CMSC389R

Binaries I

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

recap

HW8 and HW9

UMDCTF

--

Questions?

Itinerary

- How programs work
- Compilation process
- Instruction Set Architectures
- x86 Assembly
    - Language
    - Writing/running assembly programs

Computer Programs

- Interpreted
  - Write source code (Python, Ruby, etc)
  - Run in interpreter
- Compiled
  - Write source code (Java, C, etc)
  - Compile (*javac*, *gcc*, *llvm*)
  - Run it

Compilation Process (C)

Source Code → Compiler → Assembly → Assembler → Machine Code → Linker → Final Binary

Library → Linker

Compilation Process

- Source Code: human written program
- Assembly: human readable mnemonics of machine language (though translation is not always one-to-one)
- Machine code: ones/zeros the CPU directly interprets

Compilation Process

- Compiler: code -> assembly
- Assembler: assembly -> machine code
- Linker: resolves external dependencies (imports, libraries)

Compilation Process (C)

Source Code

Compiler

Assembly

Assembler

Machine Code

Linker

Library

Final Binary

Instruction Set Architecture

Instruction Set Architecture

- Complex Instruction Set Computer (CISC)
    - Single instructions are super powerful
    - Multi-step operations from a single instruction
    - Flexible in programming style due to multiple complex instructions
    - Instructions have variable length
        - i.e. 1 byte to 9 bytes, or more

Instruction Set Architecture

- Reduced Instruction Set Computer (RISC)
    - Single instructions are simple
    - Each instruction does one thing
    - Most operations involve registers
    - Few operations deal with memory
        - RISC also called “load/store” arch
    - Longer code vs CISC

Instruction Set Architecture

- Too many CPUs exist... many machine codes too
- x86: Intel CPUs, emulated by AMD
  - Desktop computers, servers
- ARM: IP licensed to companies who implement it
  - Raspberry Pi, Android phones, routers
- MIPS: Prevalent RISC arch we study today
  - Used in routers and old game consoles

Types of Computers

- Stack Machines
  - Instructions manipulate and store values on the top of the stack
- Accumulator Machines
  - Performs most calculations on and stores results in a single “accumulator” register
- Register Machines
  - Has multiple registers for operations

Assembly Language

- We’ll be using x86 assembly in 32 bit mode
- Why still learn assembly?
  - Reverse Engineering (here)
  - OS development
  - Compiler writing
  - Computer architecture design

x86

- Registers
- Syntax
- Instructions
  - Arithmetic
  - Data
  - Control Flow
- Calling Conventions
- Tooling

Registers

- Original design made heavy use of an accumulator register
  - Many opcodes to do operations on just one register
- 8 “general purpose” registers
  - Some registers have specialized purposes
  - Naming convention is mostly historical
  - A lot more registers as well

