address)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jle</td>
<td>.L4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dest (essentially the start address of the callee)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>foo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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...

<table>
<thead>
<tr>
<th>Byte</th>
<th>jxx Dest</th>
<th>call Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x41c(%rsp)</td>
<td>rmmovq %rsi, 0x41c(%rsp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td>60 06</td>
<td>addq %rax, %rsi</td>
</tr>
<tr>
<td>0x180</td>
<td></td>
<td>call &lt;foo&gt;</td>
</tr>
<tr>
<td>0x185</td>
<td></td>
<td>jmp .L0</td>
</tr>
<tr>
<td>0x200</td>
<td></td>
<td>irmovq $0xabcd, %rdx</td>
</tr>
</tbody>
</table>
Using Relative Addresses for Jumps

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>_binary</th>
<th>Instruction</th>
<th>_binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td><code>&lt;foo&gt;</code> rmmovq <code>%rsi</code>, 0x41c(%rsp)</td>
<td>40 64 1c 04 00 00 00 00 00 00 00</td>
<td>0x180</td>
<td>call <code>&lt;foo&gt;</code></td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>addq <code>%rax</code>, <code>%rsi</code></td>
<td>60 06</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>jmp <code>.L0</code></td>
<td>70 00 02 00 00 00 00 00 00 00 00 00</td>
<td>0x200</td>
<td><code>irmovq $0xabcd, %rdx</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relative addr: -0x80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The code snippet for the function `<foo>`:

```assembly
rmmovq %rsi, 0x41c(%rsp)
addq %rax, %rsi
jmp .L0
irmovq $0xabcd, %rdx
```
Using Relative Addresses for Jumps

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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>rmovq %rsi, 0x41c(%rsp)</td>
<td>40 64 1c 04 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x180</td>
<td>call &lt;foo&gt;</td>
<td>80 00 00 00 11 11 11 11 11 11</td>
</tr>
<tr>
<td>0x185</td>
<td>jmp .L0</td>
<td>70 00 02 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x200</td>
<td>irmovq $0xabcd, %rdx</td>
<td>30 f2 cd ab 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
```
Using Relative Addresses for Jumps

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

```
38 f2 cd ab 00 00 00 00 00 00 00
```

```
90
```

```
60 06
```

```
80 00 00 00 11 11 11 11 11 11
```

```
70 00 02 00 00 00 00 00 00 00
```

```
30 f2 cd ab 00 00 00 00 00 00 00
```
Using Relative Addresses for Jumps

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

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

```
30 f2 cd ab 00 00 00 00 00 00 00
90
60 06
80 00 00 00 11 11 11 11 11
70 7B 00 00 00 00 00 00 00 00
40 64 1c 04 00 00 00 00 00 00
```

```asm
rmmovq %rsi, 0x41c(%rsp)
addq %rax, %rsi
jmp .L0
```
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)

```
@0x100 <foo>
  rmmovq %rsi,0x41c(%rsp)  40 64 1c 04 00 00 00 00 00 00
  ... ... 90
  ret 60 06
  addq %rax,%rsi 60 06
  call <foo> 80 00 00 00 11 11 11 11 11 11
  jmp .L0 70 7B 00 00 00 00 00 00 00 00
  ... ... 0x7B
@0x200 .L0
  irmovq $0xabcd, %rdx 30 f2 cd ab 00 00 00 00 00 00
```
Miscellaneous Instructions

- **nop**
  
  | nop | 1 | 0 |

- **halt**
  
  | halt | 0 | 0 |

- 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