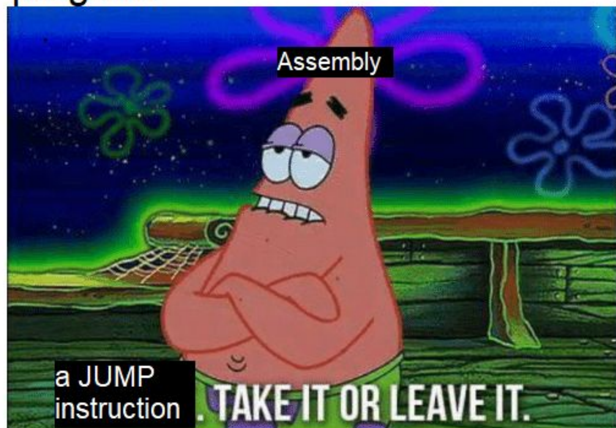
Me: man, I sure would like some nice control flow constructs for my program



CSE 351 Section 4

x86-64 Assembly

When you use gdb to see the assembly code of a program:

When you write high quality assembly code

Administrivia

- Lab 2:

  → Due **next** Friday (10/28/2022)!

  → Make sure all your phase answers are followed by a newline character.

- Homework:

  → HW 9 Due **TOMORROW** (10/21/2022)!

  → HW 10 Due **Monday** (10/24/2022)!

x86-64 Assembly

x86-64 Assembly

x86-64 is the primary 64-bit instruction set architecture (ISA) used by modern personal computers.

- It was developed by Intel and AMD and its 32-bit predecessor is called IA32.
- x86-64 is designed for complex instruction set computing (CISC), generally meaning it contains a larger set of more versatile and more complex instructions.

Other instruction sets include ARM (RISC) and PowerPC.

Data and Instructions

For this course, we will utilize only a small subset of x86-64's instruction set and omit floating point instructions. The subset of x86-64 instructions that we will use in this course take either one or two operands, usually in the form:

```
instruction operand1, operand2
```

There are three options for operands:

- Immediates: constants (e.g. `$0x400`)
- Registers: fast memory accessible to the CPU (e.g. `%rax, %edx`)
- Memory: memory addresses computed with `D(Rb, Ri, S)`
    - such as `0x400(%rdi, %rsi, 4)` = ***(%rdi + 4 * %rsi) + 0x400***

Address Computation

We can do more complicated memory accesses like so:

- `D(Rb, Ri, S)`
    - `Rb` - base register
    - `Ri` - index register
    - `S` - scale factor (1, 2, 4, 8)
    - `D` - displacement
    - Result is `Mem[Reg[Rb]+Reg[Ri]*S+D]`

So `0x400(%rdi, %rsi, 4)` evaluates to `%rdi + 4 * %rsi + 0x400`.
This is very useful for accessing elements in an array, and also for use in conjunction with `lea` (which does this address computation, but stores the raw result instead of accessing memory at the computed address).

Operand Size

The number of bytes of each operand used in an operation can be set using one of four suffixes. If `movb src, dst` copies 1 byte from src to dst, then:

- `movb src, dst` - copies 1 byte from src to dst
- `movw src, dst` - copies 2 bytes from src to dst
- `movl src, dst` - copies 4 bytes from src to dst
- `movq src, dst` - copies 8 bytes from src to dst

Midterm Reference Sheet

The reference sheet for the midterm is a great resource, especially for x86-64 (we handed out copies in class on Friday).

You can find it on the website here:
https://courses.cs.washington.edu/courses/cse351/22sp/exams/ref-mt.pdf

CSE 351 Reference Sheet (Midterm)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

IEEE 754 FLOATING-POINT STANDARD

Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

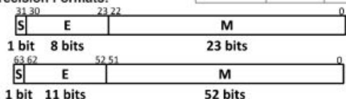
Bit fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

where Single Precision Bias = 127, Double Precision Bias = 1023.

IEEE 754 Symbols

E	M	Meaning
all zeros	all zeros	± 0
all zeros	non-zero	± denorm num
1 to MAX-1	anything	± norm num
all ones	all zeros	± ∞
all ones	non-zero	NaN

IEEE Single Precision and Double Precision Formats:



Assembly Instructions

mov a, b	Copy from a to b.
movs a, b	Copy from a to b with sign extension. Needs *two* width specifiers.
movz a, b	Copy from a to b with zero extension. Needs *two* width specifiers.
lea a, b	Compute address and store in b. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
push src	Push src onto the stack and decrement stack pointer.
pop dst	Pop from the stack into dst and increment stack pointer.
call <func>	Push return address onto stack and jump to a procedure.
ret	Pop return address and jump there.
add a, b	Add from a to b and store in b (and sets flags).
sub a, b	Subtract a from b (compute b–a) and store in b (and sets flags).
imul a, b	Multiply a and b and store in b (and sets flags).
and a, b	Bitwise AND of a and b, store in b (and sets flags).
sar a, b	Shift value of b *right* (*arithmetic*) by a bits, store in b (and sets flags).
shr a, b	Shift value of b *right* (*logical*) by a bits, store in b (and sets flags).
shl a, b	Shift value of b *left* by a bits, store in b (and sets flags).
cmp a, b	Compare b with a (compute b–a and set condition codes based on result).
test a, b	Bitwise AND of a and b and set condition codes based on result.
jmp <label>	Unconditional jump to address.
j* <label>	Conditional jump based on condition codes (*more on next page*).
set* a	Set byte a to 0 or 1 based on condition codes.

