System Programming

02. A simple computer and Y86-64 ISA (ch 4.1)

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

A simple computer

These slides are based from http://cs.lmu.edu/~ray/notes/simplecomputer/

Let us try to understand

C0000003
00000001
000F4240
00000001
30000064
40000001
10000001
50000002
E0000003
C0000009

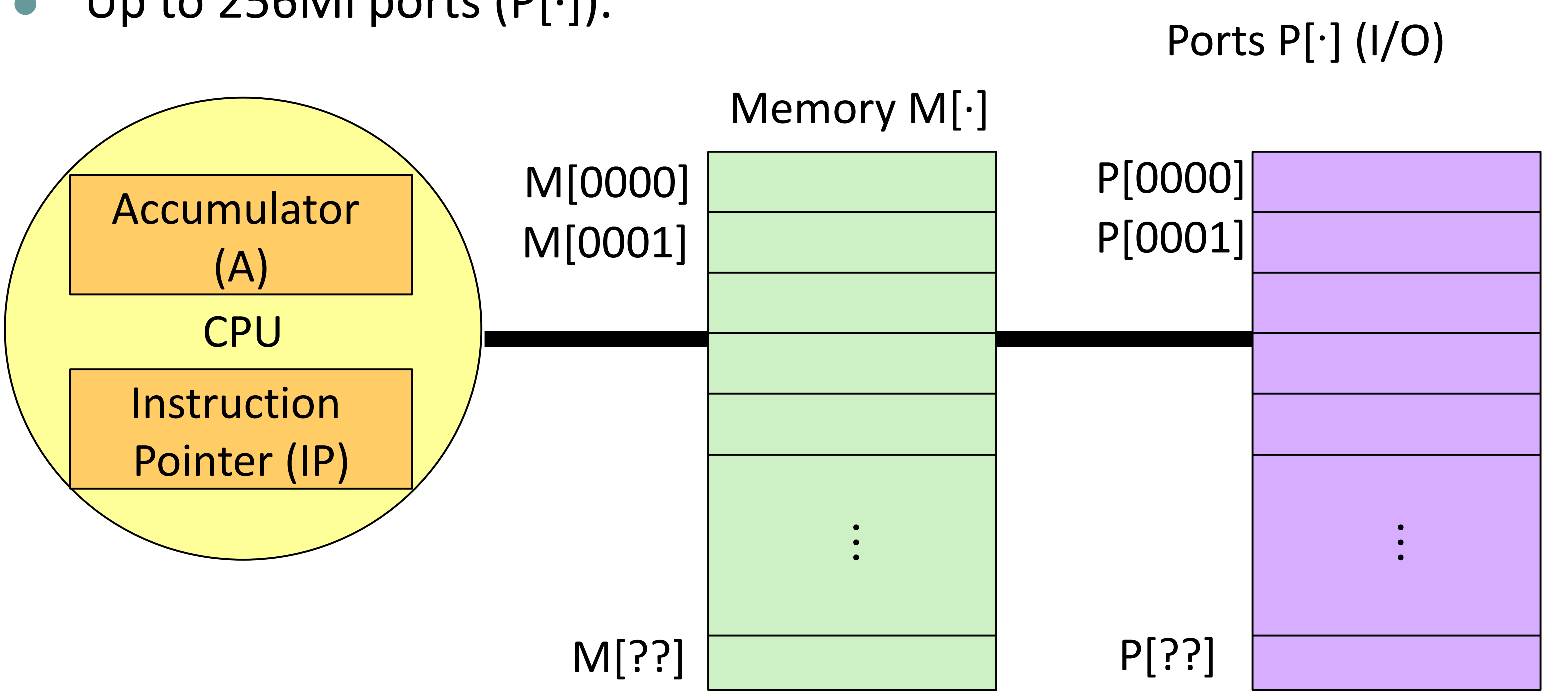
What does it mean?

How can we interpret?

What information we need?

Really, a simple computer

- Basic Components
  - A 32-bit register, called the accumulator (A).
  - A 32-bit register, called the instruction pointer (IP).
  - 256Mi words of memory, each 32-bits wide (M[·]).
    - 256 MiB = $256 * 2^{20} = 2^{8} * 2^{20} = 2^{28}$
  - Up to 256Mi ports (P[·]).



Operation

- At each “step”
  - the computer reads the memory word whose address is in IP and then increments IP
  - Then it carries out the instruction that that word represents.

  - This is how most computers, uh, cores, work. It is called the **fetch-execute cycle.**

Again, the code. What we know?

C0000003
00000001
000F4240
00000001
30000064
40000001
10000001
50000002
E0000003
C0000009

- 28 bits will refer to a memory address or port number
  - Which 28 bits?
- How about remaining 4 bits ?
- Assumption
  - The first 4 bits as the **opcode**.
  - The remaining 28 bits will refer to a memory address or port number.

The Instructions

- We formally specify the operation of each instruction as follows.
  - All arithmetic is signed, modular, 32-bit integer arithmetic.

op	Action	Remarks
0	A:=M[x]	Load accumulator from memory
1	M[x]:=A	Store accumulator to memory
2	A:=P[x]	Read from a port into the accumulator
3	P[x]:=A	Write accumulator out to a port
4	A:=A+M[x]	Add into accumulator
5	A:=A−M[x]	Subtract from accumulator
6	A:=A×M[x]	Multiply into accumulator
7	A:=A÷M[x]	Divide accumulator
8	A:=A mod M[x]	Modulo
9	A:=A∧M[x]	Bitwise AND
A	A:=A∨M[x]	Bitwise OR
B	A:=A⊕M[x]	Bitwise XOR
C	IP:=x	Jump to new address
D	if A=0 then IP:=x	Jump if accumulator is zero
E	if A<0 then IP:=x	Jump if accumulator is less than zero
F	if A>0 then IP:=x	Jump if accumulator is greater than zero

