CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework:
Hexadecimal Math Redux

Hexadecimal Math

Modify your prior solution to the binary math problem, or start with the provided solution, so that the output is in hexadecimal instead of binary. It is *suggested* to use *lower case* letters in your hexadecimal (they are easier to distinguish from digits). Place your code in a file called `addsub.asm` and make sure to correct the first line to correctly compile your code!

```
$ ./addsub 9000000000000000000 8999999999999999999
Adding:
7ce66c50e2840000
7ce66c50e283ffff
f9ccd8a1c507ffff
Subtracting:
7ce66c50e2840000
7ce66c50e283ffff
0000000000000001
```

Uses a look-up table

The offset from nyb gives the correct hex digit.

```
        section .data
nyb     db "0123456789abcdef"
```

Use successive division

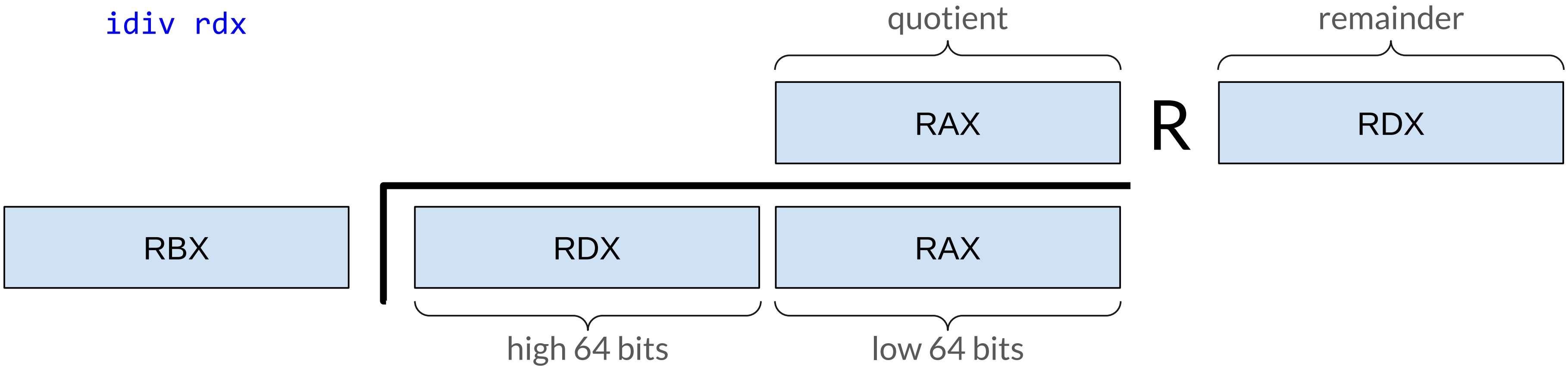
We divide by 16 each time through the loop and convert the remainder into a hex digit. We save them all in a reserved space on the stack, then print directly from the stack.

```
write_hex_qword:
        push rbp
        mov rbp, rsp
        mov rcx, 16
        push r14
        mov r14, rdi
        sub rsp, 16
.top:   mov rbx, 16
        mov rax, r14
        div rbx
        mov r14, rax
        mov al, BYTE [nyb+rdx]
        mov BYTE [rsp+rcx], al
        loop .top
        mov rdi, 1
        lea rsi, [rsp+1]
        mov rdx, 16
        mov rax, 1
        syscall
        pop r14
        leave
        ret
```

Division

one operand, two arguments, three registers

```
div rdx
idiv rdx
```



Division

You have to pay attention to what is in RDX.

In this case RDX holds 15 and RAX holds 16. The number is 15*(2**64)+16. In hex this is:
`0xf 0000 0000 0000 0010`

Dividing this number by 16 yields a quotient of:
`0xf000 0000 0000 0001`
and a remainder of zero.

```
; nasm -f elf64 div1.asm && ld -o div1 div1.o

            section .text
            global _start

_start:     mov rax, 16
            mov rdx, 15
            mov rbx, 16
            div rbx
            mov rdi, rax
            mov rax, 60
            syscall
            hlt
```

Division

The quotient *fits into* RAX. What do we see when we print the return value?

```
$ ./div1 ; echo $?
1
```

Why don't we see the full quotient (it is much larger)?

```
; nasm -f elf64 div1.asm && ld -o div1 div1.o

          section .text
          global _start

_start:   mov rax, 16
          mov rdx, 15
          mov rbx, 16
          div rbx
          mov rdi, rax
          mov rax, 60
          syscall
          hlt
```

Division

The quotient *fits into* RAX. What do we see when we print the return value?

```
$ ./div1 ; echo $?
1
```

Why don't we see the full quotient (it is much larger)? The return value is 32 bits, so we only see what is in EAX, and that's just 1.

```
; nasm -f elf64 div1.asm && ld -o div1 div1.o

          section .text
          global _start

_start:   mov rax, 16
          mov rdx, 15
          mov rbx, 16
          div rbx
          mov rdi, rax
          mov rax, 60
          syscall
          hlt
```

Division

We can make a minor change.

