

# Lecture 7: ALU operations, arithmetic

David Hovemeyer

September 11, 2024

601.229 Computer Systems Fundamentals



# Writing x86-64 assembly code

# Getting started with x86-64 assembly

- ▶ Today we're beginning our detailed look at programming in x86-64 assembly language
- ▶ One challenge in learning assembly program is knowing how to get started
  - ▶ What does a minimal program look like?
  - ▶ How to define variables, do I/O, etc.
- ▶ Today's sample programs will be posted on the course web page (alu.zip)
  - ▶ Feel free to use them as a reference, modify them, etc.

# A few essentials

- ▶ Use `.S` file extension for assembly code
  - ▶ Will be run through the C preprocessor, can use C style comments and `#define` to define named constants
- ▶ Use `gcc` to assemble `.S` file into object code (`.o` file)
- ▶ Use `gcc` to link object files (`.o`) into executable
- ▶ Overall, process is similar to developing a C program

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata                                <-- read-only data section

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n" <-- NUL terminated string constant

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text                                <-- code goes in .text section

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret
```



# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main                                <-- make `main' visible to other modules
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp                <-- align stack pointer
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret
```

<-- no vector arguments to printf

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi      <-- first arg is ptr to message string
        call printf

        addq $8, %rsp
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf                <-- call printf!

        addq $8, %rsp
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp          <-- restore stack pointer
        ret
```

# Hello, world

```
/* hello.S */

.section .rodata

sHelloMsg: .string "Hello, world\n"

.section .text

        .globl main
main:
        subq $8, %rsp
        movl $0, %eax
        movq $sHelloMsg, %rdi
        call printf

        addq $8, %rsp
        ret                                <-- return from main, ends program
```

# Assembling, linking, executing

```
$ gcc -c -no-pie -o hello.o hello.S
$ gcc -no-pie -o hello hello.o
$ ./hello
Hello, world
```

Note that the `-no-pie` option disables support for position-independent code (which would require additional magic in the assembly source code)



# ALU operations

# ALU operations

- ▶ ALU = “Arithmetic Logic Unit”
- ▶ An ALU is a hardware component within the CPU that does computations (of various kinds) on data values
  - ▶ Addition/subtraction
  - ▶ Logical operations (shifts, bitwise and/or/negation), etc.
- ▶ So, ALU instructions are the ones that do computations on values
  - ▶ Typically, ALU operates only on integer values
  - ▶ CPU will typically have floating-point unit(s) for operations on FP values

# lea instruction

- ▶ lea stands for “Load Effective Address”
- ▶ Instructions that allow a memory reference as an operand generally do an *address computation*
  - ▶ E.g., `movl 12(%rdx,%rsi,4), %eax`
  - ▶ Computed address (for source memory location) is  $\%rdx + (\%rsi \times 4) + 12$
- ▶ The lea instruction computes a memory address, but does *not* access a memory location
  - ▶ E.g., `leaq 12(%rdx,%rsi,4), %rdi`
  - ▶ Keep in mind we’re not obligated to use the computed address as an address — we can just use it as an integer
- ▶ In general, lea can do integer computations of the form  $p + (qS) + r$  where  $S$  is 0, 1, 2, 4, or 8
- ▶ lea does not set condition codes (e.g., on overflow)

# leaq example code

```
/* leaq_example.S */  
.section .rodata  
sFmt: .string "Result is: %lu\n"  
  
.section .text  
    .globl main  
main:  
    subq $8, %rsp  
    movl $0, %eax  
    movq $sFmt, %rdi  
    movq $1000, %r10  
    movq $3, %r11  
    leaq 15(%r10,%r11,8), %rsi  
    call printf  
    addq $8, %rsp  
    ret
```

# leaq example code

```
/* leaq_example.S */  
.section .rodata  
sFmt: .string "Result is: %lu\n"  
  
.section .text  
    .globl main  
main:  
    subq $8, %rsp  
    movl $0, %eax  
    movq $sFmt, %rdi  
    movq $1000, %r10  
    movq $3, %r11  
    leaq 15(%r10,%r11,8), %rsi      %rsi <- 1000 + 3*8 + 15  
    call printf  
    addq $8, %rsp  
    ret
```

# Result of leaq example program

```
$ gcc -c -no-pie -o leaq_example.o leaq_example.S
$ gcc -no-pie -o leaq_example leaq_example.o
$ ./leaq_example
Result is: 1039
```

# Clicker quiz!

Clicker quiz omitted from public slides

# Addition, subtraction

- ▶ add and sub instructions add and subtract integer values
- ▶ Two operands, second operand modified to store the result
  - ▶ Note that either operand (but not both) could be a memory reference
- ▶ E.g.,