Meaning

Memory Address	
0000	C0000003
0001	00000001
0002	000F4240
0003	00000001
0004	30000064
0005	40000001
0006	10000001
0007	50000002
0008	E0000003
0009	C0000009

C	IP:=x	Jump to new address
0	A:=M[x]	Load accumulator from memory

F4240 (hex) = 1,000,000 (decimal)

3	P[x]:=A	Write accumulator out to a port

64 (hex) = 100 (decimal)

4	A:=A+M[x]	Add into accumulator
1	M[x]:=A	Store accumulator to memory
5	A:=A−M[x]	Subtract from accumulator
E	if A<0 then IP:=x	Jump if accumulator is less than zero
C	IP:=x	Jump to new address

An Example Program

Memory Address	
0000	C0000003
0001	00000001
0002	000F4240
0003	00000001
0004	30000064
0005	40000001
0006	10000001
0007	50000002
0008	E0000003
0009	C0000009

- when loaded into memory at address 0
  - outputs powers of two, starting with 1, and going just past 1,000,000, to port 100 (64 hex):

Machine code

- Machine code
  - Hard to read
  - Just listing the contents of memory that the processor executes

0000	C0000003
0001	00000001
0002	000F4240
0003	00000001
0004	30000064
0005	40000001
0006	10000001
0007	50000002
0008	E0000003
0009	C0000009

Assembly code (1/2)

- Let's use mnemonics for each instruction
  - the mnemonics will be
    - LOAD, STORE, IN, OUT, ADD, SUB, MUL, DIV, MOD, AND, OR, XOR, JUMP, JZ, JLZ, JGZ

```
0000  C0000003           JUMP   start     ; begin by jumping over the data area
0001  00000001  pow:     1                ; store the current power value here
0002  000F4240  limit:   1000000          ; we'll be computing powers up to this amount
0003  00000001  start:   LOAD   pow       ; bring the value into accumulator to use
0004  30000064           OUT    100       ; output the current power
0005  40000001           ADD    pow       ; adding to itself makes the next power!
0006  10000001           STORE  pow       ; store it (for next time)
0007  50000002           SUB    limit     ; compare with limit, subtracting helps
0008  E0000003           JLZ    start     ; if not yet past limit, keep going
0009  C0000009  end:     JUMP   end       ; this "stops" the program!
```

Assembly code (2/2)

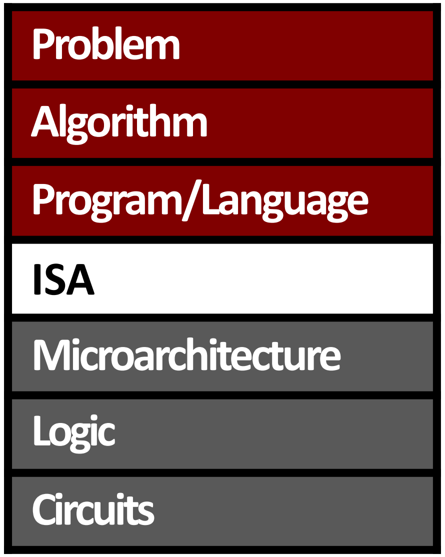
- In general, each line of an assembly language program contains:
  - An optional label
    - so you don't have to memorize physical addresses
  - Either
    - A data value
    - An instruction and its operands. An operand can be a direct value or a label.
      - Labels are just convenient shorthands for values anyway.
  - Comments, beginning with the ; character.

Ch04.1. Y86-64 ISA

These slides are based on the text.

Levels of Transformation

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - ISA, uarch, circuits
  - “Architecture” = ISA + microarchitecture



ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implements “acceleration”
  - Add instruction vs. Adder implementation
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Bit serial, ripple carry, carry lookahead adders
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, …
- Uarch usually changes faster than ISA
  - Few ISAs (x86, SPARC, MIPS, Alpha) but many uarchs

ISA

- Instructions
  - Opcodes, Addressing Modes
  - Instruction Types and Format
  - Registers, Condition Codes

- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O
- Task Management
- Power and Thermal Management
- Multi-threading support, Multiprocessor support

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 1:
Basic Architecture

Example ISAs

- x86 — dominant in desktops, servers
- ARM — dominant in mobile devices
- POWER — Wii U, IBM supercomputers and some servers
- MIPS — common in consumer wifi access points
- SPARC — some Oracle servers, Fujitsu supercomputers
- z/Architecture — IBM mainframes
- Z80 — TI calculators
- SHARC — some digital signal processors
- Itanium — some HP servers
- RISC V — some embedded
- …

ISA Tradeoffs

- operations
  - how many?
  - which ones
- operands
  - how many?
  - location
  - types
  - how to specify?
- instruction format
  - size
  - how many formats?

Instruction length (1/2)

- Fixed length: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions (Why is this bad?)
  - -- Harder-to-extend ISA (how to add new instructions?)

