

University of Kansas | Drew Davidson

ECCS 665 **COMPILER** ***CONSTRUCTION***

x64 Basics

Last Lecture

Encoding Programs

Representing Whole Programs in Hardware

- Data
- Code

ISAs

- X86 – Example target

What you should know:

- ISA concept
- That data and code share memory
 - The instruction pointer
- How to play with ASM code



Architecture

Recall: Instructions

Review x64 Intro

We've already seen two instructions:

`movq <o1> <o2>` 

- Move quadword in o₁ into quadword o₂

`syscall`

- Invoke a system service

```
.text
.globl _start
_start:
    movq $60, %rax    # Choose syscall exit
    movq $4, %rdi     # Set syscall arg - return code
    syscall
```

Recall: x64 Registers

Review x64 Intro

Computation needs to be done on registers

- General workflow:

src code

a = b <opr> c

asm code

Load b into a register

Load c into another register

Do opr on registers

Store result register into a

Name	Number	“Nickname”
%rax	0	Accumulator
%rbx	1	Base
%rcx	2	Counter
%rdx	3	I/O
%rsi	4	String source
%rdi	5	String destination
%rbp	6	Base pointer
%rsp	7	Stack pointer
%r08 - %r15	8 - 15	General purpose
%rip	-	Instruction pointer
%rflags	-	Status flags

General purpose
For our needs

Used for bookkeeping

Bonus general purpose for us

Inaccessible as
regular operands

Today's Lecture

X64 Basics

X64 Discussion

- Some perspective

Architecture Details

- Basic operators
- Memory Directives



Architecture

ASM Instruction Syntax

Lecture Outline –Writing x64

As with everything x86-related, it's complicated

- We'll use the AT&T Syntax:
 <opcode><sizesuffix> <src operand(s)> <dst operand>
- Immediates (i.e. constant values) prefixed by \$
- Registers prefixed by %
- Memory lookup (i.e. dereference) in parens

```
movq $5, (%rax)
```

*mov the 64-bit value 5 into the 64-bit
memory address specified by register rax*

Directives

Lecture Outline –Writing x64

- Indicates a command to the assembler
 - Layout, program entrypoint, etc.

Example:

`.globl X`

Indicates that symbol X is visible outside of the file

Segment Directives

Lecture Outline –Writing x64

`.text`

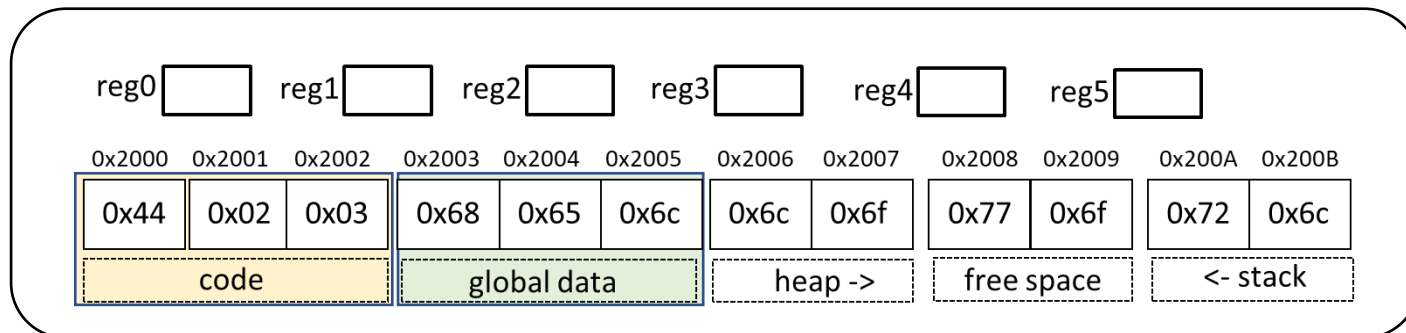
Lay out items in the
user text segment

Instructions go here

`.data`

Lay out items in the
data segment

Globals go here



Labels

Lecture Outline –Writing x64

- The assembler allows us to specify “placeholder” addresses that will be used later
 - Translated to “real” addresses by a utility called the linker
- Valid for both data and code locations

```
jmp LBL1
```

```
...
```

```
LBL1: movq $5 (%rax)
```

```
jmp 0x12d34a5678a
```

