CS 33 – Discussion 1A
Computer System Organization

Week 2

Questions before we start?

Logistics

- Homework #1 due tonight
- Lab 1 due next week Friday 4/17
- TA site made for uploading disc recordings
  - Also can upload anonymous feedback to improve disc

Agenda

10am-11am PST:

- x86-64 basics
- Linux/Vim basics
- Compilation/assembly code example

11am-11:50am PST:

- LA worksheet

x86-64 Basics – CISC/RISC

- X86-64 is a CISC is a *"Complex Instruction Set Computer"*
  - *Started because instructions were more compact – instruction memory took up less space, a concern back when we had small memory capacity*
  - *Many complex instructions possible*
    - *Vector operations (SIMD)*
    - *Single byte instructions (push/pop)*
    - *many, many more*
- RISC is "*Reduced Instruction Set Computer*"
  - *Small set of fundamental instructions that get repeated*
  - *Example architecture is MIPS*
    - *(**Microprocessor without Interlocked Pipelined Stages**)*
      - I – R – J type instructions
        - all instructions 32 bits

x86-64 Basics - ISA

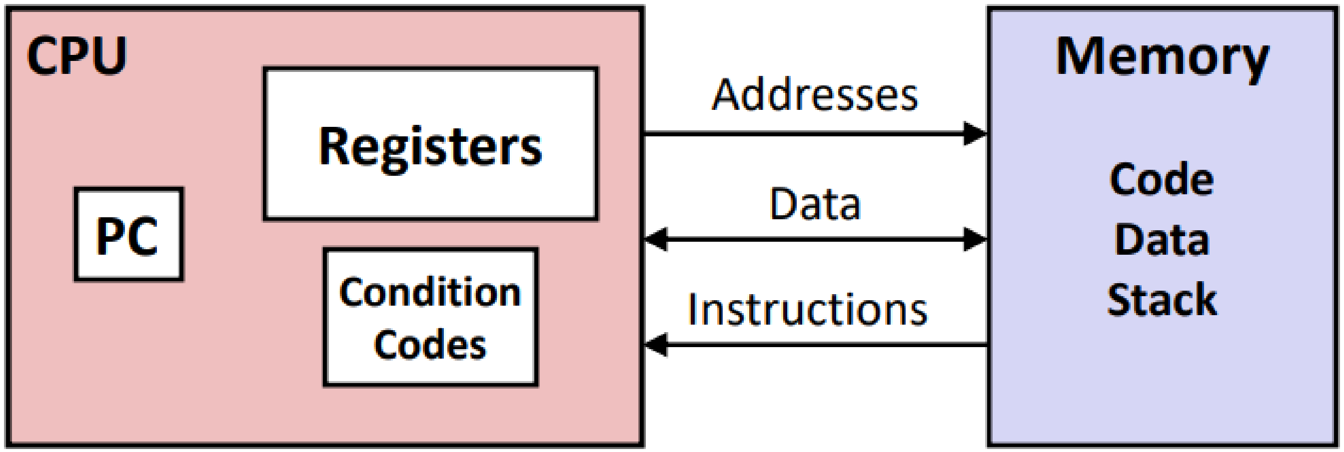
- RISC and CISC are examples of an ISA
  - *Instruction System Architecture*
    - *Essentially the interface between the hardware and software*
    - *Defines the type of instructions we see in assembly*
    - *Also defines the view of the hardware the complier sees (e.g. number of registers that can be used for register renaming/coloring)*
- Microarchitecture is the actual physical implementation of the ISA (digital circuits [NAND gates], sounds like fun right? If you want to keep going lower you reach analog circuits [BJT/CMOSFET transistors], even lower would be material science/semiconductor physics concerned with manufacturing the transistor diagrams [e.g. Molecular Beam Epitaxy or Photolithogrpahy or EBL])

x86-64 Basics - Code

- Machine code:
  - The actual 10101..1110 binary encodings of instructions that correspond to high/low voltages for the circuit that defines your processor
- Assembly code:
  - Higher level than machine code but lower level than regular code
  - Strong correspondence between instructions in assembly and object code (e.g. these 5 bytes are movl $0x01,%rax)

