Introduction to 64 Bit Intel Assembly Language Programming

Goals for Cos 284's assembly component

- Learn internal data formats
- Learn basic 64 bit Intel/AMD instructions
- Write pure assembly programs
- Write mixed C/C++ and assembly programs
- Use the `gdb` debugger for ASM
- Floating point instructions
- Arrays
- Functions
- Structs
- Using system calls, both directly and indirectly
- Data structures and high performance ASM

Problems with assembly language

- Assembly is the poster child for non-portability
  - Different CPU = different assembly instruction set.
  - Different OS = different function ABI (application binary interface)
  - Intel/AMD CPUs operate in 16, 32, and 64 bit modes
- Difficult to program
  - More time = more money
  - Less reliable as it is very error prone.
  - Difficult to maintain
- Syntax does not resemble mathematics
- No syntactic protection
  - No structured ifs, loops
- No typed variables
  - Can use a pointer as a floating point number
  - Can load a 4 byte integer from a double variable
- Variable access is roughly like using pointers

What's good about assembly language?

- Assembly language is fast
  - Optimizing C/C++ compilers will be faster than a novice most of the time.
  - You need to dissect an algorithm and rearrange it to use a special feature that the compiler can't figure out
  - Generally you must use a special instructions
  - There are over 1000 instructions
  - Still it can be faster
- Assembly programs are small
  - But memory is cheap and plentiful
  - C/C++ compilers can optimize for size
  - Compilers can re-order code sections to reduce size
- Assembly can do things not possible in C/C++
  - I/O instructions
  - Manage memory mapping registers
  - Manipulate other internal control registers

What's good about assembly for ordinary mortals?

- Teaches you how the programs really works
- Teaches you how storage and arithmetic is done in registers
- Teaches you C/C++ function register and stack usages
- Teaches you how stack frames are built and destroyed.
- Optimization techniques are explained.
- Computer bugs are more immediately related to machine instructions and limitations
- You will learn how the compiler implements
  - if/else statements
  - loops
  - functions
  - structures
  - arrays
  - recursion
- Your coding will improve.

Generation of languages

- First generation - machine language
- Second generation - assembly language
  - Names for instructions
  - Names for variables
  - Names for locations of instructions
  - Perhaps with macros - code replacement
- Third generation - not machine instructions
  - Modeled after mathematics - Fortran
  - Modeled after English - Cobol
  - List processing - Lisp
- Fourth generation - domain specific
  - SQL
- Fifth generation - describe problem, computer generates algorithm
  - Prolog

Assembly example

```
;   Program: exit
;
;   Executes the exit system call
;
;   No input
;
;   Output: only the exit status ($? in the shell)
;
    segment .text
    global  _start
_start:
    mov   eax,1        ; 1 is the exit syscall number
    mov   ebx,5        ; the status value to return
    int   0x80         ; execute a system call
```

Assembly syntax

- ; starts comments
- Labels are strings which are not instructions
  - Usually start in column 1
  - Can end with a colon to avoid confusion with instructions
- Instructions can be machine instructions or assembler instructions
  - `mov` and `int` are machine instructions or opcodes
  - `segment` and `global` are assembler instructions or pseudo-ops
- Instructions can have operands
  - `here:  mov eax, 1`
  - `here` is a label for the instruction
  - `mov` is an opcode
  - `eax` and `1` are operands

Some assembler instructions

- `section` or `segment` define a part of the program
  - `.text` is where instructions go for Linux
- `global` defines a label to be used by the linker
- `global _start` makes `_start` a global label
- `_start` or `main` is where a program starts
  - `_start` is more basic
  - `main` is called (perhaps indirectly) by `_start`

Assembling the exit program

- `yasm -f elf64 -g dwarf2 -l exit.lst exit.asm`
- `-f elf64` says we want a 64 bit object file (elf=extensible linking format)
- `-g dwarf2` says we want `dwarf2` debugging info (why dwarf?)
  - `dwarf2` works pretty well with the gdb debugger
- `-l exit.lst` asks for a listing in `exit.lst`
- `yasm` will produce `exit.o`, an object file
  - machine instructions not ready to execute

exit.lst

```
 1                                          %line 1+1 exit.asm
 2
 3
 4
 5
 6
 7
 8
 9
10                                          [segment .text]
11                                          [global _start]
12
13                                          _start:
14 00000000 B801000000                       mov eax,1
15 00000005 BB05000000                       mov ebx,5
16 0000000A CD80                             int 0x80
```

Linking

- Linking means combining object files to make an executable file
- For programs with `_start`
  - `ld -o exit exit.o`
  - Builds a file named `exit`
  - Default is `a.out`
- For programs with `main`
  - `gcc -o exit exit.o`
  - Gets default `_start` function from the C library
- `./exit` to run the program

Floating point numbers

Consider 1.75, in 32bit-IEEE 754 the number becomes:

```
0        01111111 11000000000000000000000
Positive 127      (1).75
```

Grouping into 4 bit nibbles:

```
0011 1111 1110 0000 0000 0000 0000 0000
3    f    e    0    0    0    0    0
```

But this is stored reversed and with each nibble pair swapped:

```
0    0    0    0    e    0    3    f
```

listings example part 1

Consider the following asm file “fp.asm”.

```
            segment .data
zero        dd      0.0
one         dd      1.0
neg1        dd      -1.0
a           dd      1.75
b           dd      122.5
d           dd      1.1
e           dd      10000000000.0
```

The **dd** command specifies a double word data item. A word is 2 bytes. So a double word is 32 bits.

- **dw** is a data word
- **db** is a byte
- **dq** is a data quad-word

listings example part 1

Now if we create the file listing using:

```
yasm -f elf64 -g dwarf2 -l fp.lst fp.asm
```

The result is:

```
 1                                  %line 1+1 fp.asm
 2                                  [section .data]
 3 00000000 00000000                zero dd 0.0
 4 00000004 0000803F                one dd 1.0
 5 00000008 000080BF                neg1 dd -1.0
 6 0000000C 0000E03F                a dd 1.75
 7 00000010 0000F542                b dd 122.5
 8 00000014 CDCC8C3F                d dd 1.1
 9 00000018 F9021550                e dd 10000000000.0
```

Memory mapping

- Computer memory is an array of bytes from 0 to $n-1$ where n is the memory size
- Programs perceive "logical" addresses which are mapped to physical addresses
- 2 people can run a program starting at logical address `0x4004c8` while using different physical memory
- CPU translates logical addresses to physical during instruction execution
- The CPU translation can be just as fast as if the software used physical addresses
- The x86-64 CPUs can map pages of sizes 4096 bytes and 2 megabytes
- Linux uses 2 MB pages for the kernel and 4 KB pages for programs
- Some recent CPUs support 1 GB pages

Translating an address

- Suppose an instruction references address `0x43215628`
- With 4 KB pages, the rightmost 12 bits are an offset into a page
- With `0x43215628` the page offset is `0x628`
- The page number is `0x43215`
- Let's assume that the computer is set up to translate page `0x43215` to physical addresses `0x7893000` - `0x7893fff`
- Then address `0x43215628` is mapped to `0x7893628`

Benefits of memory mapping

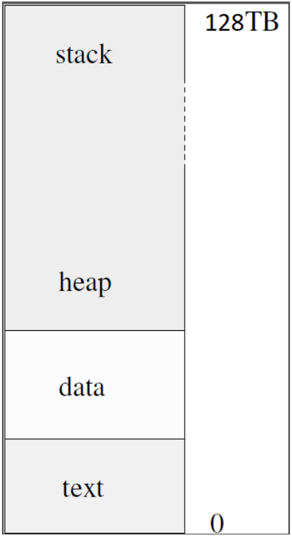
- User processes are protected from each other
  - Your process can't read my process's data
  - Your process can't write my data
- The operating system is protected from malicious or errant code
- It is easy for the operating system to give processes contiguous chunks of "logical" memory

Why study memory mapping?