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	b $>_U$ a
jb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	b $<_U$ a

Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
%rax	Return value – **Caller** saved	%eax	%ax	%al
%rbx	**Callee** saved	%ebx	%bx	%bl
%rcx	Argument #4 – **Caller** saved	%ecx	%cx	%cl
%rdx	Argument #3 – **Caller** saved	%edx	%dx	%dl
%rsi	Argument #2 – **Caller** saved	%esi	%si	%sil
%rdi	Argument #1 – **Caller** saved	%edi	%di	%dil
%rsp	Stack Pointer	%esp	%sp	%spl
%rbp	**Callee** saved	%ebp	%bp	%bpl
%r8	Argument #5 – **Caller** saved	%r8d	%r8w	%r8b
%r9	Argument #6 – **Caller** saved	%r9d	%r9w	%r9b
%r10	**Caller** saved	%r10d	%r10w	%r10b
%r11	**Caller** saved	%r11d	%r11w	%r11b
%r12	**Callee** saved	%r12d	%r12w	%r12b
%r13	**Callee** saved	%r13d	%r13w	%r13b
%r14	**Callee** saved	%r14d	%r14w	%r14b
%r15	**Callee** saved	%r15d	%r15w	%r15b

Sizes

C type	x86-64 suffix	Size (bytes)
char	b	1
short	w	2
int	l	4
long	q	8

Interpreting Instructions

What do the following assembly instructions do?

X86-64 instruction	English equivalent
movq $351, %rax	Move the number 351 into 8-byte (quad) register “rax”
addq %rdi, %rsi	
movq (%rdi), %r8	
leaq (%rax,%rax,8), %rax	

Functions (briefly)

Similar to C - functions take arguments and can return a value.

Arguments:

- First argument is stored in **`%rdi`**, second in **`%rsi`**, third in **`%rdx`**.
- Arguments have to be copied into registers before the function is called.

Return:

- By convention, **`%rax`** is used for the return value.

More on this (function calls, more arguments, etc.) in lecture!

Exercise!

Exercise 1

Symbolically, what does the following code return? Remember, register %rax is used to store the return value.

```
movl (%rdi), %eax             # %rdi -> x    *x
leal (%eax,%eax,2), %eax      # %rax -> r    *x * 3
addl %eax, %eax                              (*x * 3) * 2
andl %esi, %eax               # %esi -> y    (*x * 6) & y
subl %esi, %eax                              ((*x * 6) & y) - y
ret
```

Conditionals

Condition Codes

Condition codes include the zero (ZF), sign (SF), carry (unsigned overflow, CF), and (signed, OF) overflow flags. They are stored on the processor in their own register.

- They are implicitly set by arithmetic operations:
  - `addq src, dst`
    - `r = dst + src` (result used to set flags)
- There are also instructions to only set the condition codes:
  - `cmp a, b`
    - `r = b - a` (result sets flags, but is not stored)
  - `test a, b`
    - `r = a & b` (result sets flags, but is not stored)

Control Flow

The condition codes are often used in combination with j* (jump) and set* instructions.

These instructions take one operand and "change the instruction pointer" (j*) or set given byte (set*) respectively depending on different combinations of the condition codes.

"change the instruction pointer" => Jump to execute different instructions. We will cover how these relate next week!

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	b $>_U$ a
jb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	b $<_U$ a

Exercise 2

Write an equivalent C function for the following x86-64 code:

```
mystery:
  testl    %edx, %edx
  js       .L3
  cmpl     %esi, %edx
  jge      .L3
  movslq   %edx, %rdx
  movl     (%rdi,%rdx,4), %eax
  ret
.L3:
  movl     $0, %eax
  ret
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(? x, int y, int z)
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(? x, int y, int z) {
  if ( )

  else

}
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(? x, int y, int z) {
  if (z >= 0 && z < y)

  else

}
```

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	b $>_U$ a
jb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	b $<_U$ a

Exercise 2

```
mystery:
  testl    %edx, %edx
  js       .L3
  cmpl     %esi, %edx
  jge      .L3
  movslq   %edx, %rdx
  movl     (%rdi,%rdx,4), %eax
  ret
.L3:
  movl     $0, %eax
  ret
```

```
int mystery(int *x, int y, int z) {
  if (z >= 0 && z < y)
    return x[z];

  else

}
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(int *x, int y, int z) {
  if (z >= 0 && z < y)
    return x[z];

  else
    return 0;
}
```

Exercise 3

Convert the following C function into x86-64 assembly code. You are not being judged on the efficiency of your code – just correctness.

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
        return *x;
}
```

Exercise 3

Convert the following C function into x86-64 assembly code. You are not being judged on the efficiency of your code – just correctness.

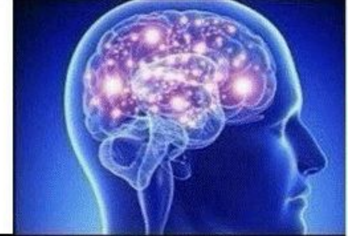
```
happy:
     cmpq %rdx, %rsi
     jle .else
     leaq (%rdx, %rsi), %rax
     ret
.else:
     movq (%rdi), %rax
     ret
```

Multiple other possibilities (e.g. switch ordering of if/else clauses, replace lea with mov/add instruction pair).

GDB!

GDB

PRINTF()



imgflip.com

Debugging methods...

The GNU Debugger (GDB)

The GNU Debugger (GDB) is a powerful debugging tool that will be critical to Lab 2 and Lab 3 and is a useful tool to know as a programmer moving forward.

There are tutorials and reference sheets available on the course webpage.

Make sure that you're familiar with GDB because you'll be using it a lot on labs 2 and 3.

Take some time to learn helpful commands like **`bt (backtrace)`** and **`tui/layout`**.

Tutorial time!

That's All, Folks!

Thanks for attending section! Feel free to stick around for a bit if you have quick questions (otherwise post on Ed or go to office hours).

See you all next week! :-)

Best Programming Language