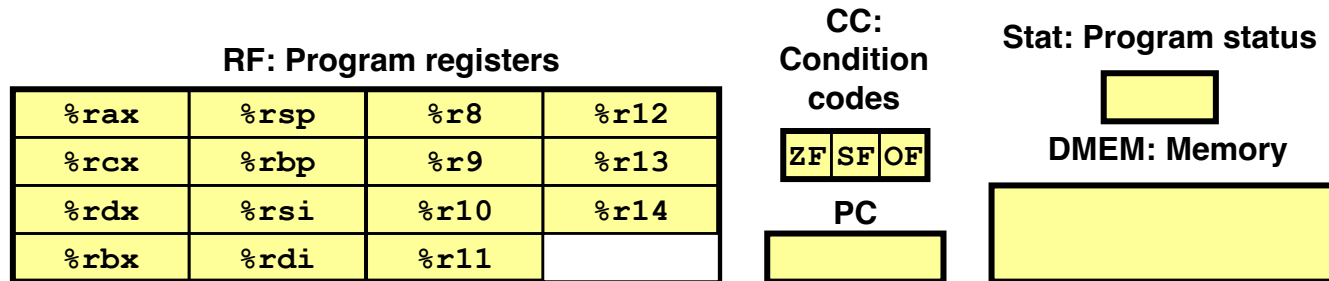


Using the Y86_64 tools for Lab 3

CSCI 237

Williams College

Review: 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 ISA

Byte	0	1	2	3	4	5	6	7	8	9	10
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						
mrmmovq 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 Tools and Tasks

- For Lab 4, we will complete 3 tasks:
 - Write small programs in Y86-64 to:
 - Iteratively sum the elements in a linked list
 - Recursively sum the elements in a linked list
 - Copy a chunk of memory from one location to another
- The only tool for creating Y86-64 code is the assembler
 - We must perform tasks that are usually done by the compiler!
- We will only be writing small programs in Y86-64, but this is exactly the type of thing you would do in the “real world”: optimize specific pieces of code by adding inline assembly

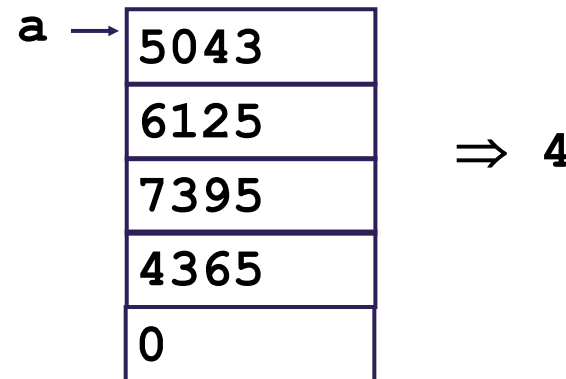
Possible Approaches to Writing Y86-64 Code

- Try to use C compiler as much as possible
 - Write code in C
 - Compile for x86-64 with `gcc -Og -S` or `gcc -O1 -S`
 - Transliterate resulting x86-64 into Y86-64
 - Reminder: What are some of the differences?
 - *Modern compilers make this more difficult...*

■ Coding Example

- Find number of elements in null-terminated list

```
long len(long a[]);
```



Y86-64 Code Generation Example

■ First try

- Write typical array code

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
    long len;
    for (len = 0; a[len]; len++);
    return len;
}
```

■ Problem

- Hard to do array indexing on Y86-64
 - Since we don't have scaled addressing modes

L3:

```
addq $1,%rax
cmpq $0, (%rdi,%rax,8)
jne L3
```

- Compile with `gcc -O1 -S`

Y86-64 Code Generation Example #2

■ Second Try

- Write C code that mimics expected Y86-64 code
- Don't explicitly use array

```
long len2(long *a)
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        len++;
        val = *(long *) ip;
    }
    return len;
}
```

■ Result

- Compiler generates exact same code as before!
- Compiler converts both versions into same intermediate form

```
L3:
    addq $1,%rax
    cmpq $0, (%rdi,%rax,8)
    jne L3
```

Y86-64 Code Generation Example #3

■ Third Try

- Flex your ASM muscles and write it yourself!

```
len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    irmovq $0, %rax         # len = 0
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx        # Test val
    je Done                 # If zero, goto Done

Loop:
    addq %r8, %rax          # len++
    addq %r9, %rdi         # a++
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx        # Test val
    jne Loop               # If !0, goto Loop

Done:
    ret
```

Register	Use
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

Y86-64 Sample Program Structure #1

```
init:                # Initialization
    . . .
    call Main
    halt

    .align 8         # Program data
array:
    . . .