- If you write programs, the mapping is automatic
- We will not discuss instructions for changing mapping tables
- So what difference does it make?
- It helps explain page faults
  - Suppose you allocate an array of 256 bytes at logical address `0x45678200`
  - Then all addresses from `0x45678000` to `0x45678fff` are valid
  - You can go well past the end of the array before you can get a segmentation violation
- Knowledge is power!

Process memory model in Linux

- A Linux process has 4 logical segments
  - text: machine instructions
  - data: static data initialized when the program starts
  - heap: data allocated by `malloc` or `new`
  - stack: run-time stack
    - return addresses
    - some function parameters
    - local variables for functions
    - space for temporaries
- In reality it is slightly more complex
- 128TB is 47 bits of all 1's (≈141 base 10, TB)
- CPU could use 48 bit logical addresses



Memory segments

- The text segment is named `.text` in `yasm`
  - `_start` and `main` are not actually at 0
  - The text segment does not need to grow, so the data segment can be placed immediately after it
- The data segment is in 2 parts
  - `.data` which contains initialized data
  - `.bss` which contains reserved data
  - "bss" stands for "Block Started by Symbol"
- The heap and the stack both need to grow
  - The heap grows up
  - The stack grows down
  - They meet in the middle and explode

Stack segment limits

- The stack segment is limited by the Linux kernel
- The typical size is 16 MB for 64 bit Linux
- This can be inspected using “`ulimit -a`” or “`ulimit -s`’
- 16 MB seems fairly small, but it is fine until you start using large arrays as local variables in functions
- The stack address range is `0x7ffff000000` to `0x7ffffffffff`
- A fault to addresses in this range are recognized by the kernel to allow the stack to grow as needed

Memory example source code

```
        segment .data
a       dd      4
b       dd      4.4
c       times   10 dd 0
d       dw      1, 2
e       db      0xfb
f       db      "hello world", 0

        segment .bss
g       resd    1
h       resd    10
i       resb    100
```

Memory example listing file

```
1                                           %line 1+1 memory.asm
2                                           [section .data]
3 00000000 04000000                         a dd 4
4 00000004 CDCC8C40                         b dd 4.4
5 00000008 00000000<rept>                   c times 10 dd 0
6 00000030 01000200                         d dw 1, 2
7 00000034 FB                               e db 0xfb
8 00000035 68656C6C6F20776F72-              f db "hello world", 0
9 00000035 6C6400
```

- Addresses are relative to start of `.data` in this file
- b = 0x408ccccd = 0 10000001 00011001100110011001101
- Sign bit is 0, exponent field is 0x81 = 129, $\Longrightarrow$ actual exponent = 2
- Fraction is 1.00011001100110011001101

Memory example listing file (2)

```
11                                        [section .bss]
12 00000000 <gap>                         g resd 1
13 00000004 <gap>                         h resd 10
14 0000002C <gap>                         i resb 100
```

- Notice that the addresses start again at 0
- The commands reserve space
- `resd 1` reserves 1 double word or 4 bytes
- `resd 10` reserves 10 double words or 40 bytes
- `resb 100` reserves 100 bytes

Examining memory

Useful tools to examine memory are:

- `gdb`
- `ebe`

Memory Mapping in 64 Bit Mode and Registers

Chapter 4

Page

- What is page?

Page

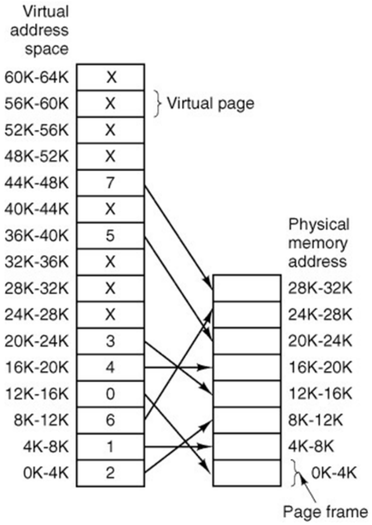
- What is page?
  - A page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table.
  - It is the smallest (usually) unit of data for memory management in a virtual memory operating system.
  - In our case we will consider 4kb pages.

Page

- What is page?
  - A page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table.
  - It is the smallest (usually) unit of data for memory management in a virtual memory operating system.
  - In our case we will consider 4kb pages.
  - Check with `getconf PAGESIZE`.
- How is the mapping of a page between virtual address space and physical address space done?

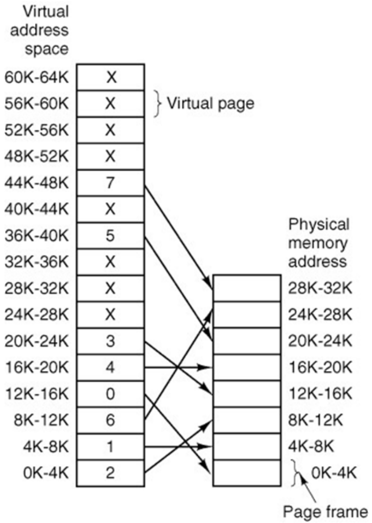
Single level paging

- To the right we have an example of a single layer paging system
- The X's indicate that the virtual page does not at present have a corresponding physical page.
  - If a program requires a virtual page marked with an X this causes a page fault.
  - The system must then allocate some physical memory to assign this virtual page to.
    - But what if there is no space left?



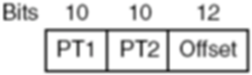
Single level paging

- The primary problems with a single level paging system is the size of the table.
  - Consider the linux process model with a virtual memory space of 128TB that would imply a table with $2^{(47-12)} = 2^{35}$ entries.
  - About 34 billion entries!
  - Assuming a 1GB of physical memory we would need to address $2^{(30-12)} = 2^{18}$ physical 4k pages.
    - So we would need 18-bits plus 1 bit to indicate if the entry is valid or invalid.
  - Just this table will occupy $2^{35} * 19$ bits, which is about 81.6 GB.



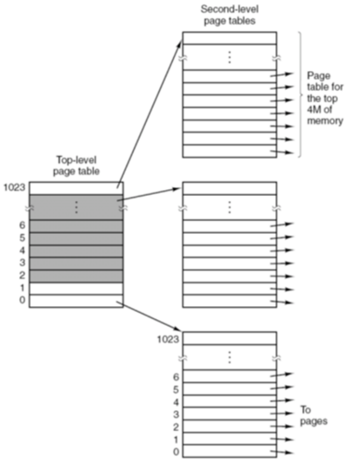
Multilevel Page Tables

- To get around the problem of having to store huge page tables in memory all the time, many computers use a multilevel page table.
- As a simple example consider we have a 32-bit virtual address (4GB) ,that is partitioned into a
  - 10-bit PT1 field
  - 10-bit PT2 field,
  - and a 12-bit Offset field (for the 4k pages).
- In general if we used PT1+PT2 together we would be working 2^{20} pages
  - That's a lot of pages to keep in memory



Multilevel Page Tables

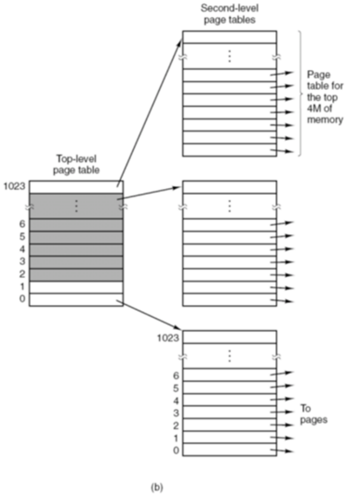
- The **top** level page table corresponds to the first 10-bits.
  - which maps to 1024 page tables
- The **second** level page table corresponds to the second 10-bits
  - Which maps to the 1024 pages of size 4K
- The last remaining 12 bits are used as the offset to address the contents of the 4K page.



(b)

Multilevel Page Tables

- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
  - In particular, those that are not needed should not be kept around.
- By marking elements of the top-level page table as absent we do not need to maintain (or store) all the second level page tables, saving substantially on space.