Registers

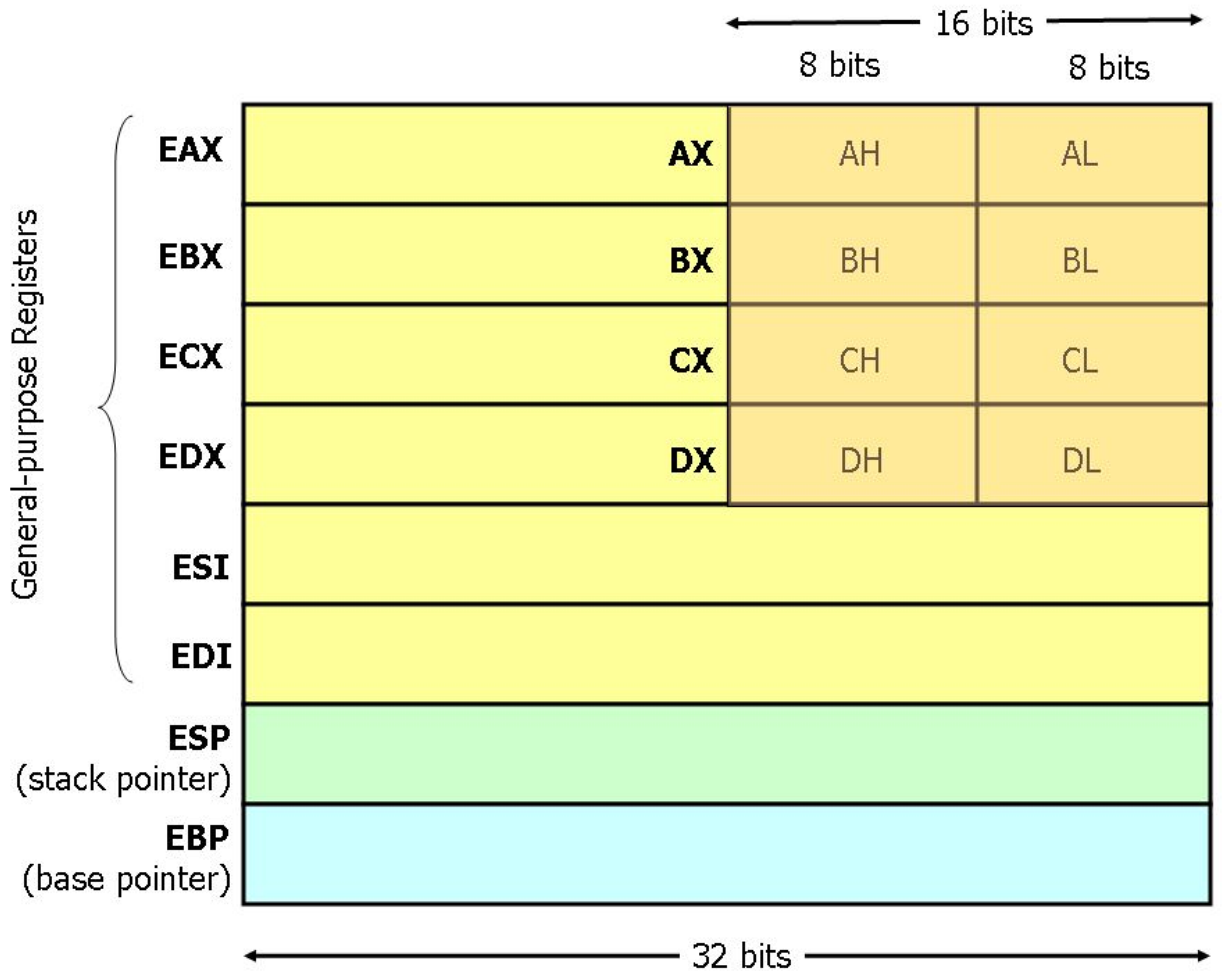
- EAX - “Accumulator” register
  - Heavy use for arithmetic
- EDX - “Data” register
  - Closely tied with EAX operations
  - e.g. stores extra data from multiplication
- ECX - “Counter” register
  - Used as loop counter and for bit shifting
- EBX - “Base” register
  - Used to be memory base pointer in 16-bit x86, but has no special purpose now :(

Registers

- Can access lower parts of EAX/EBX/ECX/EDX with smaller registers
    - EAX - “Extended” AX
    - AX - lower 16 bits of EAX
    - AH - upper 8 bits of AX
    - AL - lower 8 bits of AX
    - Same with other letters (B, C, D)
- Can only use registers together with same size
    - Need to use expansion instructions to interface w/ bigger registers

Registers

- ESI/EDI - Source/Destination Index
  - Used as a pointer for things like string manipulation
- EBP - Base Pointer
  - Points to the bottom of the current stack frame
  - Use to reference function parameters
- ESP - Stack Pointer
  - Points to top of stack
  - Used to grow/shrink stack for local variables/data
- More on history here https://www.swansontec.com/sregisters.html


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

Syntax

- x86 has two types of assembly syntax
- AT&T
  - Registers are marked with %
  - Immediates (number literals) marked with $
  - Memory addressing syntax uses () and is convoluted
  - Most instructions in format *<instr> <src>, <dst>*
- Intel
  - Registers and immediates don’t have marks
    - hex/binary immediates appended w/ h or b
    - If hex literal begins with abcdef, prepend 0
  - Memory addressing uses [] and is more intuitive
  - Most instructions in format *<instr> <dst>, <src>*

```
6c                      ins    BYTE PTR es:[rdi],dx
69 62 36 34 2f 6c 64    imul   esp,DWORD PTR [rdx+0x36],0x646c2f34
2d 6c 69 6e 75          sub    eax,0x756e696c
78 2d                   js     400275 <__uflow@plt-0x14cb>
78 38                   js     400282 <__uflow@plt-0x14be>
36 2d 36 34 2e 73       ss sub eax,0x732e3436
6f                      outs   dx,DWORD PTR ds:[rsi]
2e 32 00                xor    al,BYTE PTR cs:[rax]
```

```
6c                      insb   (%dx),%es:(%rdi)
69 62 36 34 2f 6c 64    imul   $0x646c2f34,0x36(%rdx),%esp
2d 6c 69 6e 75          sub    $0x756e696c,%eax
78 2d                   js     400275 <__uflow@plt-0x14cb>
78 38                   js     400282 <__uflow@plt-0x14be>
36 2d 36 34 2e 73       ss sub $0x732e3436,%eax
6f                      outsl  %ds:(%rsi),(%dx)
2e 32 00                xor    %cs:(%rax),%al
```