In this case RDX holds 16 and RAX holds 16. The number is 16*(2**64)+16. In hex this is:
`0x10 0000 0000 0000 0010`

Dividing this number by 16 yields a quotient of:
`0x1 0000 0000 0000 0001`
and a remainder of zero.

```
; nasm -f elf64 div2.asm && ld -o div2 div2.o

          section .text
          global _start

_start:   mov rax, 16
          mov rdx, 16
          mov rbx, 16
          div rbx
          mov rdi, rax
          mov rax, 60
          syscall
          hlt
```

Division

This quotient does *not* fit into RAX. We cannot leave part of it in RDX, since we have to store the remainder (zero in this case) in RDX.

What happens? Do we just lose the high bits of the quotient?

```
; nasm -f elf64 div2.asm && ld -o div2 div2.o

          section .text
          global _start

_start:   mov rax, 16
          mov rdx, 16
          mov rbx, 16
          div rbx
          mov rdi, rax
          mov rax, 60
          syscall
          hlt
```

Division

What happens? Do we just lose the high bits of the quotient?

```
$ ./div2
Floating point exception (core dumped)
```

There is no floating point math in here!

```
; nasm -f elf64 div2.asm && ld -o div2 div2.o

          section .text
          global _start

_start:   mov rax, 16
          mov rdx, 16
          mov rbx, 16
          div rbx
          mov rdi, rax
          mov rax, 60
          syscall
          hlt
```

Division

It's what's in the specification. In this case it is just a divide error (DE), which is reported as a floating point exception by Linux.

Protected Mode Exceptions ¶

#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Code Example

Simple Definitions

Define constants with `equ`. Define simple macros with `%define`.

Simple macros can take arguments, which are substituted in the body. As with CPP, be wary of ambiguity.

```
SYS_WRITE equ 1
STDOUT equ 1
STDERR equ 2

%define numerator QWORD [rbp-8]
%define denominator QWORD [rbp-16]
%define argv(index) QWORD [rsi+8*(index)]
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

```
%macro       err 2
             ; Macro to declare an error message.  This
declares an
             ; error message and also an errno value to
return.  The
             ; string length and errno value are local
values.
             %00: db %1
             .len equ $-%00
             .errno equ %2
%endmacro
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

Macro name

```
%macro        err 2
              ; Macro to declare an error message.  This
declares an
              ; error message and also an errno value to
return.  The
              ; string length and errno value are local
values.
              %00: db %1
              .len equ $-%00
              .errno equ %2
%endmacro
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

Two arguments

```
%macro          err 2
                ; Macro to declare an error message.  This
declares an
                ; error message and also an errno value to
return.  The
                ; string length and errno value are local
values.
                %00: db %1
                .len equ $-%00
                .errno equ %2
%endmacro
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

The label in front of the invocation… if any

```
%macro          err 2
                ; Macro to declare an error message.  This
declares
                ; error message and also an errno value to
return.  The
                ; string length and errno value are local
values.
                %00: db %1
                .len equ $-%00
                .errno equ %2
%endmacro
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

First argument

```
%macro      er
            ;   ro to declare an error message.  This
declares an
            ;   ror message and also an errno value to
return.  The
            ; string length and errno value are local
values.
            %00: db %1
            .len equ $-%00
            .errno equ %2
%endmacro
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

Second argument

```
%macro      err
            ; Macro to declare an error message.  This
declares an
            ; error message and also an errno value to
return.  The
            ; string length and errno value are local
values.
            %00: db %1
            .len equ $-%00
            .errno equ %2
%endmacro
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

```
%macro        err 2
              ; Macro to declare an error message.  This
declares an
              ; error message and also an errno value to
return.  The
              ; string length and errno value are local
values.
              %00: db %1
              .len equ $-%00
              .errno equ %2
%endmacro
```

Applying...

```
BADARG: err {'ERROR: Expect exactly two arguments.',10}, 1
```

The brackets protect the argument; otherwise the comma would indicate a second argument.

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

```
%macro       err 2
             ; Macro to declare an error message.  This
declares an
             ; error message and also an errno value to
return.  The
             ; string length and errno value are local
values.
             %00: db %1
             .len equ $-%00
             .errno equ %2
%endmacro
```

Applying...