Memory mapping pages and tables

- Linux uses 4 layer page tabling.
- Each page is $2^{12} = 4096$ bytes
- An address is 8 bytes (not all used)
- Each page table can hold $2^9 = 512$ addresses
- A 9 bit field is needed to index the mapping tables
- Current mapping uses 48 bits, so we are limited to 2^{48} bytes which is 256 TB

Logical memory address fields

63–48	47–39	38–30	29–21	20–12	11–0
unused	PML4 index	page directory pointer index	page directory index	page table index	page offset

- Bits 47-39 are used to index the PML4 table
- Bits 38-30 are used to index the selected page directory pointer table
- Bits 29-21 are used to index the selected page directory table
- Bits 20-12 are used to index the selected page table
- Bits 11-0 are the offset into the page (for 4 KB pages)

Large pages

- Using the first 3 existing levels of page tables, we can have large pages with $2^{21} = 2097152$ bytes.
- This is used by Linux for the kernel

CPU support for fast lookups

- A CPU uses a special cache called a "Translation Lookaside Buffer" or TLB to speed up memory translation
- A TLB operates much like a hash table
- Presented with a logical address, it produces the physical address or failure in about 1/2 a clock cycle
- The Intel Core i7 has 2 levels of TLBs
  - Level 1 holds 64 small page translations (or 32 big pages)
  - Level 2 holds 512 page translations
  - Large programs with small pages will experience TLB misses which can be satisfied fairly rapidly with normal cache
  - Very large programs can crawl

Chapter 5

Register basics

- Computer main memory has a latency of about 80 nanoseconds
- A 3.3 GHz CPU uses approximately 0.3 nsecs per cycle
- Memory latency is about 240 cycles
- The Core i7 has 3 levels of cache with different latencies
  - Level 3 about 48 nsec latency or about 150 cycles
  - Level 2 about 10 nsec latency or about 39 cycles
  - Level 1 about 4 nsec latency or about 12 cycles
- There is a need for even faster memory
- This ultra-fast “memory” is the CPU’s registers
- Some register-register instructions complete in 1 cycle

x86-64 registers

- CPUs running in x86-64 mode have 16 general purpose registers
- There are also 16 floating point registers (XMM0-XMM15)
- There is also a floating point register stack which we ignore
- The general purpose registers hold 64 bits
- The floating point registers can be either 128 or 256 bits
    - The CPU can use them to do 1 32 bit or 1 64 bit floating point operation in an instruction
    - The CPU can also use these to do packed operations on multiple integer or floating point values in an instruction
    - "Single Instruction Multiple Data" - SIMD
- The CPU has a 64 bit instruction pointer register - `rip`
    - contains the address of the next instruction to execute.
- There is a 64 bit flags register, `rflags`, holding status values like whether the last comparison was positive, zero or negative

General purpose registers

- These registers evolved from 16 bit CPUs to 32 bit mode and finally 64 bit mode
- Each advance has maintained compatibility with the old instructions
- The old register names still work
- The old collection was 8 registers which were not entirely general purpose
- The 64 bit collection added 8 completely general purpose 64 bit registers named `r8` - `r15`

The 64 bit registers evolved from the original 8

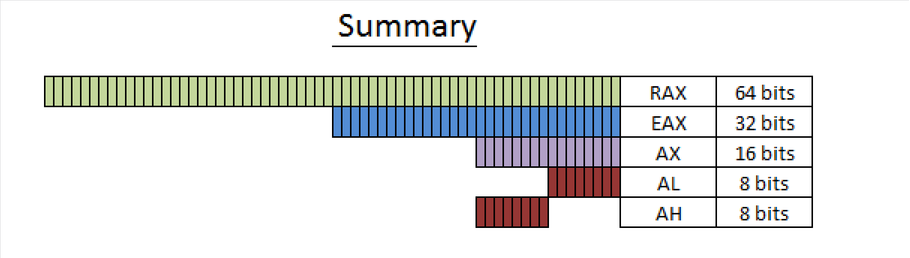
- Software uses the “r” names for 64 bit use, the “e” names for 32 bit use and the original names for 16 bit use
- `rax` - general purpose, accumulator
  - `rax` uses all 64 bits
  - `eax` uses the low 32 bits
  - `ax` uses the low 16 bits
- `rbx`, `ebx`, `bx` - general purpose
- `rcx`, `ecx`, `cx` - general purpose, count register
- `rdx`, `edx`, `dx` - general purpose
- `rdi`, `edi`, `di` - general purpose, destination index
- `rsi`, `esi`, `si` - general purpose, source index
- `rbp`, `ebp`, `bp` - general purpose, stack frame base pointer
- `rsp`, `esp`, `sp` - stack pointer, `rsp` is used to push and pop

The original 8 registers as bytes

- Kept from the 16 bit mode
  - `al` is the low byte of `ax`, `ah` is the high byte
  - `bx` can be used as `bl` and `bh`
  - `cx` can be used as `cl` and `ch`
  - `dx` can be used as `dl` and `dh`
- New to x86-64
  - `dil` for low byte of `rdi`
  - `sil` for low byte of `rsi`
  - `bpl` for low byte of `rbp`
  - `spl` for low byte of `rsp`
- There is no special direct way to access any "higher" bytes of registers

Visual summary

Break down of a 64-bit register.



The 8 new general purpose registers as smaller registers

- Here the naming convention changes
- Appending “d” to a register accesses its low double word - `r8d`
  - double word = 4 bytes = 32 bits.
- Appending “w” to a register accesses its low word - `r12w`
  - single word = 2 bytes = 16 bits.
- Appending “b” to a register accesses its low byte - `r15b`

Moving a constant into a register

- Moving is fundamental
- `yasm` uses the mnemonic `mov` for all sorts of moves
- The generated code from `gcc` uses mnemonics like `movq`
  - gcc uses AT&T syntax by default
- Most instructions can use 1, 2 or 4 byte immediate fields
- `mov` can use an 8 byte immediate value.

```
mov rax, 0x0123456789abcdef ; can move 8 byte immediates
mov rax, 0
mov eax, 5                  ; the upper half is set to 0
mov r8w, 16                 ; affects only low word
```

Moving a value from memory into a register

```
    segment .data
a       dq      175
b       dq      4097
c       db      1, 2, 3, 4, 5, 6, 7, 8
d       dd      0xffffffff
    segment .text
    mov     rax, a
    mov     rbx, [a]
    mov     rcx, [c]
    mov     edx, [d]
```

- Using simply **a** places the address of **a** into **rax**
- Using **[a]** places the value of **a** into **rbx**

A program to add 2 numbers from memory

```
    segment .data
a       dq      175
b       dq      4097
    segment .text
    global  main
main:
    mov     rax, [a]    ; mov a into rax
    add     rax, [b]    ; add b to rax
    xor     rax, rax
    ret
```

Move with sign extend or zero extend

- If you move a double word into a double word register (e.g. eax), the upper half is zeroed out
- If you move a 32 bit immediate into a 64 bit register it is sign extended
- Sometimes you might wish to load a smaller value from memory and fill the rest of the register with zeroes
- Or you may wish to sign extend a small value from memory
- For `movsx` and `movzx` you need a size qualifier for the memory operand

```
movsx  rax, byte [data]      ; move byte, sign extend
movzx  rbx, word [sum]       ; move word, zero extend
movsxd  rcx, dword [count]    ; move dword, sign extend
```

Moving values from a register into memory

- Simply use the memory reference as the first operand

```
mov    [a], rax        ; move a quad word to a
mov    [b], ebx        ; move a double word to b
mov    [c], r8w        ; move a word to c
mov    [d], r15b       ; move a byte to d
```

Moving data from one register to another

- Use 2 register operands

```
mov     rax, rbx     ; move rbx to rax
mov     eax, ecx     ; move ecx to eax, zero filled
mov     cl, al       ; move al to cl, leave rest of
; unchanged
```

A Little Bit of Math

A little bit of math

- So far we have learned how to get values into registers
- And how to place them back into memory
- Just some ordinary arithmetic can help us write slightly more useful programs
- We will only discuss integer math in this lecture.