Instruction length (2/2)

- Variable length: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding (Why is this good?)
  - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently

Addressing modes

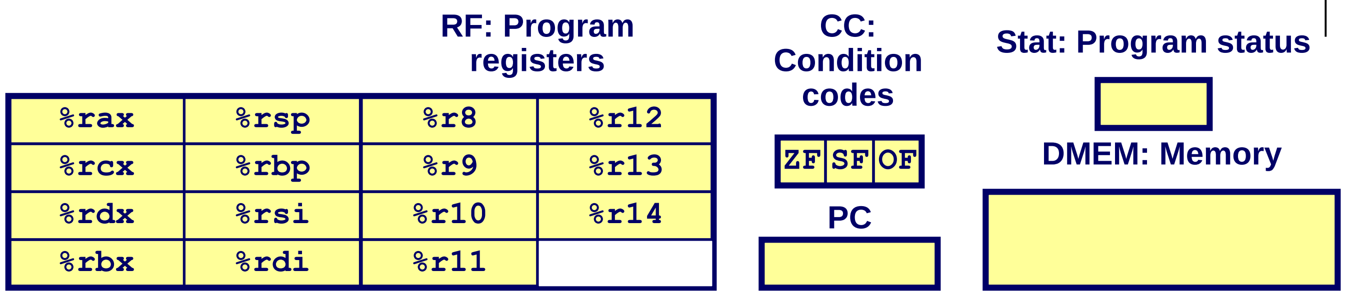
- Addressing mode
  - specifies how to obtain an operand of an instruction
    - Register
    - Immediate
    - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, …)
- Example
  - x86-64: 10(%r11,%r12,4)
  - ARM: %r11 << 3 (shift register value by constant)
  - VAX: ((%r11)) (register value is pointer to pointer)

Condition codes

- Codes
  - cmpq %r11, %r12
  - je somewhere

- could do:
  - /* _Branch if _EQual */
  - beq %r11, %r12, somewhere

Y86-64 Processor State



- Program Registers
  - 15 registers (omit %r15). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
  - ZF: Zero SF:Negative OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Y86-64 Instruction Set

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

Y86-64 Instructions

- 1 – 10 bytes of information read from memory
  - Can determine instruction length from first byte
- Only supports 64-bit operations
- RISC style
  - Not as many instruction types, and simpler encoding than with x86-64
  - Simple addressing mode: D(rA)
  - ALU instructions operate on registers (not memory)
  - Registers are specified in the fixed location, if any
- Each accesses and modifies some part(s) of the program state

Y86-64 Conditional Move Instructions

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

`rrmovq`	2 0
`cmovle`	2 1
`cmovl`	2 2
`cmove`	2 3
`cmovne`	2 4
`cmovge`	2 5
`cmovg`	2 6

Y86-64 ALU Instructions

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

`addq`	6 0
`subq`	6 1
`andq`	6 2
`xorq`	6 3

Y86-64 Conditional Branch Instructions

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

Instruction	Encoding
`jmp`	7 0
`jle`	7 1
`jl`	7 2
`je`	7 3
`jne`	7 4
`jge`	7 5
`jg`	7 6

Encoding Registers

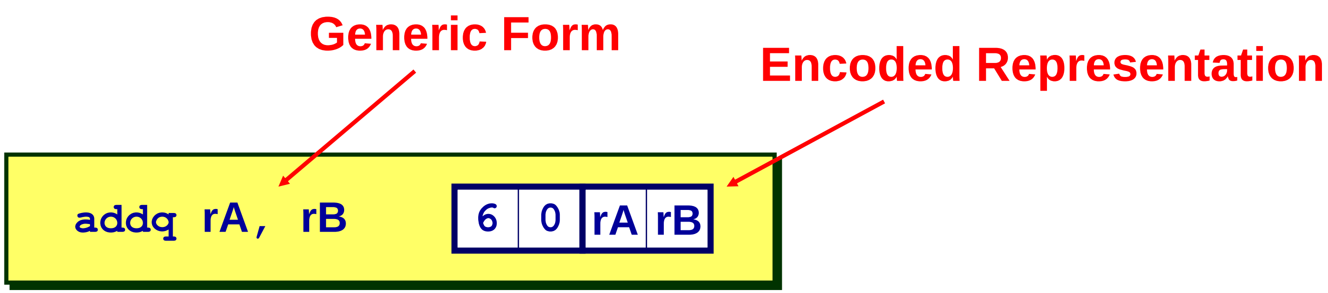
- Each register has 4-bit ID
  - Same encoding as in x86-64

%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

- Register ID 15 (0xF) indicates “no register”
  - Will use this in our hardware design in multiple places

Instruction Example

- Addition Instruction
  - Add value in register rA to that in register rB
    - Store result in register rB
    - Note that Y86-64 only allows addition to be applied to register data
  - 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

Instruction Code → 6, **Function Code** → 0

Add

`addq rA, rB`	6	0	rA	rB

Subtract (rA from rB)

`subq rA, rB`	6	1	rA	rB

And

`andq rA, rB`	6	2	rA	rB

Exclusive-Or

`xorq rA, rB`	6	3	rA	rB

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

Move Operations

Register → Register

`rrmovq rA, rB`

2	0

Immediate → Register

`irmovq V, rB`

3	0	F	rB	V

Register → Memory

`rmmovq rA, D(rB)`

4	0	rA	rB	D

Memory → Register

`mrmovq D(rB), rA`

5	0	rA	rB	D

- Like the x86-64 movq instruction
- Simpler format for memory addresses
- Give different names to keep them distinct

Move Instruction Examples

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

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

X86-64 | **Y86-64**

```
movq $0xabcd, %rdx
```
```
irmovq $0xabcd, %rdx
```
Encoding: `30 F2 cd ab 00 00 00 00 00 00`

```
movq %rsp, %rbx
```
```
rrmovq %rsp, %rbx
```
Encoding: `20 43`

```
movq -12(%rbp),%rcx
```
```
mrmovq -12(%rbp),%rcx
```
Encoding: `50 15 f4 ff ff ff ff ff ff ff`

```
movq %rsi,0x41c(%rsp)
```
```
rmmovq %rsi,0x41c(%rsp)
```
Encoding: `40 64 1c 04 00 00 00 00 00 00`