Syntax

- We’ll be using the Intel syntax for this course

Arithmetic Instructions

- *add* - adds two values together
- *sub* - subtracts source from destination value
- *inc* - increments by 1
- *dec* - decrements by 1
- *imul* - performs integer multiplication
- *idiv* - performs integer division
  - quotient -> EAX, remainder -> EDX
- *and/or/xor/not* - bitwise operations
- *neg* - performs two’s complement negation
- *shl/shr* - left and right shift by immediate or CL

Arithmetic Instructions

- *add* - adds two values together
- *sub* - subtracts source from destination val
- *inc* - increments by 1
- *dec* - decrements by 1
- *imul* - performs integer multiplication
- *idiv* - performs integer division
  - quotient -> EAX, remainder -> EDX
- *and/or/xor/not* - bitwise operations
- *neg* - performs two's complement negation
- *shl/shr* - left and right shift by immediate

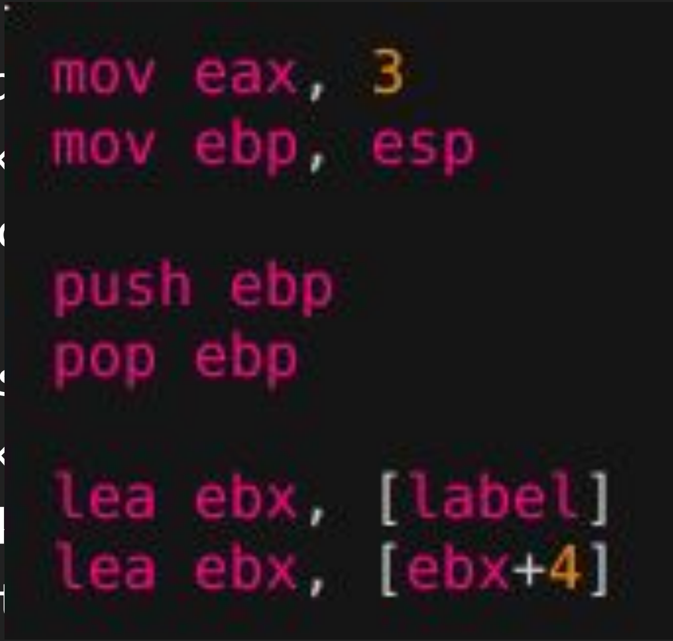
```
add eax, eax
add eax, [ebx+4]
add [ebx], 3

sub ecx, 2
dec ecx
inc [ebx+12]

imul eax, 3
idiv eax, 12

and eax, 0ffh
or eax, 2
xor eax, eax
not eax

neg edx
shl eax, 2
shr eax, 2
```

Data Manipulation Instructions

- *mov* - copies data from source operand to destination
- *push* - pushes value onto top of stack
    - Makes room on the stack by subtracting ESP by 4
        - Stack grows from higher address to lower address
    - Copies the value from operand to stack
- *pop* - removes value from top of stack
    - Copies value from top of the stack
    - Decreases stack size by adding 4 to ESP
- *lea* - “load effective address” of some value in memory
    - Use *[base + index*scale + offset]*
    - Base/index are registers, scale/offset are immediates

Data Manipulation Instructions

- *mov* - copies data from source operanc
- *push* - pushes value onto top of stack
    - Makes room on the stack by subtrac
        - Stack grows from higher addres
    - Copies the value from operand to s
- *pop* - removes value from top of stack
    - Copies value from top of the stacl
    - Decreases stack size by adding 4 t
- *lea* - “load effective address” of some value in memory
    - Use *[base + index*scale + offset]*
    - Base/index are registers, scale/offset are immediates



Control Flow Instructions

- Use labels to mark important sections in data
- *jmp* - unconditional jump to label (ALWAYS happens)
- *cmp* - compares two values and stores metadata in a special register called FLAGS
  - Contains status on last operation
  - *cmp* essentially does *sub* and only modifies FLAGS
- *je/jne/jz/jg/jge/jl/jle* - conditional *jmp* based on FLAGS
- *call* - jumps to label as if it were a function
- *ret* - return from a function call
- *syscall* - call OS level functions for I/O, etc