Negation

- The negate instruction, `neg`, converts a number to its **two's complement.**
- `neg` sets the sign and zero flags
  - Which will be useful when we perform conditional jumps and moves.
- There is only a single operand which is source and destination

```
neg     rax    ; negate the value in rax
neg     eax    ; negate the value of eax and zx the rest
neg     ax     ; negate the value of ax
neg     al     ; negate the value of al
```

Negation

- For memory operands you must include a size prefix
- The sizes are `byte`, `word`, `dword` and `qword`

```
neg     qword [x] ; negate a 8 byte integer at x
neg     dword [x] ; negate a 4 byte integer at x
neg     word [x] ; negate a 2 byte integer at x
neg     byte [x]  ; negate a 1 byte integer at x
```

The `add` instruction

- The `add` instruction always has exactly 2 operands
  - The source and, (RHS)
  - the destination (LHS)
- It adds its source value to its destination
- The source can be a
  - immediate
  - register
  - memory location
- The destination can be a
  - register
  - memory location
- Using memory locations for both source and destination is **not allowed**
  - as is the general pattern with the x86-64 instruction set.

The add instruction

- After an ADD instruction executes it sets the following flags:
  - sign flag(SF)
  - zero flag(ZF)
  - overflow flag(OF)
  - there are more, but they are no important in this course.
- There is no special "signed add" versus "unsigned add" since the logic is identical
- There is a special 1 operand increment instruction, `inc`

```
inc rax      ; add one to rax
inc byte [x]; add one to the interger byte at x
```

A program using add

Program has three “variables”: **a**=151,**b**=310, and **sum**=0. We want to:

- set **a**=**a**+9
- set **sum**=**a**+**b**+10

A program using add

```
        segment .data
a       dq      151
b       dq      310
sum     dq      0
        segment .text
        global  main
main:
        mov     rax, 9       ; set rax to 9
        add     [a], rax     ; add rax to a
        mov     rax, [b]     ; get b into rax
        add     rax, 10      ; add 10 to rax
        add     rax, [a]     ; add the contents of a
        mov     [sum], rax   ; save the sum in sum
        xor     rax, rax
        ret
```

The subtract instruction

- The `sub` instruction performs integer subtraction
- Like `add` it supports 2 operands
- Only one of the operands can be a memory operand
- There is a "subtract one" instruction, `dec`
- It sets the sign flag, the zero flag and the overflow flag
- There is no special "signed subtract" versus "unsigned subtract" since the logic is identical

A program using `sub`

Program has three "variables": **a**=100,**b**=200, and **diff**=0. We want to:

- set **a**=**a**-10
- set **b**=**b**-10
- set **diff**=**b-a**

A program using sub

```
        segment .data
a       dq      100
b       dq      200
diff    dq      0
        segment .text
        global  main
main:
        mov     rax, 10
        sub     [a], rax    ; subtract 10 from a
        sub     [b], rax    ; subtract 10 from b
        mov     rax, [b]    ; move b into rax
        sub     rax, [a]    ; set rax to b-a
        mov     [diff], rax ; move the difference to diff
        mov     rax, 0
        ret
```

Multiplication

- Unsigned multiplication is done using the `mul` instruction
- Signed multiplication is done using `imul`
- There is only 1 form for `mul`
  - It uses 1 operand, the source operand
  - The other factor is in `rax`, `eax`, `ax` or `al`
  - The destination is ax for byte multiplies
  - Otherwise the product is in `rdx:rax`, `edx:eax`, or `dx:ax`

```
mov     rax, [a]
mul     qword [b]   ; a * b will be in rdx:rax
mov     eax, [c]
mul     dword [d]   ; c * d will be in edx:eax
```

Signed multiplication

- `imul` has a single operand form just like `mul`
- It also has a 2 operand form, source and destination, like `add` and `sub`
- Finally there is a 3 operand form: destination, source and immediate source
- If you need all 128 bits of product, use the single operand form

```
imul    rax, 100        ; multiply rax by 100
imul    r8, [x]         ; multiply r8 by x
imul    r9, r10         ; multiply r9 by r10
imul    r8, r9, 11      ; store r9 * 11 in r8
```

Division

- Division returns a quotient and a remainder
- It also has signed (`idiv`) and unsigned forms (`div`)
- In both forms the dividend is stored in `rdx:rax` or parts thereof
- The quotient is stored in `rax`
- The remainder is stored in `rdx`
- No flags are set

```
mov     rax, [x]        ; x will be the dividend
mov     rdx, 0          ; 0 out rdx, so rdx:rax == rax
idiv    qword [y]             ; divide by y
mov     [quot], rax     ; store the quotient
mov     [rem], rdx      ; store the remainder
```

Conditional move instructions

- There are many variants of conditional move, `cmovCC`, where `CC` is a condition like **l** to mean **less**
- These are great for simple conditionals
- You can avoid interrupting the instruction pipeline

Instruction	effect
cmovz	move if zero flag set
cmovnz	move if zero flag not set (not zero)
cmovl	move if result was negative
cmovle	move if result was negative or zero
cmovg	move if result was positive
cmovge	result was positive or zero

* The destination operand must be a register. The source operand can be a either a register or memory.

Conditional move examples

- Here is some code to compute absolute value of rax

```
mov     rbx, rax    ; save original value
neg     rax         ; negate rax
cmovl   rax, rbx    ; replace rax if negative
```

- The code below loads a number from memory, subtracts 100 and replaces the difference with 0 if the difference is negative

```
mov     rbx, 0      ; set rbx to 0
mov     rax, [x]    ; get x from memory
sub     rax, 100    ; subtract 100 from x
cmovl   rax, rbx    ; set rax to 0 if rax was negative
```

Why use a register?

- Don't use a register if a value is needed for 1 instruction
- Don't worry about it for things which execute infrequently
- Use registers instead of memory for instructions which execute enough to matter
- If you are writing a program for a class and efficiency is not part of the grade, pick the clearest way to write the code
- With so many registers, it can create opportunities for efficiency at the cost of clarity

Print to Console

```
mov  rax, 1          ; write
mov  rdi, 1           ; stdout
mov  rsi, output       ; address of first byte in output
mov  rdx, [length]    ; load length in rdx
syscall
```

Bit Operations

Bit usage

- A bit can mean one of a pair of characteristics
- True or false
- Male or female
- Bit fields can represent larger classes
  - There are 64 squares on a chess board, 6 bits could specify a position
  - The exponent field of a float can be represented using a number of bits.
  - We could use a 3 bit field to store a color from black, red, green, blue, yellow, cyan, purple and white

Bit operations

- Individual bits have values 0 and 1
- There are instructions to perform bit operations
- Using 1 as true and 0 as false
  - 1 and 1 = 1, or in C/C++, `1 && 1 = 1`
  - 1 and 0 = 0, or in C/C++, `1 && 0 = 0`
  - 1 or 0 = 1, or in C/C++, `1 || 0 = 1`
- We are interested in operations on more bits
  - `10101000b & 11110000b = 10100000b`
  - `10101000b | 00001010b = 10101010b`
- These are called "bit-wise" operations
- We will not use bit operations on single bits, though we will be able to test/set/reset individual bits

The Not operation

- C/C++ uses `!` for a logical not
- C/C++ uses `~` for a bit-wise not

```
!0 == 1
!1 == 0
~(false) == true
~(true) == false
~10101010b == 01010101b
~0xff00 == 0x00ff
!1000000 == 0  (non-zero integer is seen as true in c/c++)
~0== ?
~1== ?
```

The Not operation

- C/C++ uses `!` for a logical not
- C/C++ uses ˜ for a bit-wise not

```
!0 == 1
!1 == 0
~(false) == true
~(true) == false
~10101010b == 01010101b
~0xff00 == 0x00ff
!1000000 == 0  (non-zero integer is see as true in c/c++)
~0== -1
~1== -2
```