Conditional Move Instructions

Move Unconditionally

`rrmovq` rA, rB	2	0	rA	rB

Move When Less or Equal

`cmovle` rA, rB	2	1	rA	rB

Move When Less

`cmovl` rA, rB	2	2	rA	rB

Move When Equal

`cmove` rA, rB	2	3	rA	rB

Move When Not Equal

`cmovne` rA, rB	2	4	rA	rB

Move When Greater or Equal

`cmovge` rA, rB	2	5	rA	rB

Move When Greater

`cmovg` rA, rB	2	6	rA	rB

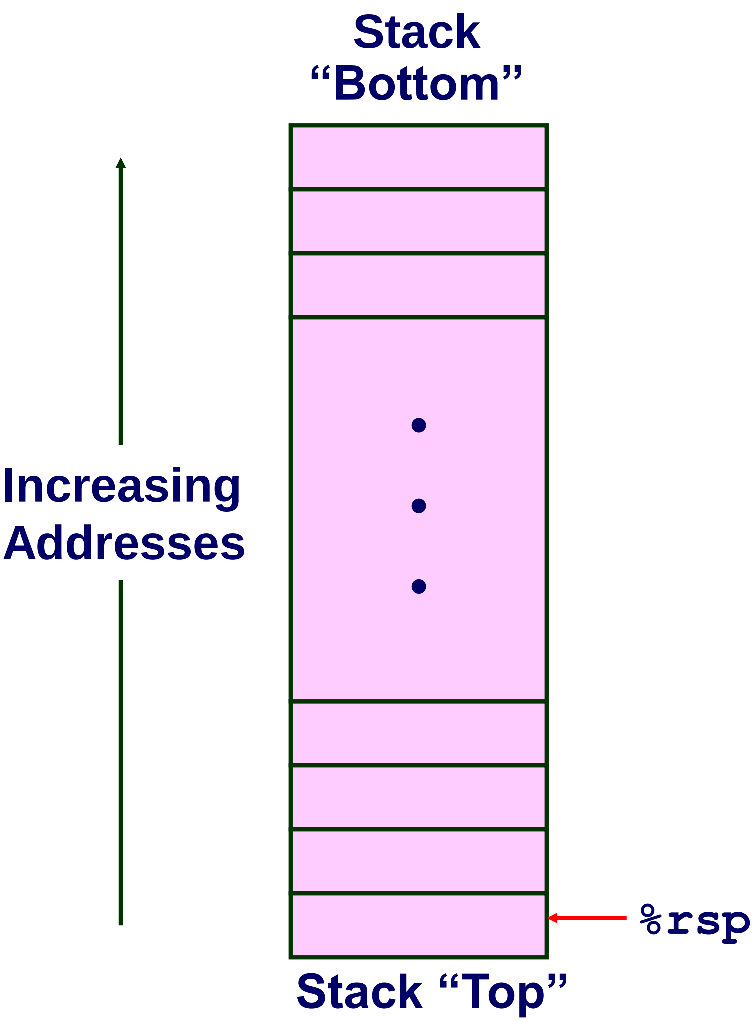
- Refer to generically as "cmovXX"
- Encodings differ only by "function code"
- Based on values of condition codes
- Variants of rrmovq instruction
  - (Conditionally) copy value from source to destination register

Jump Instructions

Instruction	Encoding		
Jump Unconditionally			
`jmp` **Dest**	7	0	Dest
Jump When Less or Equal			
`jle` **Dest**	7	1	Dest
Jump When Less			
`jl` **Dest**	7	2	Dest
Jump When Equal			
`je` **Dest**	7	3	Dest
Jump When Not Equal			
`jne` **Dest**	7	4	Dest
Jump When Greater or Equal			
`jge` **Dest**	7	5	Dest
Jump When Greater			
`jg` **Dest**	7	6	Dest

- Refer to generically as “jXX”
- Encodings differ only by “function code”
- Based on values of condition codes
- Same as x86-64 counterparts
- Encode full destination address
  - Unlike PC-relative addressing seen in x86-64

Y86 Program Stack



- Region of memory holding program data
- Used in Y86-64 (and x86-64) for supporting procedure calls
- Stack top indicated by %rsp
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer

Stack Operations

`pushq rA`

A	0	rA	F

- Decrement %rsp by 8
- Store word from rA to memory at %rsp
- Like x86-64

`popq rA`

B	0	rA	F

- DecreRead word from memory at %rsp
- Save in rA
- Increment %rsp by 8
- Like x86-64

Subroutine Call and Return

`call` **Dest**	8	0	Dest

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

`ret`	9	0

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

Miscellaneous Instructions

nop	1	0

- Don't do anything

halt	0	0

- Stop executing instructions
- x86-64 has comparable instruction, but can't execute it in user mode
- We will use it to stop the simulator
- Encoding ensures that program hitting memory initialized to zero will halt