Control Flow Instructions

- Use labels to mark important sections
- *jmp* - unconditional jump to label (ALL
- *cmp* - compares two values and stores r
  special register called FLAGS
  - Contains status on last operation
  - *cmp* essentially does *sub* and only
- *je/jne/jz/jg/jge/jl/jle* - conditional
- *call* - jumps to label as if it were a
- *ret* - return from a function call
- *syscall* - call OS level functions for I/O, etc

```
jmp label
cmp eax, 2
je equal_label

sub eax, 50
jz zero_label

call printf

ret

mov eax, 1
mov esi, hello_world_label
mov edx, 11
syscall
```

More Instructions

- More instructions here with explanation http://www.felixcloutier.com/x86/
- More here http://ref.x86asm.net/
- C compiler explorer https://godbolt.org/
    - Can type C code and view the disassembly

Calling Convention

- When calling functions, one doesn’t know which registers are used by other function
- Need common way to to save register and pass parameters
- Establish rules for the “caller” and “callee”
- We’ll use a convention inspired by C and is widely used

cdecl convention: caller

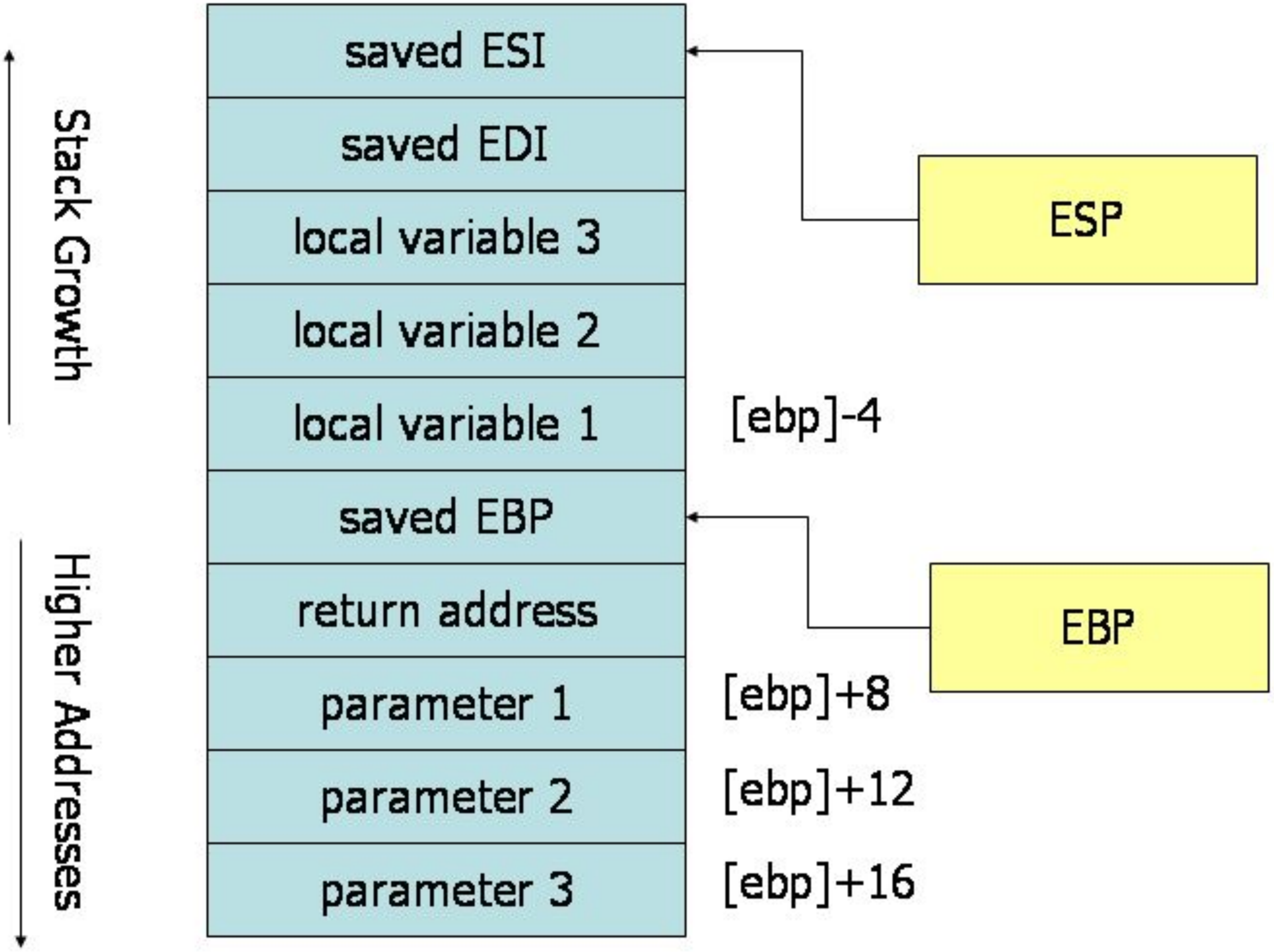
1. Save *caller-saved* registers
    a. push EAX, ECX, and EDX to stack
2. Pass parameters in reverse order
    a. e.g. int f(int a, int b, int c) pushes c, b, then a
3. Call subroutine with *call* instruction