The Not instruction

- The `not` instruction flips all the bits of a number - one's complement
- The `not` operator does not affect any flags
- There is only a single operand which is the source and destination
- For memory operands you must include a size prefix
- The sizes are `byte`, `word`, `dword` and `qword`

```
not     rax         ; invert all bits of rax
not     dword [x]   ; invert double word at x
not     byte [x]    ; invert a byte at x
```

And operation

&	0	1
0	0	0
1	0	1

- C/C++ uses && for a logical and
- C/C++ uses & for a bit-wise and

```
11001100b & 00001111b == 00001100b
11001100b & 11110000b == 11000000b
0xabcdefab & 0xff == 0xab
0x0123456789abcdef & 0xff00ff00ff00ff00 == 0x010045008900cd00
```

- Bit-wise **and** is a bit selector

And instruction

- The **and** instruction performs a bit-wise **and**
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Both destination and source cannot be in memory
- The sign flag and zero flag are set (or cleared)

And Example

- We wish to extract bits 0-3 and store them in `rbx`

```
mov     rax, 0x12345678
mov     rbx, rax
and     rbx, 0xf            ; rbx has the low nibble 0x8
```

- We wish to extract bits 4-7 and store them in `rax`

```
mov     rdx, 0              ; prepare to divide
mov     rcx, 16             ; by 16
idiv    rcx                 ; rax has 0x1234567
and     rax, 0xf            ; rax has the nibble 0x7
```

Or operation

|	0	1
0	0	1
1	1	1

- C/C++ uses `||` for a logical or
- C/C++ uses `|` for a bit-wise or

```
11001100b | 00001111b == 11001111b
11001100b | 11110000b == 11111100b
0xabcdefab | 0xff == 0xabcdefff
0x0123456789abcdef | 0xff00ff00ff00ff00 == 0xff23ff67ffabffef
```

- **or** is a bit setter

Or instruction

- The **or** instruction performs a bit-wise **or**
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Both destination and source cannot be in memory
- The sign flag and zero flag are set (or cleared)

Or example

- Make a number odd

```
mov     rax, 0x1124
or      rax, 1          ; make the number odd
```

- Set bits 8-15.

```
mov     rax, 0x1000
or      rax, 0xff00     ; set bits 15-8
```

How would you make a number even?

Exclusive or operation

^	0	1
0	0	1
1	1	0

- C/C++ uses ^ for exclusive or

```
00010001b ^ 00000001b == 00010000b
01010101b ^ 11111111b == 10101010b
01110111b ^ 00001111b == 01111000b
0xaaaaaaaa ^ 0xffffffff == 0x55555555
0x12345678 ^ 0x12345678 == 0x00000000
```

- Exclusive or is a bit flipper

Exclusive or instruction

- The **xor** instruction performs a bit-wise **exclusive or**
- It has 2 operands, a destination and a source
- The source can be an immediate value, a memory location or a register
- The destination can be a register or memory
- Both destination and source cannot be in memory
- The sign flag and zero flag are set (or cleared)
- `mov rax, 0` uses 7 bytes
- `xor rax, rax` uses 3 bytes
- `xor eax, eax` uses 2 bytes

Exclusive or example

- Zero out a register.

```
mov     rax, 0x12345678
xor     eax, eax                ; set rax to 0
```

- Flip bits 0-3

```
mov     rax, 0x1234
xor     rax, 0xf                ; change to 0x123b
```

- Swap the value in two registers

```
xor rax, rbx
xor rbx, rax
xor rax, rbx
```

Shift operations

- C/C++ uses << for shift left and >> for shift right
- Shifting left introduces low order 0 bits
- Shifting right propagates the sign bit in C++ for signed integers
- Shifting right introduces 0 bits in C++ for unsigned integers
- Shifting left is like multiplying by a power of 2
- Shifting right is like dividing by a power of 2

```
101010b >> 3 == 101b
111111b << 2 == 11111100b
125 << 2 == 500   (125=>1111101<<2==111110100=>500)
0xabcd >> 4 == 0xabc
```

Shift instructions

- Shift left: `shl`
- Shift right: `shr`
- Shift arithmetic left: `sal`
- Shift arithmetic right: `sar`
- `shl` and `sal` are the same
- `shr` introduces 0 bits on the top end
- `sar` propagates the sign bit
- All the shifts use 2 operands
    - A destination register or memory
    - In immediate number of bits to shift
        - Or from old 16 bit asm the `cl` register can be used
- The sign and zero flags are set (or cleared)
- The carry flag is set to the last bit shifted out

Extracting a bit field

- There are at least 2 ways to extract a bit field
- Shift right followed by an **And** operation
  - To extract bits k to m (inclusive) with $m \geq k$, shift right k bits
  - And this value with a mask of $m - k + 1$ bits all set to 1

Extracting a bit field with shift/and

Need to extract bits 9–3

1	1	0	0	0	1	1	1	1	0	0	1	0	1	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift right 3 bits

0	0	0	1	1	0	0	0	1	1	1	1	0	0	1	0

And with 0x7f

0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0

Extracting a bit field

- The second way
- Shift left and then right
  - Shift left until bit m is the highest bit
  - With 64 bit registers, shift left $63 - m$ bits
  - Shift right to get original bit k in position 0
  - With 64 bit registers, shift right $63 - (m - k)$ bits

Extracting a bit field with shift/shift

Need to extract bits 9–3

1	1	0	0	0	1	1	1	1	0	0	1	0	1	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift left 6 bits

1	1	1	0	0	1	0	1	1	0	0	0	0	0	0	0

Shift right 9 bits

0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0

Rotate instructions

- The `ror` instruction rotates the bits of a register or memory location to the right
  - Values from the low end start filling in the top bits
- The `rol` instruction rotates left
  - Values from the top end of the value start filling in the low order bits
- These are 2 operand instructions like the shift instructions
- The first operand is the source to rotate (and the destination)
- The second operand is the number of bits to rotate
- The second operand is either an immediate value or `cl`
- Assuming 16 bit rotates

```
1 ror 2 = 0100000000000000b
0xabcd ror 4 = 0xdabc
0x4321 rol 4 = 0x3214
```

Filling a field

- There are at least 2 ways of filling in a field (with existing values)
- Use shifts and a mask.
  - Working with a 64 bit register, filling bits k to m (inclusive)
  - Prepare a mask of $m - k + 1$ bits all 1
  - Shift the new value and the mask left k bits
  - Negate the mask
  - And the old value and the mask
  - Or in the new value for the field

Filling a field 1

					We want to replace bits 6-3											
Original	1	1	0	0	0	1	1	1	1	0	0	1	0	1	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					with											
Value													1	1	0	1
	create mask of length 6-3+1=4															
Mask	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
	Shift both by k=3															
Value	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
Mask	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
	Negate the mask															
Mask	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1
	And with original															
Original	1	1	0	0	0	1	1	1	1	0	0	1	0	1	0	0
Mask	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1
	1	1	0	0	0	1	1	1	1	0	0	0	0	1	0	0
	or value with result															
	1	1	0	0	0	1	1	1	1	1	1	0	1	1	0	0

Filling a field

- Second method
- Use rotate and shift instructions and or in new value
    - Rotate the register right k bits
    - Shift the register right $m - k + 1$ bits
    - Shift the register left $m - k + 1$ bits
    - Or in the new value
    - Rotate the register left k bits

Filling a field 2

					We want to replace bits 6-3											
Original	1	1	0	0	0	1	1	1	1	0	0	1	0	1	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					with											
Value													1	1	0	1
	Rotate original right by k=3															
	1	0	0	1	1	0	0	0	1	1	1	1	0	0	1	0
	Shift right by m-k+1=6-3+1=4															
	0	0	0	0	1	0	0	1	1	0	0	0	1	1	1	1
	shift left by 4															
	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0
	or with value															
	1	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
	1	0	0	1	1	0	0	0	1	1	1	1	1	1	0	1
	Rotate left by k=3															
	1	1	0	0	0	1	1	1	1	1	1	0	1	1	0	0