Status Conditions

Mnemonic	Code
AOK	1

• Normal operation

Mnemonic	Code
HLT	2

• Halt instruction encountered

Mnemonic	Code
ADR	3

• Bad address (either instruction or data) encountered

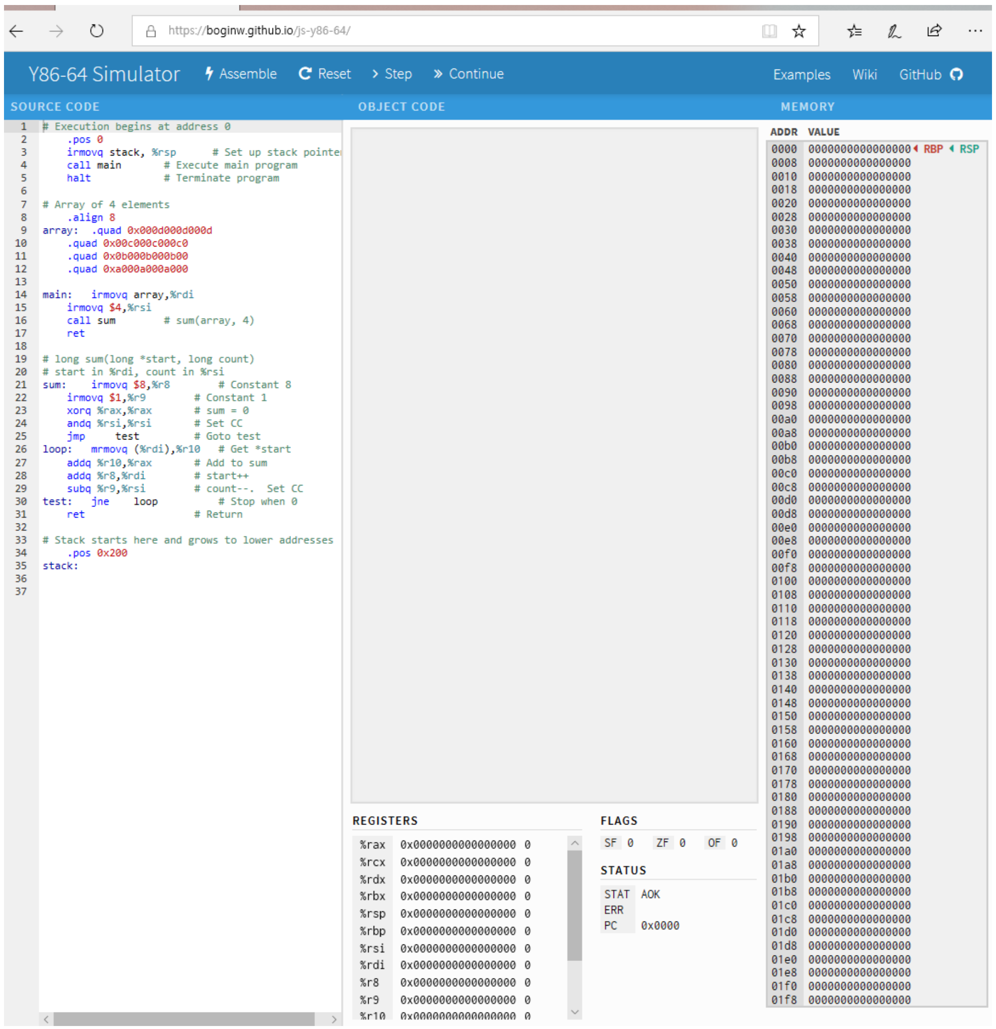
Mnemonic	Code
INS	4

• Invalid instruction encountered

- Desired Behavior
  - If AOK, keep going
  - Otherwise, stop program execution

Y86-64 Simulator

- https://boginw.github.io/js-y86-64/



Y86 Assembly example (1/2)

```
# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp      # Set up stack pointer
    call main       # Execute main program
    halt            # Terminate program

# Array of 4 elements
    .align 8
array:  .quad 0x000d000d000d
    .quad 0x00c000c000c0
    .quad 0x0b000b000b00
    .quad 0xa000a000a000

main:   irmovq array,%rdi
    irmovq $4,%rsi
    call sum        # sum(array, 4)
    ret

# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:    irmovq $8,%r8        # Constant 8
    irmovq $1,%r9        # Constant 1
    xorq %rax,%rax       # sum = 0
    andq %rsi,%rsi       # Set CC
    jmp     test         # Goto test
loop:   mrmovq (%rdi),%r10   # Get *start
    addq %r10,%rax       # Add to sum
    addq %r8,%rdi        # start++
    subq %r9,%rsi        # count--.  Set CC
test:   jne     loop         # Stop when 0
    ret                  # Return

# Stack starts here and grows to lower addresses
    .pos 0x200
stack:
```

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data

Y86 Assembly example (2/2)

```
# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp      # Set up stack pointer
    call main       # Execute main program
    halt            # Terminate program

# Array of 4 elements
    .align 8
array:  .quad 0x000d000d000d
    .quad 0x00c000c000c0
    .quad 0x0b000b000b00
    .quad 0xa000a000a000

main:   irmovq array,%rdi
    irmovq $4,%rsi
    call sum        # sum(array, 4)
    ret

# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:    irmovq $8,%r8        # Constant 8
    irmovq $1,%r9        # Constant 1
    xorq %rax,%rax       # sum = 0
    andq %rsi,%rsi       # Set CC
    jmp     test         # Goto test
loop:   mrmovq (%rdi),%r10   # Get *start
    addq %r10,%rax       # Add to sum
    addq %r8,%rdi        # start++
    subq %r9,%rsi        # count--.  Set CC
test:   jne     loop         # Stop when 0
    ret                  # Return

# Stack starts here and grows to lower addresses
    .pos 0x200
stack:
```

Initialization

Program data

Main function

sum function

Placement of stack

Lets to be a human assembler (compiler)

```
# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp      # Set up stack pointer
    call main               # Execute main program
    halt                    # Terminate program

# Array of 4 elements
    .align 8
array:  .quad 0x000d000d000d
    .quad 0x00c000c000c0
    .quad 0x0b000b000b00
    .quad 0xa000a000a000

main:   irmovq array,%rdi
    irmovq $4,%rsi
    call sum                # sum(array, 4)
    ret

# long sum(long *start, long count)
# start in %rdi, count in %rsi
sum:    irmovq $8,%r8        # Constant 8
    irmovq $1,%r9           # Constant 1
    xorq %rax,%rax          # sum = 0
    andq %rsi,%rsi          # Set CC
    jmp     test            # Goto test
loop:   mrmovq (%rdi),%r10   # Get *start
    addq %r10,%rax          # Add to sum
    addq %r8,%rdi           # start++
    subq %r9,%rsi           # count--.  Set CC
test:   jne    loop          # Stop when 0
    ret                     # Return

# Stack starts here and grows to lower addresses
    .pos 0x200
stack:
```