```
BADARG: err {'ERROR: Expect exactly two arguments.'  , 1
```

The second argument is actually here.

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

```
%macro       err 2
             ; Macro to declare an error message.  This
declares an
             ; error message and also an errno value to
return.  The
             ; string length and errno value are local
values.
             %00: db %1
             .len equ $-%00
             .errno equ %2
%endmacro
```

Applying...

```
BADARG: err {'ERROR: Expect exactly two arguments.',10}, 1
```

Yields...

```
      %00: db %1
      .len equ $-%00
      .errno equ %2
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

```
%macro        err 2
              ; Macro to declare an error message.  This
declares an
              ; error message and also an errno value to
return.  The
              ; string length and errno value are local
values.
              %00: db %1
              .len equ $-%00
              .errno equ %2
%endmacro
```

Applying...

```
BADARG: err {'ERROR: Expect exactly two arguments.',10}, 1
```

Yields...

```
      BADARG: db %1
      .len equ $-BADARG
      .errno equ %2
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

```
%macro      err 2
            ; Macro to declare an error message.  This
declares an
            ; error message and also an errno value to
return.  The
            ; string length and errno value are local
values.
            %00: db %1
            .len equ $-%00
            .errno equ %2
%endmacro
```

Applying...

```
BADARG: err {'ERROR: Expect exactly two arguments.',10}, 1
```

Yields...

```
BADARG: db 'ERROR: Expect exactly two arguments.',10
.len equ $-BADARG
.errno equ %2
```

Macros

NASM has a powerful, but somewhat obscure, macro facility.

Lots of trickery.

```
%macro       err 2
             ; Macro to declare an error message.  This
declares an
             ; error message and also an errno value to
return.  The
             ; string length and errno value are local
values.
             %00: db %1
             .len equ $-%00
             .errno equ %2
%endmacro
```

Applying...

```
BADARG: err {'ERROR: Expect exactly two arguments.',10}, 1
```

Yields...

```
      BADARG: db 'ERROR: Expect exactly two arguments.',10
      .len equ $-BADARG
      .errno equ 1
```

Macros

If you want to see how macros are expanded, run NASM with `-e`. This runs the preprocessor and stops.

```
%line 232+0 division.asm


 BADARG: db 'ERROR: Expect exactly two arguments.',10
 .len equ $-BADARG
 .errno equ 1
%line 233+1 division.asm
```

Homework:
Entry Point
Due: Tuesday, March 3

Entry Point

The entry point is a property of the ELF file (it is found in the ELF file header).

Code is organized into sections, and each section ends up getting a starting virtual address and a length. The entry point should be in exactly one section.

Information about the section is in the section header.

```
# Get the segment top and the entry point.
top_addr = section.header.sh_addr
entry_point = elf.header.e_entry

# Compute the *offset* from the top to the entry point.
offset = entry_point - top_addr
if offset < 0 or offset >= section.header.sh_size:
    print("Entry point not in .text section.")
    exit()

# Disassemble.
for i in md.disasm(code[offset:], entry_point):
    print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
    if 1 in i.groups or 7 in i.groups:
        branches = True
```

Basic Blocks

Basic Block

A **basic block** is a straight-line code sequence with *no branches in* except to the entry (a single entry) and *no branches out* except at the exit (a single exit). Basic blocks form the *vertices* in a *control flow graph*.

A basic block is ended by: a non-returning syscall (but not most calls), a conditional branch, a jump, and a halt. A basic block is *broken into two* if there is a jump into the block from outside it.

The *edges* between basic blocks are the branches and jumps.

Basic Block Algorithm

You can do this in two passes.

Pass 1: Identify leaders. An address is a leader iff it is the first instruction (the entry point or one of the given addresses), it is the target of a *conditional* or *unconditional* branch, it is the address immediately after a conditional branch, it is the target of a call, or it is the address immediately after a call.

Pass 2. Starting from each leader, the sequence of all instructions to the next basic block terminator *or* leader is the basic block for that leader.

Basic Block Algorithm

Instructions that *terminate* a basic block are the following:

- unconditional branches (`jmp`)
- the `hlt`, `ret`, and `iret` instructions
- an interrupt
- conditional branches
- looping constructs like `loop`

Homework:
Due: Tuesday, March 10

Basic Blocks

Modify your entry_point.py code (or start with the provided solution) to identify basic blocks. Call your program `basic_blocks.py`. Store the basic blocks and then, at the end, print them in order by address.

Accept a file name followed by a (possibly empty) series of hexadecimal addresses as command line arguments and assume these are basic block leaders. If *no addresses* are given on the command line, add the entry point to the stack as a basic block leader.

Print the block leader address, the block disassembly indented two spaces, and the next address(es) after the block. If you don't know the addresses, print "unknown."

Basic Blocks

The first time you run this for a program that uses the C runtime, you will probably get a single basic block.

What should you do next?