Bit testing and setting

- It takes a few instructions to extract or set bit fields
- The same technique could be used to test or set single bits
- It can be more efficient to use special instructions operating on a single bit
    - The `bt` instruction tests a bit
        - the CF flag gets set to the value of the tested bit
        - we can gain access to the flag using `setc cl` (for example)
    - `bts` tests a bit and sets it
        - tested bit gets set to 1
    - `btr` tests a bit and resets it
        - tested bit gets set to 0
    - `btc` tests a bit and flips it
        - tested bit gets complemented
- These are all 2 operand instructions
- The first operand is a register or memory location
- The second is the bit to work on, either an immediate value or a register

Bit testing and setting example

- Checking if a number is odd

```
mov  rax, 101
bt   rax, 0
setc dl ;  1 will be stored in dl, i.e the number is odd
```

- Setting the 7th and 33rd bit of the qword A in memory to 1

```
bts qword [A],  7
bts qword [A],  33
```

Branching and Looping

Branching and looping

- So far we have only written "straight line" code
- Conditional moves gave us an avenue for trivial if like structures.
- But we really need
  - To handle code structures like if/else. So we need both conditional and unconditional branch statements
  - We need loops

Unconditional jump

- An unconditional jump is equivalent to a `goto`
- But jumps are necessary in assembly, while high level languages could exist without `goto`
- The unconditional jump looks like

  ```
  jmp label
  ```
- The `label` can be any label in the program's text segment
- We might think of parts of the text segment as functions
  - The computer will let you jump anywhere
  - You can try to jump to a label in the data segment, which hopefully will fail
- The assembler will generate an instruction register (`rip`) relative location to jump
  - The simplest form uses an 8 bit immediate: -128 to +127 bytes
  - The next version is 32 bits: plus or minus 2 GB
  - The short version takes up 2 bytes; the longer version 5 bytes
  - The assembler figures this out for you (Yay)

Unconditional jumps can vary

It is possible to use an unconditional jump to simulate a conditional jump.

- It is possible to jump to an address stored in a register.
- We can control the value of the register using a conditional move.

```
mov rax, a
mov rbx, b
cmovl rax, rbx ; rather jmp to b if the sign flag is set
jmp rax

a:
 .....
 .....
 jmp end
b:
 .....
 .....
end:
```

Unconditional jumps can vary

- Though it is simpler to just use a conditional jump.
- However you can construct an efficient switch statement by expanding this idea
    - You need an array of addresses and an index for the array to select which address to use for the jump

Unconditional jump used as a switch

```
        segment .data
switch: dq      case0
        dq      case1
        dq      case2
i:      dq      2
        segment .text
        global  main                ; tell linker about main
main:   mov     rax, [i]            ; move i to rax
        jmp     [switch+rax*8]      ; switch ( i )
case0:
        mov     rbx, 100            ; go here if i == 0
        jmp     end
case1:
        mov     rbx, 101            ; go here if i == 1
        jmp     end
case2:
        mov     rbx, 102            ; go here if i == 2
end:    xor     eax, eax
        ret
```

Conditional jump

- First you need to execute an instruction which sets some flags
- Then you can use a conditional jump
- The general pattern is
  `jCC label`
- The `CC` means a condition code

instruction	meaning	aliases	flags
jz	jump if zero	je	ZF=1
jnz	jump if not zero	jne	ZF=0
jg	jump if $>$ zero	jnle ja	ZF=0, SF=0
jge	jump if $\geq$ zero	jnl	SF=0
jl	jump if $<$ zero	jnge js	SF=1
jle	jump if $\leq$ zero	jng	ZF=1 or SF=1
jc	jump if carry	jb jnae	CF=1
jnc	jump if not carry	jae jnb	CF=0

Compare operation

- It can become cumbersome to always have to preform a calculation and store the result simply to use condition jump.
- This is where the compare operation comes in handy
  - `cmp`
- cmp takes 2 operand.
- cmp subtracts the second operand from the first and sets the appropriate flags.
- But, the result is not actually stored.
- At most one operand can be an immediate value.

Simple if statement

```
if ( a < b ) {
    temp = a;
    a = b;
    b = temp;
}
```

```
    mov    rax, [a]
    mov    rbx, [b]
    cmp    rax, rbx
    jge    in_order
    mov    [a], rbx
    mov    [b], rax
in_order:
```

If statement with an else clause

```
    if ( a < b ) {
        max = b;
    } else {
        max = a;
    }
```

```
        mov   rax, [a]
        mov   rbx, [b]
        cmp   rax, rbx
        jnl   else
        mov   [max], rbx
        jmp   endif
else:   mov   [max], rax
endif:
```

Looping with conditional jumps

- You can construct any form of loop using conditional jumps
- We will model our code after C's loops
- `while`, `do...while` and `for`
- We will also consider `break` and `continue`
- `break` and `continue` can be avoided in C, though sometimes the result is less clear
- The same consideration applies for assembly loops as well

Sum 1 to 1000

```
sum = 0;
i = 1;
while ( i <= 100 )
{
   sum +=i;
   i++;
}
```

Now the assembler version (no optimization done to keep things simple)

Sum 1 to 1000

```
    segment .data
sum dq 0
    segment .text
    global  _start
_start:
    mov rcx,1 ; i=1
while:
    cmp rcx,100
    jg ewhile
    add [sum],rcx
    inc rcx
    jmp while
ewhile:
```

Counting 1 bits in a quad-word

```
sum = 0;
i = 0;
while ( i < 64 )
{
    sum += data & 1;
    data = data >> 1;
    i++;
}
```

- There are much faster ways to do this
- But this is easy to understand and convert to assembly

Counting 1 bits in a quad-word in assembly

Assume we have the following data segment:

```
   segment .data
data dq 0xfedcba9876543210
sum  dq 0
```

Counting 1 bits in a quad-word in assembly

```
        segment .text
        global  main
main:   mov     rax, [data] ; rax holds the data
        xor     ebx, ebx    ; clear since setc will fill in bl
        xor     ecx, ecx    ; i = 0;
        xor     edx, edx    ; sum = 0;
while:  cmp     rcx, 64     ; while ( i < 64 ) {
        jnl     end_while   ; requires testing on opposite
        bt      rax, 0      ; data & 1
        setc    bl          ; move result of test to bl
        add     edx, ebx    ; sum += data & 1;
        shr     rax, 1      ; data = data >> 1;
        inc     rcx         ; i++;
        jmp     while       ; end of the while loop
end_while:
        mov     [sum], rdx  ; save result in memory
        xor     eax, eax    ; return 0 from main
        ret
```

Counting 1 bits in a quad-word in assembly

To be more true to the C-code. we could replace

```
bt      rax, 0
setc    bl
add     edx, ebx
```

with

```
mov     r8, rax
and     r8, 1
add     edx,r8d
```

Branching and Looping Continued

Do-while loops

- Strict translation of a while loop uses 2 jumps

```
while:
    some compare
    conditional jump to "ewhile"
    .....
    .....
    jump to "while"
ewhile:
```

- However, a do-while only requires one jump.

```
do_while:
    .....
    .....
    .....
    some compare
    conditional jump to "do_while"
```

Do-while loops

Any **while** loop can be simulated by a **do-while** loop wrapped in an **if** statement. For example

```
while ( condition )
{
   statements;
}
```

can be simulated as

```
if ( condition )
{
   do
   {
      statements;
   } while ( condition );
}
```

Ugly C code to search through a character array

```
//Looking for an character x other than 0. Store index in n
// data is a null terminated character array
    i = 0;
    c = data[i];
    if ( c != 0 )
    do
    {
        if ( c == x )
          break;
        i++;
        c = data[i];
    } while ( c != 0 );

    n = c == 0 ? -1 : i;
```

Assume we have the following data segment

Assume we have the following data segment

```
   section .data
data   db    "hello world",0
n      dq     0
x      db    'w'
```