System Calls

Lecture Outline –Writing x64

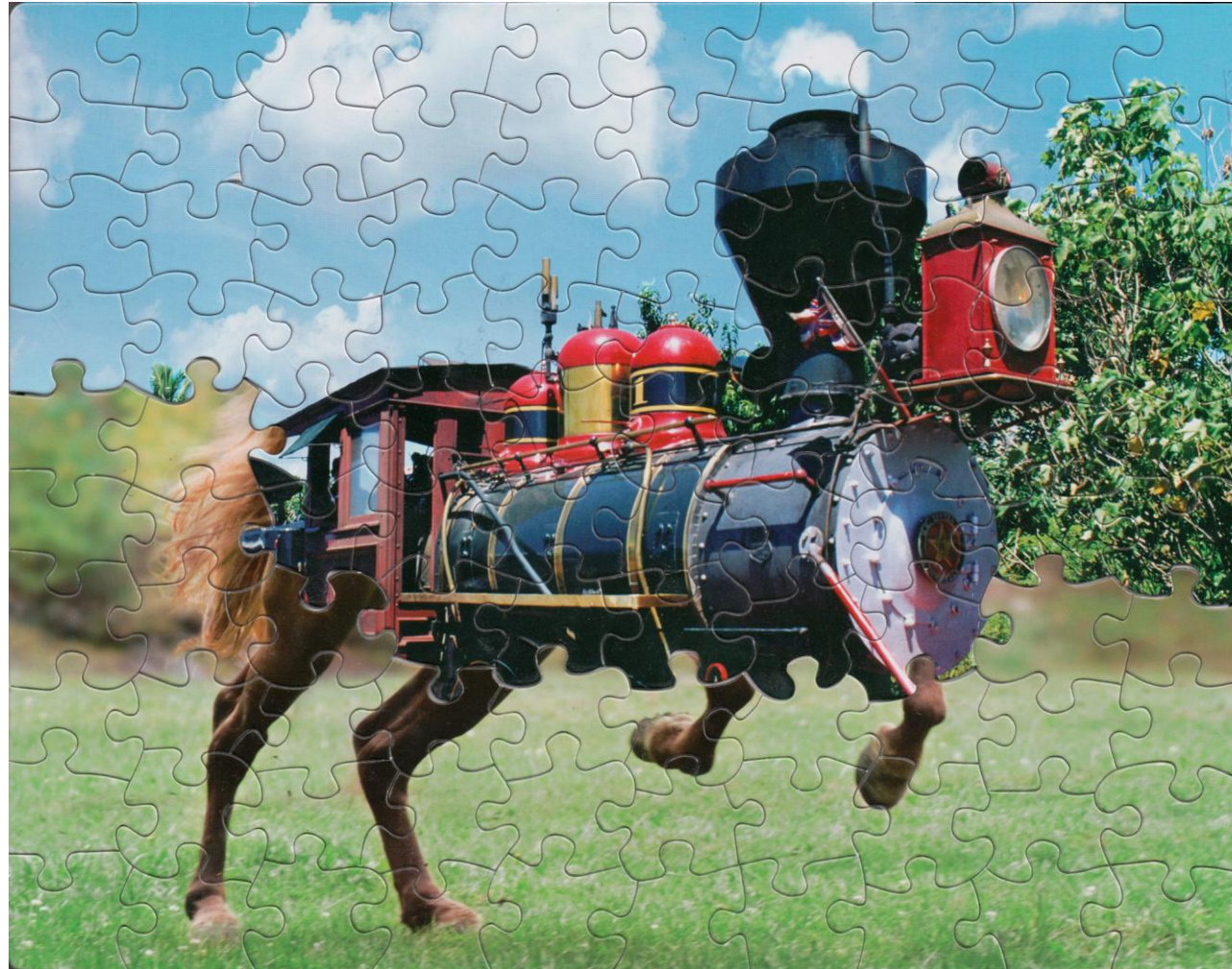
To interact outside program memory, need the help of the OS

```
syscall
```

```
%rax    # Which system call (60 is exit)
%rdi    # Set syscall arg - (exit takes the return code)
```

Time to put it all together!