Main:                # Main function
    . . .
    call len
    . . .

len:                 # len function

    .pos 0x100      # Placement of stack

Stack:
    . . .
```

- Must set up stack
 - Where located
 - Pointer values
 - Make sure don't overwrite code!
- Must initialize data array
- init calls Main, which in turn calls len
 - (details on following slides)
- This is a good model for lab 3!

Y86-64 Program Structure #2

```
init:
    .pos 0                # Start at address 0
    irmovq Stack, %rsp   # Set up stack pointer
    call Main            # Execute main program
    halt                # Terminate

# Array of 4 elements + terminating 0
    .align 8
array:
    .quad 0x000d000d000d000d
    .quad 0x00c000c000c000c0
    .quad 0x0b000b000b000b00
    .quad 0xa000a000a000a000
    .quad 0
```

- Program starts at address 0 (`.pos 0`)
- Must set up stack and stack pointer (`.pos 0x100`)
- Must initialize data array
- Can use symbolic names (labels)

Y86-64 Program Structure #3

Main:

```
    irmovq array,%rdi
    call len          #len(array)
    ret
```

- Set up call to len
 - Follow x86-64 procedure conventions
 - Push array address as argument

Assembling Y86-64 Program

```
unix> yas len.yo
```

- Generates “object code” file `len.yo`
 - Actually looks like disassembler output
 - Notice the byte encodings

```
0x054:          | len:
0x054: 30f80100000000000000 |   irmovq $1, %r8           # Constant 1
0x05e: 30f90800000000000000 |   irmovq $8, %r9           # Constant 8
0x068: 30f00000000000000000 |   irmovq $0, %rax          # len = 0
0x072: 50270000000000000000 |   mrmovq (%rdi), %rdx       # val = *a
0x07c: 6222             |   andq %rdx, %rdx          # Test val
0x07e: 73a00000000000000000 |   je Done                   # If zero, goto Done
0x087:             | Loop:
0x087: 6080             |   addq %r8, %rax           # len++
0x089: 6097             |   addq %r9, %rdi           # a++
0x08b: 50270000000000000000 |   mrmovq (%rdi), %rdx       # val = *a
0x095: 6222             |   andq %rdx, %rdx          # Test val
0x097: 74870000000000000000 |   jne Loop                  # If !0, goto Loop
0x0a0:             | Done:
0x0a0: 90               |   ret
```

Simulating Y86-64 Program

```
unix> yis len.yo
```

- Instruction set simulator
 - Computes effect of each instruction on processor state
 - Prints changes in state from original

```
Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
```

```
Changes to registers:
```

%rax:	0x0000000000000000	0x0000000000000004
%rsp:	0x0000000000000000	0x0000000000000100
%rdi:	0x0000000000000000	0x0000000000000038
%r8:	0x0000000000000000	0x0000000000000001
%r9:	0x0000000000000000	0x0000000000000008

```
Changes to memory:
```

0x00f0:	0x0000000000000000	0x0000000000000053
0x00f8:	0x0000000000000000	0x0000000000000013

Lab 3 Starter Code

```
# Initial code
.init
    irmovq Stack,%rsp
    XXX
    call sum_list
    halt

# Sample linked list
.align 8
ele1:
    .quad 0x00a
    .quad ele2

ele2:
    .quad 0x0b0
    .quad ele3

ele3:
    .quad 0xc00
    .quad 0

# long sum_list(list_ptr ls)
# ls in %rdi
sum_list:
    XXX
    ret                # return

.pos 0x100
Stack:
```

Hint: `andq` in Y86

- **`andq`** performs a bitwise AND operation between two operands and sets the **ZF** (zero flag) condition code
 - If the result of the `andq` operation is zero, the ZF is set to 1
 - If the result of the `andq` operation is non-zero, the ZF is set to 0
- When you want to test for null pointers, you can use `andq` and check the value of the ZF after the `andq` operation
- **`je`** only jumps if the ZF is set to 1 (which would mean the result of the `andq` operation is zero and the pointer was null)
- In x86, we would usually use the `testq` operation instead of `andq`, which is more intuitive. But since the result of a bitwise AND of a value with itself is the original value, `andq` in this particular usage would have identical behavior to `testq`