Assembly code to search through an array

```
        mov     bl, [x]         ; value being sought
        mov     rcx, 0          ; i = 0;
        mov     al, [data+rcx]  ; c = data[i]
        cmp     al, 0           ; if ( c != 0 ) {
        jz      end_do_while    ; skip loop for empty string
do_while:
        cmp     al, bl          ; if ( c == x ) break;
        je      found
        inc     rcx             ; i++;
        mov     al, [data+rcx]  ; c = data[i];
        cmp     al, 0           ; while ( c != 0 );
        jnz     do_while
end_do_while:
        mov     rcx, -1         ; If we get here, we failed
found:  mov     [n], rcx        ; Assign either -1 or the
                                ; index where x was found
```

Assembly code to search through an array (Using only 64 bit registers)

```
        movzx   rbx, byte[x]          ;<----
        mov     rcx, 0
        movzx   rax, byte [data+rcx] ;<----
        cmp     rax, 0                ;<----
        jz      end_do_while
do_while:
        cmp     rax, rbx
        je      found
        inc     rcx
        movzx   rax, byte [data+rcx] ;<----
        cmp     rax, 0                ;<----
        jnz     do_while
end_do_while:
        mov     rcx, -1
found:  mov     [n], rcx
```

Counting loops

```
// assume we have 3 arrays of size n.
// Each containing longs (quad words)
```

```
    for ( i = 0; i < n; i++ )
    {
        c[i] = a[i] + b[i];
    }
```

Counting loops

```
//assume there are 3 contiguous segemnts in memory
// each containing n quad words.
   mov     rdx, [n]         ; use rdx for n
   xor     ecx, ecx         ; i = 0
for:
   cmp     rcx, rdx         ; i < n
   je      end_for          ; get out if equal
   mov     rax, [a+rcx*8]  ; get a[i]
   add     rax, [b+rcx*8]  ; a[i] + b[i]
   mov     [c+rcx*8], rax  ; c[i] = a[i] + b[i];
   inc     rcx              ; i++
   jmp     for
end_for:
```

Nested loops

Consider the double summation

$$\sum_{i=1}^{N} \sum_{j=1}^{i} j \tag{1}$$

ignoring the fact that

$$\sum_{i=1}^{N} \sum_{j=1}^{i} j = \frac{N(N+1)(N+2)}{6} \tag{2}$$

We will code this double sum in assembler.

Nested loops

Assuming we have:

```
segment .data
Sum:    dq 0
N:      dq 5
```

Nested loops

```
    mov rbx, [N]
    mov rax, 0          ; sum=0
    mov r8,1            ; i=1
loop1:
    cmp r8,rbx
    jg eloop1           ; !(i<=N)
    mov rcx,1           ; j=1
    loop2:
        cmp rcx, r8
        jg  eloop2      ; !(j<=i)
        add rax,rcx
        inc rcx
        jmp loop2
    eloop2:
    inc r8
    jmp loop1
eloop1:
   mov [Sum], rax
```

Loop instructions

- The CPU has instructions like
  - loop
  - loope
  - loopne

  which are designed for looping.
- They decrement `rcx` and do the branch if `rcx` is not 0
  - loope checks if zero flag is set as well.
  - loopne checks if the zero flag is not set as well
- It is faster to use `dec` and `jnz` instead
- The label must be within -128 to +127 bytes of `rip`
- Probably pointless on modern architecture. (it was fast on old architecture.)

Loop instructions

Add 5 to a sum 64 times.

```
   xor rax,rax; sum=0
   mov rcx, 64;
loop1:
   add rax, 5
   loop loop1
```

string (array)

- Let use first consider the simple array instruction
  `movsb`
- we must load the address of source data in `rsi`
- we must load the address of destination data in `rdi`
- on execution `movsb` will move the value at rsi to rdi, and increment both addresses by 1.

```
....
mes1: db "abcdefg"
mes2: db "1234567"
....
   mov rsi, mes1
   mov rdi, mes2
   movsb
```

mes2 will equal "a234567" and rsi=mes1+1 and rdi=mes2+1

Repeat string (array) instructions

But how is this useful?

- we utilize the string operation with the `rep` instruction.
- rep will repeatedly call the string operation until `rcx`$= 0$.

For example, let us copy an array of 1000 bytes.

```
lea rsi, [source]
lea rdi, [destination]
mov rcx, 1000
rep movsb
```

- `lea`?
- lea rsi, [source] = mov rsi, source

Repeat string (array) instructions

- What is if the array contains non bytes?
- We simple use the different size specifier
  - `movsw`
  - `movsd`
  - `movsq`
- Now instead of incrementing the addresses by 1,
- We will increase it by the size.
- e.g. with movsq we increment by 8

Repeat string (array) instructions

- Up to now we have relied on an incrementing the source and destination addresses.
- actually the address is only increased if the direction flag(DF)=0 (default)
- If DF=1, the addresses will decrement after each string instruction
- We can set the direction flag to 1 with

```
std
```

- or 0 with

```
cld
```

Store instruction

- The `stosb` instruction stores the byte in `al` at the address specified in `rdi` and increments `rdi`
- If the direction flag is set it decrements `rdi`
- There are also `stosw`, `stosd` and `stosq` to operate 2, 4 and 8 byte quantities

```
mov     eax, 1
mov     ecx, 1000000
lea     rdi, [destination]
rep     stosd       ; place 1000000 1's in destination
```

Scan instruction and Compare instruction

Scan: (`scas`)

- There are a collection of scan string instructions which scan data from the address pointed at by `rdi` and increment (or decrement) `rdi`
- They compare data against `al`, `ax`, `eax`, ...
- Used with `repe`, and will stop once data found or `rcx`=0

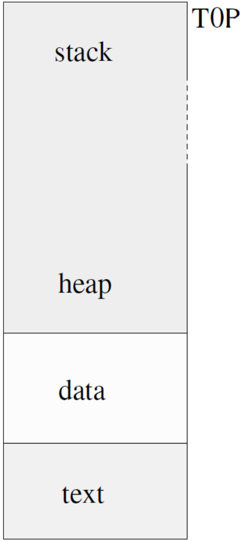
Compare: (`cmps`)

- The compare string instructions compare the data pointed at by `rdi` and `rsi`
- End once `rcx` has reached zero of a match is found.
- Used with `repe`, and will stop once match found or `rcx`=0

The Stack and Functions

The Stack

- Up until now we have mostly ignored the stack.
- Instead we have relied on the data/bss segment.
- This has been sufficient so far, but this will change.
- We are now in a position where our programs will start being complex enough that we cannot rely entirely on fixed size allocations.



The stack in a Familiar Context

- Consider the following(intentionally inefficient)recursive C++ function. Assuming we don't know that

$$fib(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}} \tag{1}$$

```
// assuming fib(1)=1
int fib(int n)
{
   if(n<2)
      return 1;
   int fibM1=fib(n-1);
   int fibM2=fib(n-2);
   return fibM1+fibM2;
}
```

- If N is too large our program will experience a stack overflow.

The Stack in a Familiar Context

- But why?

The Stack in a Familiar Context

- But why?
- There are two causes (though in a sense they are the same)
  - The more obvious one is each execution of the *fib* will push at least two integers onto the stack. One for each of `FibM1`, and `FibM2`. Now the space complexity of this naive fibincci algorithm is $O(n)$. We can try and roughly calculate $fib(10\,000\,000)$ we need $2 * 4\text{bytes}*10\,000\,000 =$ 76.29mb just to store the max required number of `FibM1`s, and `FibM2`s.
    - Most Linux distros have a stack between $8mb$ and $16mb$

The Stack in a Familiar Context

- But why?
- There are two causes (though in a sense they are the same)
  - The more obvious one is each execution of the *fib* will push at least two integers onto the stack. One for each of `FibM1`, and `FibM2`. Now the space complexity of this naive fibincci algorithm is $O(n)$. We can try and roughly calculate $fib(10\,000\,000)$ we need $2 * 4\text{bytes}*10\,000\,000 = 76.29\text{mb}$ just to store the max required number of `FibM1s`, and `FibM2s`.
    - ★ Most Linux distros have a stack between $8mb$ and $16mb$
- What if we implement our recursive function to somehow not push any local variable data onto the stack?
  - Even if this was possible there is one stack item we would have to push.