cdecl convention: callee

1. Push EBP to stack, and set EBP to previous ESP value
   a. *push ebp; mov ebp, esp*
2. Allocate space on stack for local variables
   a. *sub esp, 12* (for 3 local integers)
3. Save *callee-saved* registers
   a. pushes EBX, EDI, and ESI to the stack
4. Proceed as normal

cdecl convention: callee

5. When finished, put return value in EAX
6. Restore *callee-saved* registers EBX, EDI, ESI
7. Deallocate local variables
    a. Done by shrinking the stack
    b. Easily accomplished by restoring ESP: *mov esp, ebp*
8. Restore base pointer
    a. Can do *pop ebp* since ESP now points to the old EBP
9. Use *ret* to return to previous code




```
push 36
push [ebx + 4]
push edx

call myFunc

add esp, 12


myFunc:
push ebp
mov ebp, esp
sub esp, 12
push edi
push esi

mov eax, [ebp+8]
mov esi, [ebp+12]
mov edi, [ebp+16]

mov [ebp-4], edi
add [ebp-4], esi
add eax, [ebp-4]

pop esi
pop edi
mov esp, ebp
pop ebp

ret
```

Sections

- Declare a section with *section .sect*
- .*text* - where assembly code goes
- .*data* - where hardcoded data goes
  - Strings
  - Constants
  - Formatting here https://www.tortall.net/projects/yasm/manual/html/nasm-pseudop.html
- .*comment* - comments can go here
- More http://www.tortall.net/projects/yasm/manual/html/objfmt-elf-section.html

Writing and Assembling

- Start program with *global _start* and *section .text*
- Label first line of code with *_start*
- Write code
- Assemble and link

NOTE: since we're using 32 bit assembly, use *int 80h* to call the syscall interrupt

- Normal *syscall* won't work for 32 bit mode
- x86-32 syscall table: https://syscalls.kernelgrok.com/

Writing and Assembling

- Start program with *global _start* and *section .text*
- Label first line of code with *_start*
- Write code
- Assemble and link

NOTE: since we're using 32 bit assembly, u
call the syscall interrupt

- Normal *syscall* won't work for 32 bit mo
- x86-32 syscall table: https://syscalls

```
global _start
section .text

_start:

; stuff goes here

mov eax, 1
mov edi, 0
int 80h
```

Tools

- Assembler - assembly -> machine code
  - *gas* - GNU assembler (AT&T syntax)
  - *nasm* - netwide assembler (Intel syntax)
  - *yasm* - nasm rewrite (AT&T and Intel syntax)
- Disassembler - final binary or machine code -> assembly
  - *objdump* - displays information on object files
- Debugger - debug programs while running live
  - *gdb* - GNU debugger

yasm

- Do *apt-get install yasm* to install
- *yasm -felf32 <file>.s*
  - produces <file>.o
- *ld -m elf_i386 <file>.o -o <file>.x*
  - produces an executable <file>.x

```
global _start
section .text

_start:
mov eax, 1
mov edi, 0
int 80h
```

objdump

- *objdump <flags> <file>*
  - *-D* will disassemble the given object file
  - *-Mintel* will output with Intel syntax

```
[j@b0x:~][130]$ cat test.s                                        (04-20 09:20)


    global _start
    section .text

    _start:
    mov eax, 1
    mov edi, 0
    int 80h
[j@b0x:~]$ yasm -felf32 test.s                                    (04-20 09:20)
[j@b0x:~]$ ld -m elf_i386 test.o                                  (04-20 09:20)
[j@b0x:~]$ objdump -D -Mintel a.out                               (04-20 09:20)

a.out:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       b8 01 00 00 00          mov    eax,0x1
 8048065:       bf 00 00 00 00          mov    edi,0x0
 804806a:       cd 80                   int    0x80
```

activity

Two exercises

- First, download examples from git repo
  - *git clone https://github.com/UMD-CS-STICs/389Rspring18.git*
  - *cd 389Rspring18/week/12/examples*
  - *make* -- make sure this prints “Hello There” before the second exercise!
- Second, try to reverse engineer this assembly
  - Located in examples/exercise.s
  - Let us know when you have the value in EAX!

homework #10

will be posted soon.

Let us know if you have any questions!

This assignment has 2 parts.

It is due by 4/26 at 11:59PM.