```
$ python3 basic_blocks.py `which python3`
/usr/bin/python3:

block at: 0x5cff70
  endbr64
  xor     ebp, ebp
  mov     r9, rdx
  pop     rsi
  mov     rdx, rsp
  and     rsp, 0xfffffffffffffff0
  push    rax
  push    rsp
  mov     r8, 0x679500
  mov     rcx, 0x679490
  mov     rdi, 0x4cf960
  call    qword ptr [rip + 0x26905a]
  hlt
next: unknown
```

Basic Blocks

The first time you run this for a program that uses the C runtime, you will probably get a single basic block.

What should you do next?

```
$ python3 basic_blocks.py \
  `which python3` 0x4cf960
```

```
$ python3 basic_blocks.py `which python3`
/usr/bin/python3:

block at: 0x5cff70
  endbr64
  xor     ebp, ebp
  mov     r9, rdx
  pop     rsi
  mov     rdx, rsp
  and     rsp, 0xfffffffffffffff0
  push    rax
  push    rsp
  mov     r8, 0x679500
  mov     rcx, 0x679490
  mov     rdi, 0x4cf960
  call    qword ptr [rip + 0x26905a]
  hlt
next: unknown
```

Random Access Disassembly Class

```
class RAD:
    def __init__(self, code, arch, bits, offset):
        self.md = Cs(arch, bits)
        self.md.skipdata = True
        self.md.detail = True
        self.code = code
        self.offset = offset
        self.size = len(code)

    def at(self, address):
        index = address - self.offset
        if index < 0 or index >= self.size:
            raise AddressException(address, self.offset, self.size)
        return next(self.md.disasm(self.code[index:index+15], address, count=1))

    def in_range(self, address):
        index = address - self.offset
        return index >= 0 and index < self.size
```

Instruction Groups

You'll need to look at the *instruction group*. An instruction can be in multiple groups; be careful about the order you test.

These can be found in the Capstone source.

https://github.com/aquynh/capstone/blob/master/include/capstone/x86.h

To find out what particular instruction you have you can check the *mnemonic*.

```
/// Group of X86 instructions
typedef enum  x86_insn_group {
        X86_GRP_INVALID = 0, ///< = CS_GRP_INVALID
        // Generic groups
        // all jump instructions (conditional+direct+indirect jumps)
        X86_GRP_JUMP,   ///< = CS_GRP_JUMP
        // all call instructions
        X86_GRP_CALL,   ///< = CS_GRP_CALL
        // all return instructions
        X86_GRP_RET,    ///< = CS_GRP_RET
        // all interrupt instructions (int+syscall)
        X86_GRP_INT,    ///< = CS_GRP_INT
        // all interrupt return instructions
        X86_GRP_IRET,   ///< = CS_GRP_IRET
        // all privileged instructions
        X86_GRP_PRIVILEGE,      ///< = CS_GRP_PRIVILEGE
        // all relative branching instructions
        X86_GRP_BRANCH_RELATIVE, ///< = CS_GRP_BRANCH_RELATIVE
```

Instruction Groups

`X86_GRP_JUMP`	1	Jump target is a leader. End block.
`X86_GRP_CALL`	2	Call target and next instruction are leaders. End block.
`X86_GRP_RET`	3	End block.
`X86_GRP_INT`	4	Instruction after interrupt is a leader.
`X86_GRP_IRET`	5	End block.
`X86_GRP_PRIVILEGE`	6	Ignore.
`X86_GRP_BRANCH_RELATIVE`	7	Branch target and next instruction are leaders. End block.
Otherwise		Ignore.

Resolving RIP-based Addresses

You have to decide what to do for each group.

```
nextaddr = i.address + len(i.bytes)
# ...
if i.group(2):
  # This is a call.  Both the call target and the
  # instruction after this are potential leaders.
  if is_imm(i.operands[0]):
    # The call target will be a leader.
    leaders.append(i.operands[0].value.imm)
    leaders.append(nextaddr)
  elif is_mem(i.operands[0]):
    # We can only handle RIP-based addressing.
    if i.reg_name(i.operands[0].value.mem.base) == 'rip':
      # Now we can compute the address of the call.
      leaders.append(nextaddr+i.operands[0].value.mem.disp)
      leaders.append(nextaddr)
# ...
```

Next time:
Live Values and Slicing