x86-64 Basics - CPU

Assembly/Machine Code View

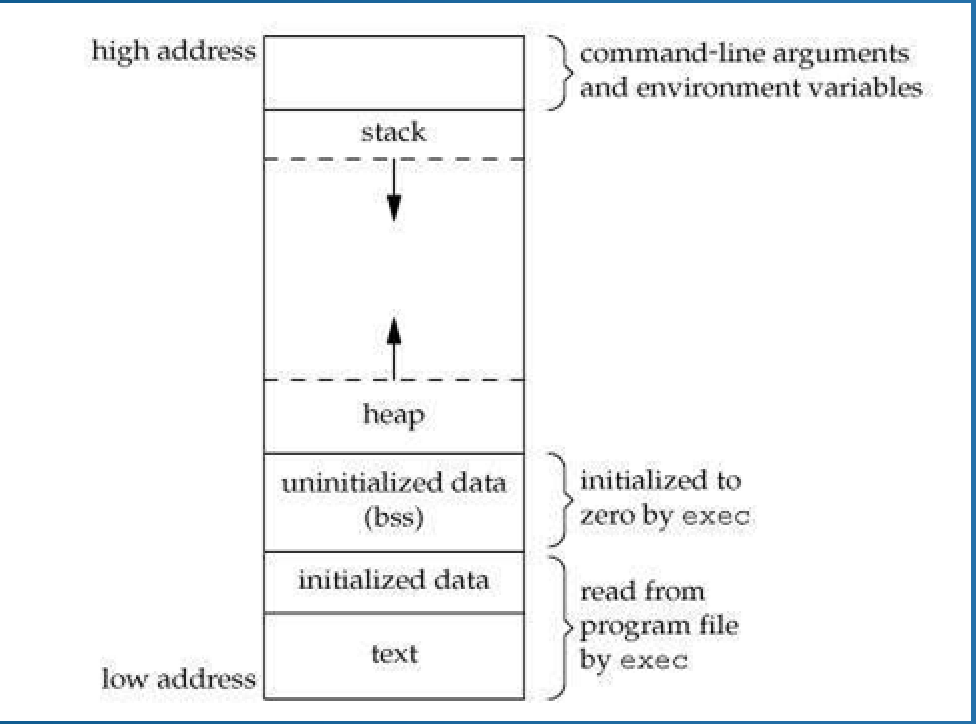


Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

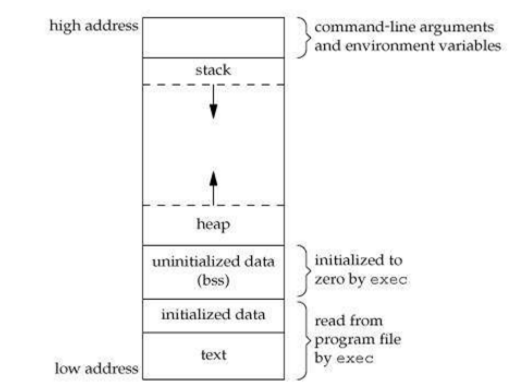
x86-64 Basics – Memory layout

- We assume memory is one large contiguous byte addressable array (we will see later this is an abstraction we can use thanks to virtual memory)



