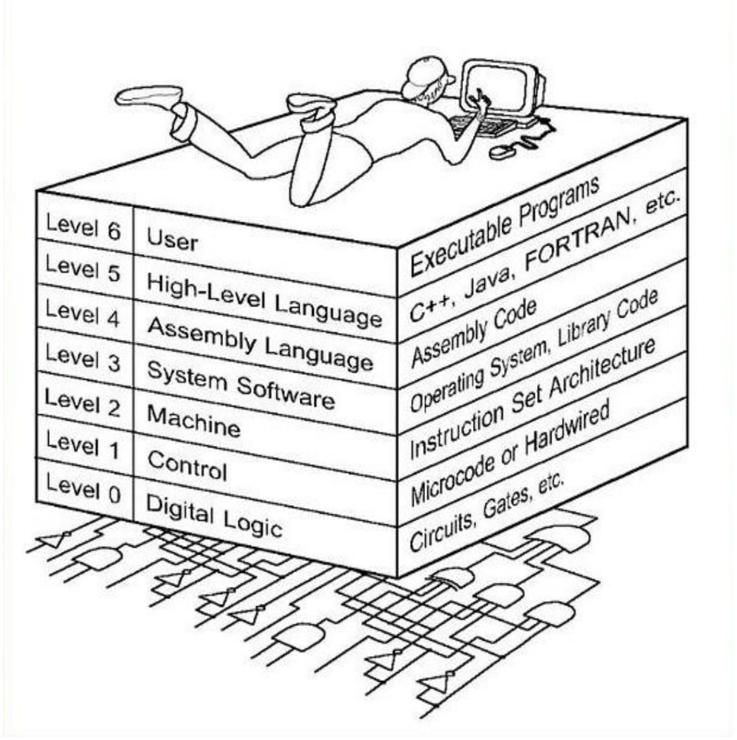
UMBC

x86 Assembly

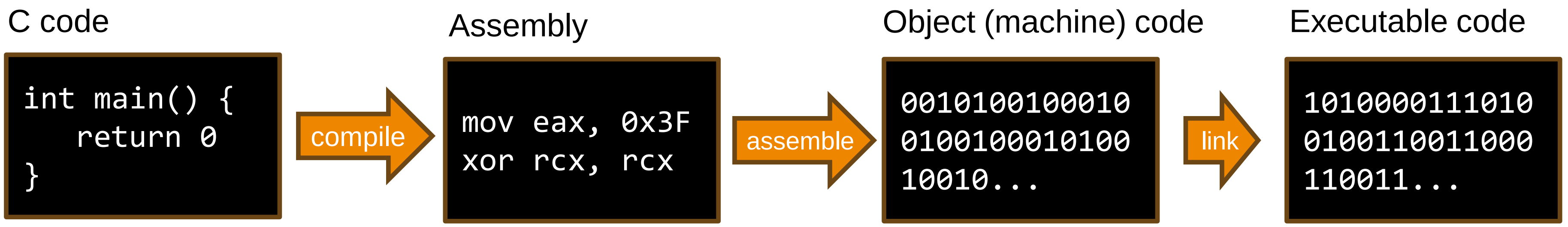
CMSC 313
Raphael Elspas

Computer Level Hierarchy

- In this class we focused on level 0 and level 4, an intro to level 1 & 2.
- Level 1 and 2 covered more in CMSC411
- Level 3 is covered in CMSC421
- Level 5 has been covered in previous classes



Assembling vs compiling



```
int main() {
    return 0
}
```

```
mov eax, 0x3F
xor rcx, rcx
```

```
0010100100010
0100100010100
10010...
```

```
1010000111010
0100110011000
110011...
```

The C compiler has a built-in assembler and linker, which allows us to not worry about the assembly or linking step if we're just developing C code

Note: the examples of assembly, object code, and executable code are not real compiled or assembled versions of the c code.