Lecture Outline –Writing x64



A Complete Program

Lecture Outline –Writing x64

```
.text
.globl _start
_start:
    movq $60, %rax    # Choose syscall exit
    movq $4, %rdi     # Set syscall arg - return code
    syscall
```

Actually Running a Program

Lecture Outline – Writing x64

Invoking the assembler and linker

```
as -o start.o start.s
ld start.o -o prog
./prog
echo $?
```

Summary

ISAs

ISAs

- Provide an interface from software to hardware
- We'll target assembly code, assembler will take it from there

X64

- A popular architecture
- We've covered the basic instruction format and a simple program

Compiler Construction

Progress Pics

Done:

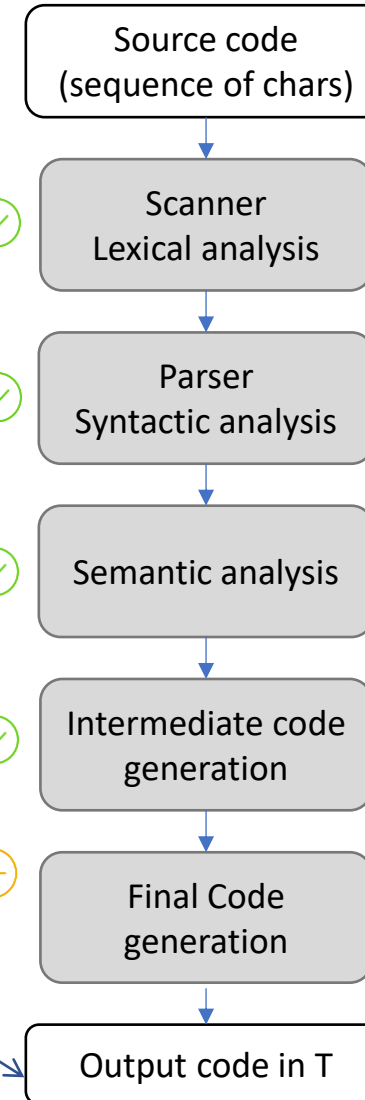
- Validated and abstracted input program
- Flattened the input program to a linear representation

ToDo:

- Concretize to the target architecture

We gotta understand this!

In progress 🕒



A Moment of Appreciation

x64 Basics: Perspective

We're writing real-deal x64 - no special tools needed

- Our x64 code looks (mostly) like what comes out of gcc

```
gcc -S prog.c -o prog.s
```

```
gcc -c prog.s -o prog.o
```

```
gcc prog.o -o prog.exe
```

Unfortunately for clarity, there are a lot of extra directives we might want to strip:

```
gcc -fno-asynchronous-unwind-tables -fno-dwarf2-cfi-asm -O0 -S  
blah.c -o blah.nodebug.S
```