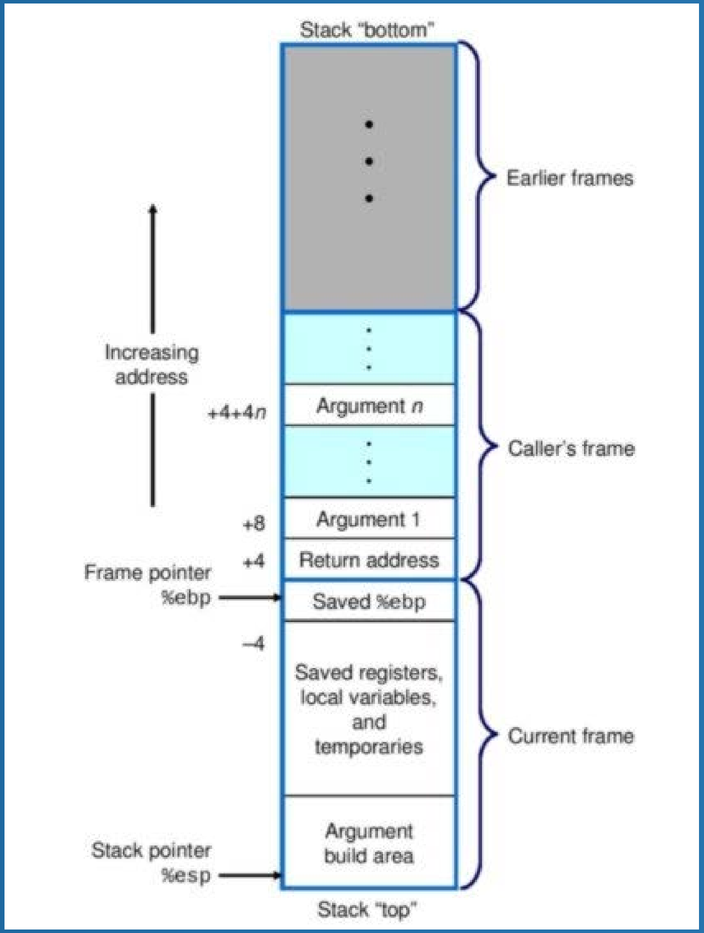
x86-64 Basics – Memory layout

- Stack contains the memory for the function call stack (grows downwards)
- Heap contains the memory dynamically allocated (grows upwards
- Memory in-between them is free until you over allocated (forgot to clean up memory that you “new”)
- .bss is reserved space for uninitialized global/static vars
- .data is reserved space for global/static variables
- Text contains the instructions themselves