Convert initial lines to object codes (1/5)

```
1 # Execution begins at address 0
2     .pos 0
3     irmovq stack, %rsp          # Set up stack pointer
4     call main                   # Execute main program
5     halt                        # Terminate program
```

- Line2: what is the meaning of .pos 0
  - It means the beginning address of program is 0x0000
- Line2: What is the address of line2?
  - 0x0000. Why?
- Line3: How many bytes are used for irmovq?
  - 10 bytes
- Line3: What is the meaning of stack?
  - It is a label, in other words memory address

Convert initial lines to object codes (2/5)

```
1 # Execution begins at address 0
2     .pos 0
3     irmovq stack, %rsp          # Set up stack pointer
4     call main                   # Execute main program
5     halt                        # Terminate program
```

- Line3: Do we know address of stack?
  - Not yet. Later we will know.
  - Two passes are used for assemblers for determining exact memory address for labels, variables and functions.
- Look at Lines 34-35: .pos 0x200
  - → Address for stack is 0x200
  - → 8bytes notation: 0x 00 00 00 00 00 00 02 00
  - → **Little endian** notation: 0x 00 02 00 00 00 00 00 00

Convert initial lines to object codes (3/5)

```
1 # Execution begins at address 0
2     .pos 0
3     irmovq stack, %rsp          # Set up stack   er
4     call main                   # Execute mai    am
5     halt                        # Terminate pr
```

%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

irmovq V, rB

3	0	F	rB	V

- Line3: What is binary for irmovq stack, %rsp
  - 0x 30 F4 “address for stack (8 bytes)”

→ 0x 30 F4 00 02 00 00 00 00 00 00

Convert initial lines to object codes (4/5)

```
1 # Execution begins at address 0
2       .pos 0
3       irmovq stack, %rsp          # Set up stack pointer
4       call main                   # Execute main program
5       halt                        # Terminate program
```

- Line4: What is the address?
  - 0x000a : 0000 + 10 (size of irmovq)
- Line4: What is length for call?
  - 9 bytes
- Line4: what is main?
  - It is a label. We do not know the address not yet.
- Line4: Binary for main
  - 0x: 80 “address for main (8bytes)”

Convert initial lines to object codes (5/5)

```
1 # Execution begins at address 0
2     .pos 0
3     irmovq stack, %rsp          # Set up stack pointer
4     call main                   # Execute main program
5     halt                        # Terminate program
```

- Line5: what is the address?
  - 0x0013 : 0x000a + 9 (size of main)

- Line5: what is the binary for halt?
  - 0x0013: 00

So far,

```
# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp          # Set up stack pointer
    call main                   # Execute main program
    halt                        # Terminate program
```

- Object codes (Address and binary codes)

```
                                   | # Execution begins at address 0
0x0000:                            | .pos 0
0x0000: 30f40002000000000000       | irmovq stack, %rsp # Set up stack pointer
0x000a: 80"8bytes addr for main"   | call main # Execute main program
0x0013: 00                         | halt # Terminate program
                                   |
```

Convert data to binaries (1/3)

```
7  # Array of 4 elements
8      .align 8
9  array:  .quad 0x000d000d000d
10     .quad 0x00c000c000c0
11     .quad 0x0b000b000b00
12     .quad 0xa000a000a000
```

- Line 8: what is the address?
  - 0x0014 (0x0013 + 1)
- Line 8: what is the meaning of .align 8?
  - Data should be fit into just one 8-bytes
  - Beginning address should be multiple of 8
- Line9: what is the address for array?
  - 0x0018 (why? Due to the alignment)
  - Thus, there are nothing (or garbage) in 0x0014-0x0017

Convert data to binaries (2/3)

```
7  # Array of 4 elements
8      .align 8
9  array:   .quad 0x000d000d000d
10     .quad 0x00c000c000c0
11     .quad 0x0b000b000b00
12     .quad 0xa000a000a000
```

- Line 9: what is the meaning .quad?
  - Data size is quad word (1 word = 2 bytes in x86, y86)

  → .quad == 8 bytes of data
- Line 9: what is the value?
  - 0x 00 0d 00 0d 00 0d
- Line 9: What is the binary?
  - 0x0018: 0d 00 0d 00 0d 00 00 00 (little endian)

Convert data to binaries (3/3)

```
7   # Array of 4 elements
8       .align 8
9   array:   .quad 0x000d000d000d
10      .quad 0x00c000c000c0
11      .quad 0x0b000b000b00
12      .quad 0xa000a000a000
```

- Line10: What is the address?
  - 0x 0020 (0x 0018 + 8)
- Line10: How to convert binaries?
  - 0x 0020: c0 00 c0 00 c0 00 00 00 00
- Lines11-12: How to convert binaries?
  - 0x 0028: 00 0b 00 0b 00 0b 00 00 00
  - 0x 0030: 00 a0 00 a0 00 a0 00 00 00

So far

```
# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp      # Set up stack pointer
    call main               # Execute main program
    halt                    # Terminate program

# Array of 4 elements
    .align 8
array:  .quad 0x000d000d000d
    .quad 0x00c000c000c0
    .quad 0x0b000b000b00
    .quad 0xa000a000a000
```

- Object codes (Address and binary codes)

```
                                     | # Execution begins at address 0
0x0000:                              | .pos 0
0x0000: 30f40002000000000000         | irmovq stack, %rsp # Set up stack pointer
0x000a: 80"8bytes addr for main"     | call main # Execute main program
0x0013: 00                           | halt # Terminate program
                                     |
                                     | # Array of 4 elements
0x0014:                              | .align 8
0x0018: 0d000d000d                   | array: .quad 0x000d000d000d
0x0020: c000c000c0                   | .quad 0x00c000c000c0
0x0028: 000b000b000b                 | .quad 0x0b000b000b00
0x0030: 00a000a000a0                 | .quad 0xa000a000a000
```