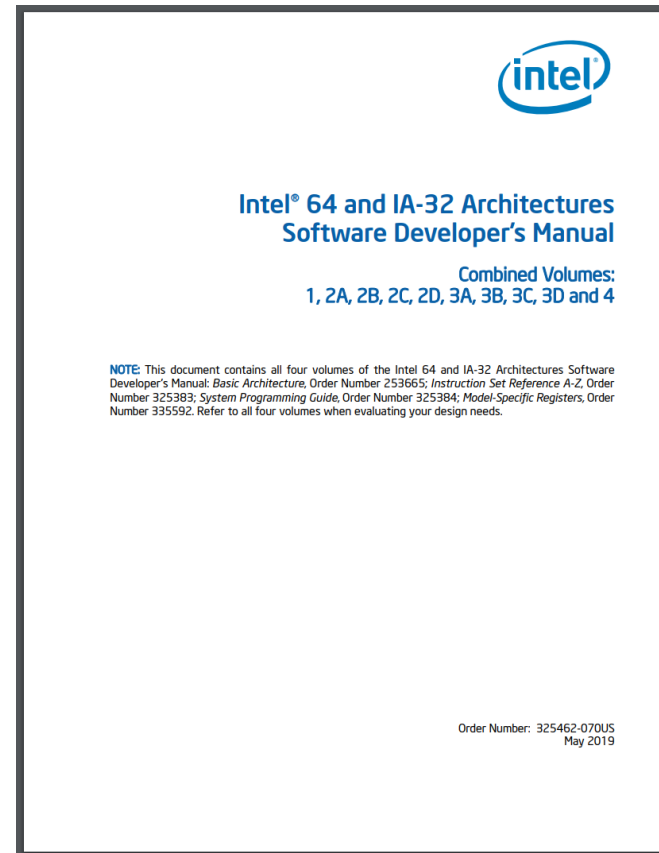
Docs: A Benefit to x64's Popularity

x64 Basics: Perspective

Lots of good, thorough documentation out there

- Some best used as reference rather than walkthrough

*4922
Pages!*



<https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>

There are a LOT of Instructions in X64!

X64 Basics: Perspective



The @x86instructions parody twitter account, which just posts made-up instructions

Keeping to the “Easy” Path

X64 Basics: Perspective

We’ll sidestep most of the x64 complexity

- We’ll just stick to those instructions we need
- Give you the pointers to explore the rest of the ISA



Today's Lecture

X64 Basics

X64 Discussion

- ✓ • Some perspective

Architecture Details

- Basic operators
- Memory Directives



Architecture

Let's Dive In to X64!

X64 Basics – Architecture Details



Simple Arithmetic Instructions

x64 Basics

`negq <o1>`

$$O_1 = -O_1$$

Example

```
movq $4, %rax
negq %rax
```

At this pt:
`%rax = -4`

“Support” operand *“Main” operand*

`addq <o1> <o2>`

$O_2 = O_2 + O_1$ Take *<o₂>* and “do” *<o₁>* to it

Example

```
movq $4, %rax
movq $5, %rbx
addq %rax, %rbx
```

At this pt:
`%rbx = 9`

`subq <o1> <o2>`

$$O_2 = O_2 - O_1$$

Example

```
movq $4, %rax
movq $5, %rbx
subq %rax, %rbx
```

At this pt:
`%rbx = 1`

Wacky Arithmetic Instructions

x64 Basics

`imulq <o1>`

$\%rdx:\%rax = \%rax * o_1$

Example

```
movq $2, %rax  
movq $4, %r11  
imulq %r11
```

} 2 * 4

At this pt:

$\%rax = 8$

$\%rdx = 0$

`idivq <o1>`

$\%rax = \%rdx:\%rax / o_1$

$\%rdx = \%rdx:\%rax \% o_1$

Example

```
movq $0, %rdx  
movq $6, %rax  
movq $2, %r8  
idivq %r8
```

} 6 / 2

At this pt:

$\%rax = 3$

$\%rdx = 0$

Logical Instructions

x64 Basics

`notq <o1>`

$o_1 = \text{bitflip } o_1$

Example

```
movq $1, %rcx
notq %rcx
```

At this pt:
%rcx = -2

Why -2?

Before bitflip

%rcx 0..01

After bitflip

%rcx 1..10

2's complement...