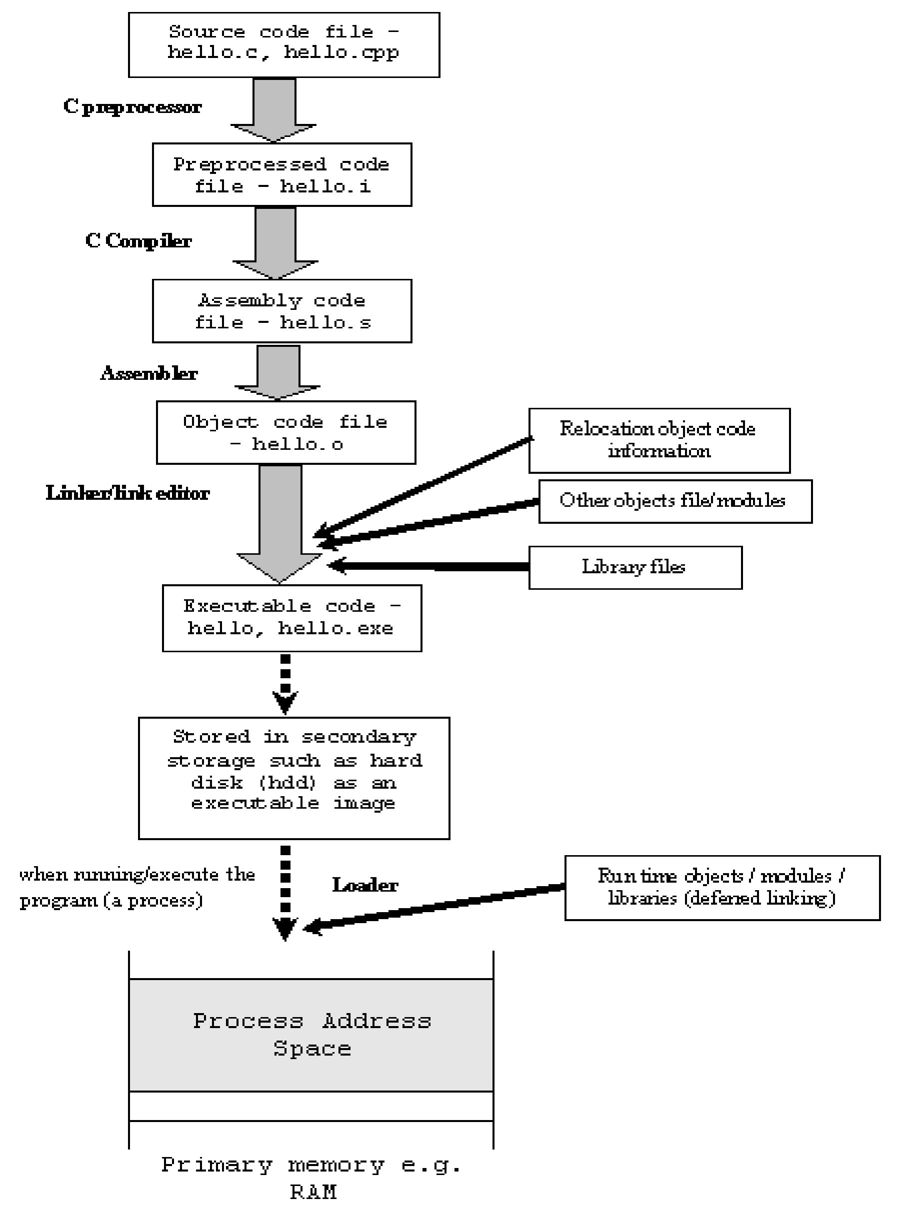


x86-64 Basics – Memory layout

- Early look at a stack frame
- Func stack frame contains:
  - Local vars
  - Return addr
  - Extra func args
    - Cant fit in reg file
  - Calle saved registers



x86-64 Basics – text to object code



Linux Basic Commands

- pwd – print working directory
- cd – change directory
- ls – lists contents of directory (-a lists all, -l prints more info)
- mkdir dir_name – makes a directory with dir_name
- rm file/dir – removes file or dir (use –r flag for dir)
- touch file_name.file_type – creates file_name.file_type in current dir
- vim file_name.file_type – opens file_name.file_type (e.g. bits.c) with the vim text editor
- emacs is an alternative text editor (or nano)
  - or you can develop locally and winscp files to seas

Linux Basic Commands

- gcc flags output_file input_files* - compiles input_files into output_file using a compiler that follows c coding standards
- gdb – launches GNU debugger
  - works for many languages like ada C/C++ fortran etc
  - can set breakpoints and step through code
  - print out stack traces
  - disassemble object code (reverse engineer object code)
- objdump –d obj_file – disassembles object file outside of gdb

VIM Basic Commands

- gg – go to last line of doc
- G – go to first line of doc
- #G – go to line number #
- cntrl + f – move forward one screen
- cntrl + b – move back one screen
- i – switch to insert mode
- v – switch to visual mode (lets you select things to yank (copy) text
  - y – yank (copy) marked text
  - d – delete marked text
  - > / < - shift left / right
- p – paste
- yy – copy line
- dd – cut line
- :w – write but don't exit
- :wq – write and exit
- :q – quit, need to do :q! to force quit (drop any un-saved changes)
- /pattern – search for pattern, ?pattern searches backwards for pattern
- u - undo

Compilation example

Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

Objectdump example

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:   53                push   %rbx
  400596:   48 89 d3          mov    %rdx,%rbx
  400599:   e8 f2 ff ff ff    callq  400590 <plus>
  40059e:   48 89 03          mov    %rax,(%rbx)
  4005a1:   5b                pop    %rbx
  4005a2:   c3                retq
```

- **Disassembler**
  `objdump -d sum`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

Gdb example

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**

  `gdb sum`

  `disassemble sumstore`
  - Disassemble procedure

  `x/14xb sumstore`
  - Examine the 14 bytes starting at `sumstore`

x86-64 registers

Register	Callee Save	Description
`%rax`		result register; also used in `idiv` and `imul` instructions.
`%rbx`	yes	miscellaneous register
`%rcx`		fourth argument register
`%rdx`		third argument register; also used in `idiv` and `imul` instructions.
`%rsp`		stack pointer
`%rbp`	yes	frame pointer
`%rsi`		second argument register
`%rdi`		first argument register
`%r8`		fifth argument register
`%r9`		sixth argument register
`%r10`		miscellaneous register
`%r11`		miscellaneous register
`%r12-%r15`	yes	miscellaneous registers

x86-64 registers

4.3 Register Usage

There are sixteen 64-bit registers in x86-64: **`%rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp,`** and **`%r8-r15`**. Of these, **`%rax, %rcx, %rdx, %rdi, %rsi, %rsp,`** and **`%r8-r11`** are considered caller-save registers, meaning that they are not necessarily saved across function calls. By convention, **`%rax`** is used to store a function's return value, if it exists and is no more than 64 bits long. (Larger return types like structs are returned using the stack.) Registers **`%rbx, %rbp`**, and **`%r12-r15`** are callee-save registers, meaning that they are saved across function calls. Register **`%rsp`** is used as the *stack pointer*, a pointer to the topmost element in the stack.

Additionally, **`%rdi, %rsi, %rdx, %rcx, %r8,`** and **`%r9`** are used to pass the first six integer or pointer parameters to called functions. Additional parameters (or large parameters such as structs passed by value) are passed on the stack.

In 32-bit x86, the *base pointer* (formerly **`%ebp`**, now **`%rbp`**) was used to keep track of the base of the current stack frame, and a called function would save the base pointer of its caller prior to updating the base pointer to its own stack frame. With the advent of the 64-bit architecture, this has been mostly eliminated, save for a few special cases when the compiler cannot determine ahead of time how much stack space needs to be allocated for a particular function (see Dynamic stack allocation).

%rip – instruction pointer, contains address for next ins for processor

x86-64 basic commands