Convert main() to binaries (1/3)

```
14 main:    irmovq array,%rdi
15      irmovq $4,%rsi
16      call sum                # sum(array, 4)
17      ret
```

- Line14: what is address for main?
  - main: 0x0038 (0x 0030 + 8 bytes quad word)
  - Now we can get the values for line 4
    - 8 bytes in little endian: 38 00 00 00 00 00 00 00
- Line14: what is address for array?
  - The address for array is the same with the first data in Line 9
  - 0x 0018 → 18 00 00 00 00 00 00 00 (little endian)

Convert main() to binaries (2/3)

```
14 main:   irmovq array,%rdi
15     irmovq $4,%rsi
16     call sum                # sum(array, 4)
17     ret
```

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

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

`irmovq V, rB`

3	0	F	rB	V

- Line14: what is the binary for irmovq?
  - 0x0038: 30 F7 18 00 00 00 00 00 00 00
- Line 15: what is address?
  - 0x0042 (0x 0038 + 10 bytes)
- Line 15: what is the binary?
  - 0x0042: 30 F6 04 00 00 00 00 00 00 00

Convert main() to binaries (3/3)

```
14 main:    irmovq array,%rdi
15      irmovq $4,%rsi
16      call sum               # sum(array, 4)
17      ret
```

- Line16: what is address?
  - 0x004c (0x0042 + 10 bytes)
- Line16: what is the address for sum?
  - We do not know yet.
- Line16: what is the binary?
  - 0x 004c: 80 “addr for sum (8 bytes)
- Line17: what is the binary for ret?
  - 0x 0055: 90

So far

- We resolve the address for main

```
                                 | # Execution begins at address 0
0x0000:                          | .pos 0
0x0000: 30f40002000000000000     | irmovq stack, %rsp # Set up stack pointer
0x000a: 803800000000000000       | call main # Execute main program
0x0013: 00                       | halt # Terminate program
                                 |
                                 | # Array of 4 elements
0x0014:                          | .align 8
0x0018: 0d000d000d               | array: .quad 0x000d000d000d
0x0020: c000c000c0               | .quad 0x00c000c000c0
0x0028: 000b000b000b             | .quad 0x0b000b000b00
0x0030: 00a000a000a0             | .quad 0xa000a000a000
                                 |
0x0038: 30f71800000000000000     | main: irmovq array,%rdi
0x0042: 30f60400000000000000     | irmovq $4,%rsi
0x004c: 80"addr to sum (8bytes)" | call sum # sum(array, 4)
0x0055: 90                       | ret
                                 |
```

Convert sum() to binaries (1/6)

```
21 sum:    irmovq $8,%r8         # Constant 8
22         irmovq $1,%r9         # Constant 1
23         xorq %rax,%rax        # sum = 0
24         andq %rsi,%rsi        # Set CC
25         jmp     test          # Goto test
26 loop:   mrmovq (%rdi),%r10    # Get *start
27         addq %r10,%rax        # Add to sum
28         addq %r8,%rdi         # start++
29         subq %r9,%rsi         # count--.  Set CC
30 test:   jne     loop          # Stop when 0
31         ret                   # Return
```

- Line21: what is address?
  - 0x0056 (0x0055 + 1 (size of ret))
- Line21: what is binary?
  - 0x0056: 30 F8 08 00 00 00 00 00 00 00
- Line22: what is binary?
  - 0x0060: 40 F9 01 00 00 00 00 00 00 00

Convert sum() to binaries (2/6)

```
21 sum:    irmovq $8,%r8        # Constant 8
22         irmovq $1,%r9        # Constant 1
23         xorq %rax,%rax       # sum = 0
24         andq %rsi,%rsi       # Set CC
25         jmp     test         # Goto test
26 loop:   mrmovq (%rdi),%r10   # Get *start
27         addq %r10,%rax       # Add to sum
28         addq %r8,%rdi        # start++
29         subq %r9,%rsi        # count--.  Set CC
30 test:   jne     loop         # Stop when 0
31         ret                  # Return
```

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

xorq rA, rB	6	3	rA	rB

- Line23: what is address?
  - 0x006A (0x0060 + 10 (size of irmovq))
- Line23: what is binary?
  - 0x006A: 63 00
- Line24: what is binary?
  - 0x006C: 62 66

andq rA, rB	6	2	rA	rB

Convert sum() to binaries (3/6)

```
21 sum:    irmovq $8,%r8         # Constant 8
22         irmovq $1,%r9         # Constant 1
23         xorq %rax,%rax        # sum = 0
24         andq %rsi,%rsi        # Set CC
25         jmp    test           # Goto test
26 loop:   mrmovq (%rdi),%r10    # Get *start
27         addq %r10,%rax        # Add to sum
28         addq %r8,%rdi         # start++
29         subq %r9,%rsi         # count--.  Set CC
30 test:   jne    loop           # Stop when 0
31         ret                   # Return
```

`jXX` Dest

7	fn	Dest

- Line25: what is address?
  - 0x006E (0x006a + 2 (size of adnq))
- Line25: what is address for test?
  - Not yet know, later
- Line25: what is binary?
  - 0x006E: 70 “addr for test”

Convert sum() to binaries (4/6)

```
21  sum:    irmovq $8,%r8         # Constant 8
22          irmovq $1,%r9         # Constant 1
23          xorq %rax,%rax        # sum = 0
24          andq %rsi,%rsi        # Set CC
25          jmp     test          # Goto test
26  loop:   mrmovq (%rdi),%r10    # Get *start
27          addq %r10,%rax        # Add to sum
28          addq %r8,%rdi         # start++
29          subq %r9,%rsi         # count--.
30  test:   jne     loop          # Stop when 0
31          ret                   # Return
```