```
movq $1, %r9
movq $2, %r10
addq %r9, %r10
/* %r10 now contains the value 3 */
```

- ▶ Overflow is possible!
  - ▶ Can detect using condition codes



# Addition/subtraction example program

```
/* addsub.S */
.section .rodata
sPrompt: .string "Enter an integer value: "
sInputFmt: .string "%u"
sFmt: .string "Result is %u\n"

.section .data
val: .space 4

.section .text
.globl main
main:
    subq $8, %rsp

    movl $0, %eax
    movq $sPrompt, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    movq $val, %rsi
    call scanf

    addl $10, val
    subl $2, val

    movl $0, %eax
    movq $sFmt, %rdi
    movl val, %esi
    call printf

    addq $8, %rsp

    ret
```

# Addition/subtraction example program

```
/* addsub.S */
.section .rodata
sPrompt: .string "Enter an integer value: "
sInputFmt: .string "%u"
sFmt: .string "Result is %u\n"

.section .data
val: .space 4      <- global variable

.section .text
.globl main
main:
    subq $8, %rsp

    movl $0, %eax
    movq $sPrompt, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    movq $val, %rsi
    call scanf

    addl $10, val
    subl $2, val

    movl $0, %eax
    movq $sFmt, %rdi
    movl val, %esi
    call printf

    addq $8, %rsp

    ret
```

# Addition/subtraction example program

```
/* addsub.S */
.section .rodata
sPrompt: .string "Enter an integer value: "
sInputFmt: .string "%u"
sFmt: .string "Result is %u\n"

.section .data
val: .space 4

.section .text
.globl main
main:
    subq $8, %rsp

    movl $0, %eax
    movq $sPrompt, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    movq $val, %rsi    <- pass address of val to scanf
    call scanf

    addl $10, val
    subl $2, val

    movl $0, %eax
    movq $sFmt, %rdi
    movl val, %esi
    call printf

    addq $8, %rsp

    ret
```

# Addition/subtraction example program

```
/* addsub.S */
.section .rodata
sPrompt: .string "Enter an integer value: "
sInputFmt: .string "%u"
sFmt: .string "Result is %u\n"

.section .data
val: .space 4

.section .text
.globl main
main:
    subq $8, %rsp

    movl $0, %eax
    movq $sPrompt, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    movq $val, %rsi
    call scanf

    addl $10, val      <- add 10 to val
    subl $2, val

    movl $0, %eax
    movq $sFmt, %rdi
    movl val, %esi
    call printf

    addq $8, %rsp

    ret
```

# Addition/subtraction example program

```
/* addsub.S */
.section .rodata
sPrompt: .string "Enter an integer value: "
sInputFmt: .string "%u"
sFmt: .string "Result is %u\n"

.section .data
val: .space 4

.section .text
.globl main
main:
    subq $8, %rsp

    movl $0, %eax
    movq $sPrompt, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    movq $val, %rsi
    call scanf

    addl $10, val
    subl $2, val      <- subtract 2 from val

    movl $0, %eax
    movq $sFmt, %rdi
    movl val, %esi
    call printf

    addq $8, %rsp

    ret
```

# Addition/subtraction example program

```
/* addsub.S */
.section .rodata
sPrompt: .string "Enter an integer value: "
sInputFmt: .string "%u"
sFmt: .string "Result is %u\n"

.section .data
val: .space 4

.section .text
.globl main
main:
    subq $8, %rsp

    movl $0, %eax
    movq $sPrompt, %rdi
    call printf

    movl $0, %eax
    movq $sInputFmt, %rdi
    movq $val, %rsi
    call scanf

    addl $10, val
    subl $2, val

    movl $0, %eax
    movq $sFmt, %rdi
    movl val, %esi
    call printf      <- print value in val

    addq $8, %rsp

    ret
```

# Addition/subtraction example program

```
$ gcc -c -no-pie -o addsub.o addsub.S
$ gcc -no-pie -o addsub addsub.o
$ ./addsub
Enter an integer value: 11
Result is 19
```

# Increment, decrement

- ▶ `inc` and `dec` instructions increment or decrement by 1
- ▶ One operand, can be either register or memory
- ▶ Examples:

```
incq %rax    /* increment %rax */  
incl 4(%rbp) /* increment 32 bit value at addr %rbp+4 */  
decq %rdi    /* decrement %rdi */
```

- ▶ Overflow is possible, check condition codes



# Shifts

- ▶ Left shift: `shl`
- ▶ Right shift: `sar` (arithmetic), `shr` (logical)
  - ▶ `sar` shifts in the value of the sign bit, `shr` shifts in zeroes
- ▶ Examples (see `shift_example.S` in `alu.zip`):