```
add %r10,%r11     // add r10 and r11, put result in r11
add $5,%r10       // add 5 to r10, put result in r10
call label        // call a subroutine / function / procedure
cmp %r10,%r11     // compare register r10 with register r11.  The comparison sets flags in the processor status register which affect conditional jumps.
cmp $99,%r11      // compare the number 99 with register r11.  The comparison sets flags in the processor status register which affect conditional jumps.
div %r10          // divide rax by the given register (r10), places quotient into rax and remainder into rdx (rdx must be zero before this instruction)
inc %r10          // increment r10
jmp label         // jump to label
je  label         // jump to label if equal
jne label         // jump to label if not equal
jl  label         // jump to label if less
jg  label         // jump to label if greater
mov %r10,%r11     // move data from r10 to r11
mov $99,%r10      // put the immediate value 99 into r10
mov %r10,(%r11)   // move data from r10 to address pointed to by r11
mov (%r10),%r11   // move data from address pointed to by r10 to r10
mul %r10          // multiplies rax by r10, places result in rax and overflow in rdx
push %r10         // push r10 onto the stack
pop %r10          // pop r10 off the stack
ret               // routine from subroutine (counterpart to call)
syscall           // invoke a syscall (in 32-bit mode, use "int $0x80" instead)
```

x86-64 command suffixes

- "byte" refers to a one-byte integer (suffix **b**),
- "word" refers to a two-byte integer (suffix **w**),
- "doubleword" refers to a four-byte integer (suffix **l**), and
- "quadword" refers to an eight-byte value (suffix **q**).

Most instructions, like **mov**, use a suffix to show how large the operands are going to be. For example, moving a quadword from **%rax to %rbx** results in the instruction **movq %rax, %rbx**. Some instructions, like `ret`, do not use suffixes because there is no need. Others, such as **movs** and **movz** will use two suffixes, as they convert operands of the type of the first suffix to that of the second. Thus, assembly to convert the byte in **%al** to a doubleword in **%ebx** with zero-extension would be **movzbl %al, %ebx**.

x86-64 operand types

- *Imm* refers to a constant value, e.g. **0x8048d8e or 48**,
- E_x refers to a register, e.g. **%rax**,
- $R[E_x]$ refers to the value stored in register E_x, and
- $M[x]$ refers to the value stored at memory address x.

Note you cannot use two memory operands in one operation

x86-64 addressing modes

Type	From	Operand Value	Name
Immediate	*$Imm*	*Imm*	Immediate
Register	E_a	$R[E_a]$	Register
Memory	*Imm*	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_b]]$	Absolute
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + (R[E_i] \times s)]$	Scaled indexed

Note you cannot use two memory accessing in one operation

- Why?

Questions before we start worksheet?

Resources used

- https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf
- https://cs.stackexchange.com/questions/76871/how-are-variables-stored-in-and-retrieved-from-the-program-stack
- https://www.geeksforgeeks.org/memory-layout-of-c-program/
- https://vim.rtorr.com/
- https://www.tenouk.com/ModuleW.html
- Professor Reinman's slides (CCLE)

- **Credit for slides: Attiano Purpura-Potoniere**