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

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

mrmovq D (rB), rA	5	0	rA	rB	D

- Line26: what is address?
  - 0x0077 (0x006e + 9(size of jmp))
  - Note that loop has this address.
- Line26: what are two registers?
  - %rdi 7 as rB, %r10 A as rA
- Line27: what is the binary?
  - 0x0077: 50 A7 00 00 00 00 00 00 (D equals to 0)

Convert sum() to binaries (5/6)

```
21  sum:    irmovq $8,%r8        # Constant 8
22          irmovq $1,%r9        # Constant 1
23          xorq %rax,%rax       # sum = 0
24          andq %rsi,%rsi       # Set CC
25          jmp     test         # Goto test
26  loop:   mrmovq (%rdi),%r10   # Get *start
27          addq %r10,%rax       # Add to sum
28          addq %r8,%rdi        # start++
29          subq %r9,%rsi        # count--.
30  test:   jne     loop         # Stop when 0
31          ret                  # Return
```

%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

- Line27: what is the binary?
  - 0x0081: 60 AE
- Line28: what is the binary?
  - 0x0083: 60 87
- Line29: what is the binary?
  - 0x0085: 61 96

Convert sum() to binaries (6/6)

```
21 sum:    irmovq $8,%r8          # Constant 8
22         irmovq $1,%r9          # Constant 1
23         xorq %rax,%rax         # sum = 0
24         andq %rsi,%rsi         # Set CC
25         jmp     test           # Goto test
26 loop:   mrmovq (%rdi),%r10     # Get *start
27         addq %r10,%rax         # Add to sum
28         addq %r8,%rdi          # start++
29         subq %r9,%rsi          # count--.
30 test:   jne     loop           # Stop when 0
31         ret                    # Return
```

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

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

jXX Dest

7	fn	Dest

- Line30: what is the address?
  - 0x0087: 0x0085 + 2 (size of subq)
- Line30: what is the addr for loop?
  - 0x0077
- Line30: what is the binary?
  - 0x0087: 74 77 00 00 00 00 00 00 00

Binary code for sum()

```
                              | # long sum(long *start, long count)
                              | # start in %rdi, count in %rsi
0x0056: 30f80800000000000000  | sum: irmovq $8,%r8 # Constant 8
0x0060: 30f90100000000000000  | irmovq $1,%r9 # Constant 1
0x006a: 6300                  | xorq %rax,%rax # sum = 0
0x006c: 6266                  | andq %rsi,%rsi # Set CC
0x006e: 708700000000000000    | jmp test # Goto test
0x0077: 50a70000000000000000  | loop: mrmovq (%rdi),%r10 # Get *start
0x0081: 60a0                  | addq %r10,%rax # Add to sum
0x0083: 6087                  | addq %r8,%rdi # start++
0x0085: 6196                  | subq %r9,%rsi # count--. Set CC
0x0087: 748700000000000000    | test: jne loop # Stop when 0
0x0090: 90                    | ret # Return
                              |
                              | # Stack starts here, grows to lower addresses
```

Y86 code (1/2)

```c
long sum(long *start, long count)
{
  long sum = 0;
  while (count) {
    sum += *start;
    start ++;
    count--;
  }
  return sum;
}
```

```
                                  | # Execution begins at address 0
0x0000:                           | .pos 0
0x0000: 30f40002000000000000      | irmovq stack, %rsp # Set up stack pointer
0x000a: 803800000000000000        | call main # Execute main program
0x0013: 00                        | halt # Terminate program
                                  |
                                  | # Array of 4 elements
0x0014:                           | .align 8
0x0018: 0d000d000d                | array: .quad 0x000d000d000d
0x0020: c000c000c0                | .quad 0x00c000c000c0
0x0028: 000b000b000b              | .quad 0x0b000b000b00
0x0030: 00a000a000a0              | .quad 0xa000a000a000
|
0x0038: 30f71800000000000000      | main: irmovq array,%rdi
0x0042: 30f60400000000000000      | irmovq $4,%rsi
0x004c: 805600000000000000        | call sum # sum(array, 4)
0x0055: 90                        | ret
                                  |
```

Y86 code (2/2)

```
long sum(long *start, long count)
{
    long sum = 0;
    while (count) {
        sum += *start;
        start ++;
        count--;
    }
    return sum;
}
```

```
                              | # long sum(long *start, long count)
                              | # start in %rdi, count in %rsi
0x0056: 30f80800000000000000  | sum: irmovq $8,%r8 # Constant 8
0x0060: 30f90100000000000000  | irmovq $1,%r9 # Constant 1
0x006a: 6300                  | xorq %rax,%rax # sum = 0
0x006c: 6266                  | andq %rsi,%rsi # Set CC
0x006e: 708700000000000000    | jmp test # Goto test
0x0077: 50a70000000000000000  | loop: mrmovq (%rdi),%r10 # Get *start
0x0081: 60a0                  | addq %r10,%rax # Add to sum
0x0083: 6087                  | addq %r8,%rdi # start++
0x0085: 6196                  | subq %r9,%rsi # count--. Set CC
0x0087: 747700000000000000    | test: jne loop # Stop when 0
0x0090: 90                    | ret # Return
                              |
                              | # Stack starts here, grows to lower addresses
0x0091:                       | .pos 0x200
0x0200:                       | stack:
```

Summary

- Y86-64 Instruction Set Architecture
  - Similar state and instructions as x86-64
  - Simpler encodings
  - Somewhere between CISC and RISC

Q&A