```
movl $0xFFFF0000, %esi
shll $1, %esi          /* %esi set to 0xFFFE0000 */
movl $0xFFFF0000, %esi
sarl $1, %esi          /* %esi set to 0xFFFF8000 */
movl $0xFFFF0000, %esi
shrl $1, %esi          /* %esi set to 0x7FFF8000 */
```

- ▶ Shifts commonly used to multiply or divide by power of two
  - ▶ Left shift one position → multiply by 2
  - ▶ Right shift one position → divide by 2 (and discard remainder)

# Clicker quiz!

Clicker quiz omitted from public slides

# Bitwise logical operations

- ▶ Two-operand logical operations: and, or, xor
- ▶ Unary logical operation: not
- ▶ Examples (see logic\_example.S in alu.zip):

```
/* Note: 0x30 = 00110000b,
          0x50 = 01010000b */
movb $0x30, %al; movb $0x50, %bl
andb %bl, %al      /* set %al=0x10 (00010000b) */
movb $0x30, %al; movb $0x50, %bl
orb %bl, %al       /* set %al=0x70 (01110000b) */
movb $0x30, %al; movb $0x50, %bl
xorb %bl, %al      /* set %al=0x60 (01100000b) */
movb $0x30, %al
notb %al           /* set %al=0xCF (11001111b) */
```

# Multiplication

- ▶ Two forms of `imul` instruction
- ▶ Two operand: multiply operands and truncate
  - ▶ Example:

```
imulq %rdi, %rsi /* set %rsi to %rdi * %rsi,  
                  truncated to 64 bits */
```

- ▶ One operand: multiply 64 bit operand and value in `%rax`, 128-bit result in `%rdx:%rax`
  - ▶ Signed (`imulq`) and unsigned (`mulq`) variants
  - ▶ Example:

```
mulq %rdi /* set %rdx:%rax to unsigned product  
           %rax * %rdi */
```

# Division

- ▶ `idivq` and `divq`: signed and unsigned integer division
- ▶ 128-bit dividend in `%rdx:%rax`, 64 bit quotient in `%rax`, 64 bit remainder in `%rdx`
  - ▶ For a 64 bit dividend, set `%rdx` to 0 (unsigned division) or the replication of the sign bit of `%rax` (`ctqo` instruction replicates the sign bit of `%rax`)
- ▶ Example:

```
divq %r10          /* divide %rdx:%rax by %r10,  
                   put quotient in %rax,  
                   remainder in %rdx */
```

# Putting it all together

# Computing a weighted average

- ▶ Let's say you want to know your grade in the course
- ▶ Weighting is 55% assignments, 20% midterm exam, 20% final exam, 5% clicker quiz participation
- ▶ Example run:

```
Enter weight (0 when done): 55
Enter value: 84
Enter weight (0 when done): 20
Enter value: 89
Enter weight (0 when done): 20
Enter value: 93
Enter weight (0 when done): 5
Enter value: 100
Enter weight (0 when done): 0
Weighted average is 87
```

# Program outline (full code in weighted\_avg.S in alu.zip)

```
.section .rodata
    read-only strings

.section .bss
    zero-initialized global variables

.section .text
    .globl main
main:
    subq $8, %rsp

.LinputLoop:
    read weight
    if weight is 0, we're done
    read value
    multiply value by weight, add to sum
    add weight to sum of weights
    jmp .LinputLoop

.LdoneWithInput:
    divide sum by sum of weights
    print result

    addq $8, %rsp
    ret
```



# Read-only strings, global variables

```
.section .rodata

sWeightPrompt: .string "Enter weight (0 when done): "
sValuePrompt: .string "Enter value: "
sInputFmt: .string "%ld"
sResultMsg: .string "Weighted average is %ld\n"

.section .bss

valueIn: .space 8
weightIn: .space 8
sum: .space 8
weightSum: .space 8
```

# Loop body: read weight and value (end loop if weight=0)

```
/* read weight */
movl $0, %eax
movq $sWeightPrompt, %rdi
call printf
movl $0, %eax
movq $sInputFmt, %rdi
movq $weightIn, %rsi
call scanf

/* if weight is 0, we're done */
cmpq $0, weightIn
jz .LdoneWithInput

/* read value */
movl $0, %eax
movq $sValuePrompt, %rdi
call printf
movl $0, %eax
movq $sInputFmt, %rdi
movq $valueIn, %rsi
call scanf
```

## Loop body: update sum and weightSum variables

```
/* multiply value by weight, add to sum */  
movq weightIn, %r10  
movq valueIn, %r11  
imulq %r10, %r11  
addq %r11, sum  
  
/* add weight to sum of weights */  
movq weightIn, %r10  
addq %r10, weightSum
```

## After loop finished, compute weighted average and print

```
/* divide sum by sum of weights */  
movq $0, %rdx  
movq sum, %rax  
divq weightSum          /* quotient will be stored in %rax */  
  
/* print result */  
movq %rax, %rsi  
movl $0, %eax  
movq $sResultMsg, %rdi  
call printf
```