0..01 Flip all bits

0..10 Add 1

$0 + 2 + 0 = (\text{negative}) 2$

`andq <o1> <o2>`

$o_2 = o_2 \& o_1$

Example

```
movq $12, %rdx
movq $10, %rax
andq %rax, %rdx
```

At this pt:
%rdx = 8

Before and

%rdx 0..1100

%rax 0..1010

After and

%rax 0..1000

$0 + 8 + 0 + 0 + 0 = 8$

`orq <o1> <o2>`

$o_2 = o_2 | o_1$

Example

```
movq $12, %rdx
movq $10, %rax
orq %rax, %rdx
```

At this pt:
%rdx = 14

Before or

%rdx 0..1100

%rax 0..1010

After or

%rax 0..1110

$0 + 8 + 4 + 2 + 0 = 14$

`xorq <o1> <o2>`

$o_2 = o_1 \wedge o_2$

Example

```
movq $5, %rdx
movq $6, %rax
xorq %rax, %rdx
```

At this pt:
%rdx = 3

Before xor

%rdx 0..0101

%rax 0..0110

After xor

%rax 0..0011

$0 + 2 + 1 = 3$

Conditionals

x64 Basics

```
movq $1, %rbx
movq $1, %rax
cmpq %rax, %rbx
je LBL_1
```

Jump to LBL_1

Another weird x86 thing

- conditional jump instructions – no explicit condition!
- Uses the flags register rflags
 - Assumes a previous operation has set the flags register

```
je <LBL>
```

Jump to label LBL if equal, else fallthrough

“Wait... jump if *what* is equal?”

```
cmpq %rax, %rbx
je <LBL>
```

Jump to label LBL if rax and rbx are equal, else fallthrough

Conditionals and rflags

x64 Basics

rflags: bitvector register of conditions

OF “Overflow Flag”

Result exceeds storage

SF “Sign Flag”

Result was negative

ZF “Zero Flag”

Result was zero

```
movq $1, %rax
movq $2, %rbx
cmpq %rax, %rbx
sete <o1>
```

At this pt:
 $\langle o_1 \rangle = 0$

cmpq src1 src2 : Set flags as though computing src2 – src1

je	sete	E	(E)quality	ZF
jne	setne	NE	(N)ot (E)qual	$\neg$ ZF
		G	(G)reater	$(\neg ZF \wedge \neg SF) \oplus OF$
jg	setg	L	(L)ess	$SF \oplus OF$
jl	setl			
jge	setge	GE	(G)reater or (E)qual	$\neg SF \oplus OF$
jle	setle	LE	(L)ess or (E)qual	$(SF \oplus OF) \vee ZF$

 sete %rax

 sete %al

You can now (kinda) write assembly programs!

x64 Basics

src code

```
int main() {  
    return 2 - 1;  
}
```



asm code

```
.globl _start  
.text  
_start:  
    movq $1, %r10  
    movq $2, %r11  
    subq %r10, %r11  
  
    #exit  
    movq $60, %rax  
    movq %r11, %rdi  
    syscall
```

Today's Lecture

X64 Basics

X64 Discussion

- Some assurances

Architecture Details

- Basic operators
- Memory directives



Architecture

Manipulating Memory

X64 Basics

Data can't always fit into registers

Two things we need to do:

1. Allocate memory
2. Read/write memory



A different kind of memory being manipulated

Recall: Memory as an Array

X64 Basics

Memory is just a big ol' array of bytes (with OS mediation)

- Assembler and friends will map the code into memory
- We still need to map data to memory
 - variables, objects, strings, arrays, etc.

Assembly code

Lbl1: `subq %r12, %r13`

Lbl2: `movq %r13, %rdi`

????????????????

Binary memory

0x400086: 0x4d 0xe9 0x25

0x400089: 0x4c 0x89 0xef

0x40008C: 0x7

Address	Address	Address	Address	Address	Address	Address	Address	Address	Address
0x400086	0x400087	0x400088	0x400089	0x40008A	0x40008B	0x40008C	0x40008D	0x40008E	0x40008F
0x4d	0xe9	0x25	0x4c	0x89	0xef	0x07	0x00	0x00	0x00
subq %r12, %r13			movq %r13, %rdi			The 32-bit value 7			
code						data			

Lecture Done!

x64 Basics: Summary

Some basic x64 details

- Instructions
- Data directives
- Memory intuition

Next Time

- Dive further into x64
- Describe memory operations