Assembly Language

- Low level programming language
- It uses human readable words to represent machine code
- Assembly is specific to the processors architecture
- Level “above” machine code
- An assembler converts assembly to object code (machine code)
- We will use the NASM Assembler for intel x86 assembly
- We will focus on 64-bit architecture, which can also run 32-bit assembly
    - Apple will not let you run 32 on 64-bit architecture

Basic NASM Syntax

- Assembly is written as lines rather than statements
- The Basic NASM syntax has 4 components on a typical line:
    - Label
    - Opcode
    - Operand(s)
    - Comment

```
Label: opcode operand(s) ; comment
```

Basic NASM Syntax: Label

- A label is an address to that line of code
- Followed by :
- Some words are reserved, like ADD
- Optional
- If label is not used, indent to keep lines neat
- Label can be on a line by itself
- Should be the first thing on a line

```
Label: opcode operand(s) ; comment
```

Basic NASM Syntax: Opcode

- An opcode is an instruction
- Not case sensitive. ADD = add
- Can be a:
  - machine instruction
  - an assembler directive (pseudo-instruction)
  - macro call

```
Label: opcode operand(s) ; comment
```

Basic NASM Syntax: Operand(s)

- Depends on opcode
- Can be a combination of
  - Registers
  - Constants
  - Memory references
  - Or empty (like for RET)

```
Label: opcode operand(s) ; comment
```

Intel syntax vs AT&T syntax

- Depending on your assembler, the operands may be expected in different orders. Both are available for x86-64 assembly.
- There are two typical operand orders
  - Destination, source – called intel syntax
    - we will use intel syntax for this class
  - Source, destination – called AT&T syntax

```
ADD eax, 5
```

```
ADD $5, %eax
```

These both mean:
eax = eax + 5

Basic NASM Syntax: comment

- Comments begin with ;
- Are optional, but encouraged
- No easy way to do multiline comments, you need ; before every comment

```
Label: opcode operand(s) ; comment
```

Sections

- One of the NASM assembler directives is the “SECTION” or “section” directive
- There are 4 predefined sections for the ELF format: .data, .bss, .rodata, .text
- The format for indicating the beginning of a section is the word “section” followed by the section name. For example:

```
section .data
```

.data section

- The .data section has these properties:

```
section .data ; initialized data
              ; writeable, not executable
              ; default alignment 8 bytes
```

- Example:

  `int x = 3;`

  in compiled C++ would go in the “.data” section

.bss section

- .bss stands for Block Started by Symbol
- The .bss section has these properties:

```
section .bss ; uninitialized data
             ; writeable, not executable
             ; default alignment 8 bytes
```

- Example:

`int x;`

in compiled C++ would go in the “.bss” section

.rodata section

- The .rodata section has these properties:

```
section .rodata ; initialized data
                ; read only, not executable
                ; default alignment 8 bytes
```

- Example:

`const float pi = 3.14;`

in compiled C++ would go in the ".rodata" section

.text section

- The .text section is the only section where instructions go

```
section .text ; not writeable, executable
              ; default alignment 16 bytes
```

- Example:

  `printf();` and

  `x = x + 2;`

  would go in the .text section

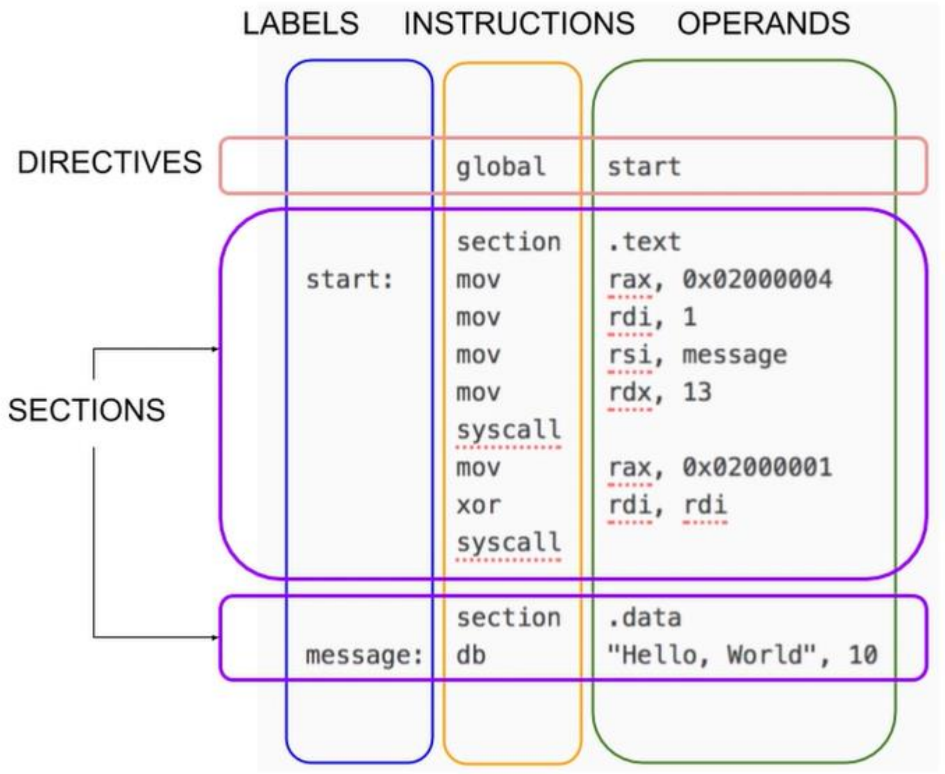
Special section

- You can create your own sections by using a section name other than .data, .bss, .rodata, and .text
- Don't use quotes in the defining of your section.

```
section “other” ; not writeable, not executable
                ; default alignment 1 byte
```

- Write whatever you want here, but it's not typically used for anything.

Summary



```
          global    start

          section   .text
start:    mov       rax, 0x02000004
          mov       rdi, 1
          mov       rsi, message
          mov       rdx, 13
          syscall
          mov       rax, 0x02000001
          xor       rdi, rdi
          syscall

          section   .data
message:  db        "Hello, World", 10
```

We store this file as a .asm file

ISA Reference

- There are many references online to the ISA for x86-64
- below are some links, but there are many more locations to find x86 instruction listings.
- Official x86 instruction listing: https://cdrdv2.intel.com/v1/dl/getContent/671200
- Compressed listing of x86 instructions: https://www.felixcloutier.com/x86/
- Has a description of reading instruction data sheets: http://ref.x86asm.net/

Integer vs float operations

- Some operations use integers as inputs and others use floats.
- For example: ADD performs integer addition and FADD performs float addition.
- The operations that use floats require floating point registers as inputs and cannot use integer registers as inputs

Common Instructions

- ADD – add 2 numbers
- SUB – subtract a number from another number
- INC – increment: add 1 to a number
- DEC – decrement: subtract 1 from a number
- MOV – mov a value from one location to another
- NOP – No Operation: do nothing

Logical Instructions

- AND – Logical and two numbers
- OR – logical or two numbers
- NOT – logical not a number
- XOR – logical xor two numbers
- SHL – logical Shift left
- SHR – logical Shift right
- SAL – Arithmetic Shift left
- SAR – Arithmetic Shift right
- ROL – logical Rotate left
- ROR – logical Rotate right
- RCL – rotate through carry left
- RCR – rotate through carry right

Arithmetic instructions

- NEG – 2s complement negation of operand
- MUL – unsigned multiply of operands
- IMUL – signed multiply of operands
- DIV – unsigned divide of operands
- IDIV – signed divide of operands

Subroutine instructions

- PUSH – push value onto stack
- POP – pop value from stack
- CALL – call a subroutine
- RET – return from subroutine

References

- Ivan Sekyonda’s slides
- https://en.wikipedia.org/wiki/FLAGS_register
- http://asmdebugger.com/
- https://cdrdv2.intel.com/v1/dl/getContent/671200
- https://www.felixcloutier.com/x86/
- http://ref.x86asm.net/