The Stack in a Familiar Context

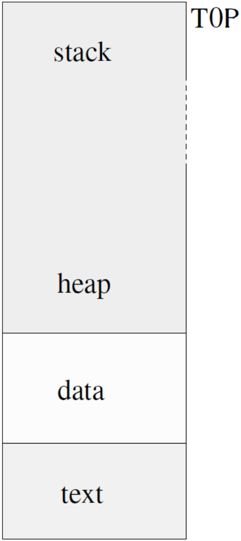
- But why?
- There are two causes (though in a sense they are the same)
  - The more obvious one is each execution of the *fib* will push at least two integers onto the stack. One for each of `FibM1`, and `FibM2`. Now the space complexity of this naive fibincci algorithm is $O(n)$. We can try and roughly calculate $fib(10\,000\,000)$ we need $2 * 4\text{bytes}*10\,000\,000 = 76.29\text{mb}$ just to store the max required number of `FibM1s`, and `FibM2s`.
    - Most Linux distros have a stack between $8mb$ and $16mb$
- What if we implement our recursive function to somehow not push any local variable data onto the stack?
  - Even if this was possible there is one stack item we would have to push.
  - The return address.

The Stack

- Now that we have seen how it can break, lets us use the stack.
- The first point to recall is that the stack starts at :0x7fffffffffff. Unless there is some stack randomization in place.
- The second point is that the *rsp* register stores the stack pointer.
- If nothing has been added to the stack *rsp* =0x7fffffffffff.
- We can interact with the stack in a number of ways, simplest of which are the `push` and `pop` instructions.

The Stack

- The `push` instruction decrements the `rsp` register and stores the value being pushed at this address
- The `pop` instruction places the value at the top of the stack into its operand and increments `rsp`
- With the x86-64 instructions you should push and pop 8 bytes at a time
  - It is also possible to push and pop 2 bytes (word) at a time
  - Direct 4 byte push and pops is not enabled in 64-bit mode



A Stack Example

Example with start of the stack at $0x7ffffffff$

```
;A
mov  rax,74
push rax
;B
inc  rax
push rax
;C
inc  rax
push rax
;D
pop rax
;E
pop rax
;F
```

0x7ffffffffff0	x	74	74	74	74	74
0x7fffffffffe8	x	x	75	75	75	x
0x7fffffffffe0	x	x	x	76	x	x
.	x	x	x	x	x	x
.	x	x	x	x	x	x
.	x	x	x	x	x	x
	A	B	C	D	E	F

In GDB

- You can use x/1dg $rsp to test the stack content.
- You can use p/x $rsp to test the stack pointer itself.
- Just remember that stack randomization effects the initial stack pointer.

The Stack

- Stack space is often reserved for local variables by subtracting the size needed from the stack pointer (`rsp`).
- Then an offset is used to refer to the variables.

```
sub rsp,      16  ;subtract 16 bytes
mov qword [rsp+8], 123 ;set our first qword variable to 12
mov qword [rsp],   24  ;set our second qword variable to 2
```

The stack

0x7fffffffffff				
0x7ffffffffff0	x		123	123
0x7fffffffffe8	x			24
0x7fffffffffe0	x	x	x	x
.	x	x	x	x
.	x	x	x	x
.	x	x	x	x
	x	x	x	x
		sub rsp, 16	mov [rsp+8],123	mov [rsp], 24

The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.

```
first   equ 8
second  equ 0
mov qword [rsp+first],123
mov qword [rsp+second],24
```

The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.

```
first   equ 8
second  equ 0
mov qword [rsp+first],123
mov qword [rsp+second],24
```

- How do we delete the variables after use?

The Stack

- Remembering offsets for a large number of variables can become a burden. Instead use the equ pseudo-op.

```
first   equ 8
second  equ 0
mov qword [rsp+first],123
mov qword [rsp+second],24
```

- How do we delete the variables after use?
  - We just move the stack pointer back.

    ```
    add rsp,16
    ```

Functions

- Assuming we have loaded our parameters (will be explained shortly)
- You can call a function using

```
call    my_function
```

- my_function should be an appropriate address/label in the code segment
- The function's return value will be in rax or xmm0
- The effect of a function call is much like

```
    push    next_instruction
    jmp     my_function
next_instruction:
```

The Return Instruction

- You can return to the location a function was called from using

```
ret
```

- The effect of the return instruction (`ret`) is to pop an address off the stack and branch to it
- We could get much the same effect using

```
pop   rdi
jmp   rdi
```

Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack

Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.

Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.
- Linux and Mac OS/X pass integer and address parameters 1 through 6 in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
  - The remaining integer and address parameters are pushed onto the stack
  - The first 6 floating point parameters are passed in registers `xmm0` - `xmm5`

Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.
- Linux and Mac OS/X pass integer and address parameters 1 through 6 in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
  - The remaining integer and address parameters are pushed onto the stack
  - The first 6 floating point parameters are passed in registers `xmm0` - `xmm5`
- Windows uses registers `rcx`, `rdx`, `r8` and `r9` for the first 4 integer and address parameters
  - The remaining integer and address parameters are pushed onto the stack. (there is also a set amount of stack padding)
  - Windows uses `xmm0` - `xmm3`

Function Parameters

- How do we get our parameters to our function?
- On 32 bit Linux all parameters were pushed onto the stack
- On x86-64 there are 8 more registers, so some parameters are passed in registers.
- Linux and Mac OS/X pass integer and address parameters 1 through 6 in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`
    - The remaining integer and address parameters are pushed onto the stack
    - The first 6 floating point parameters are passed in registers `xmm0` - `xmm5`
- Windows uses registers `rcx`, `rdx`, `r8` and `r9` for the first 4 integer and address parameters
    - The remaining integer and address parameters are pushed onto the stack. (there is also a set amount of stack padding)
    - Windows uses `xmm0` - `xmm3`
- In all cases pushed parameters are pushed in reverse order

Function Parameters (2)

- Functions like `printf` having a variable number of parameters must place the number of floating point parameters in `rax`

Register	Usage	Preserved across function calls
`rax`	temporary register; with variable arguments passes information about the number of vector registers used; 1^{st} return register	No
`rbx`	callee-saved register; optionally used as base pointer	Yes
`rcx`	used to pass 4^{th} integer argument to functions	No
`rdx`	used to pass 3^{rd} argument to functions; 2^{nd} return register	No
`rsp`	stack pointer	Yes
`rbp`	callee-saved register; optionally used as frame pointer	Yes
`rsi`	used to pass 2^{nd} argument to functions	No
`rdi`	used to pass 1^{st} argument to functions	No
`r8`	used to pass 5^{th} argument to functions	No
`r9`	used to pass 6^{th} argument to functions	No
`r10`	temporary register, used for passing a function's static chain pointer	No
`r11`	temporary register	No
`r12-r15`	callee-saved registers	Yes

Simple Function

Simple function that returns the larger of two longs.

```
; long max(long a, long b)
max:
   mov rax, rdi    ; move parm1 to rax
   cmp rax, rsi    ; compare rax to parm2
   cmovl rax, rsi ; if parm2 > rax then move parm 2 to rax
   ret
```

Simple Function

Calling the simple function

```
mov rdi, 123 ; load parm1
mov rsi, 742 ; load parm2
call max
```

Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.

Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.
  - The reason behind this requirement is to make it possible for local variables (on the stack) to be on 16 byte boundaries,
  - a requirement for some SSE(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.

Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.
  - The reason behind this requirement is to make it possible for local variables (on the stack) to be on 16 byte boundaries,
  - a requirement for some SSE(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.
- Conforming functions generally start with "`push rbp`" which re-establishes the 16 byte bounding temporarily **botched by the function call**
  - Remember, the `call` operation pushes a 8 byte value onto the stack (the return address)

Real World Functions

- Both Linux and Windows require the maintenance of the stack on 16 byte boundaries during the main part of functions.
  - The reason behind this requirement is to make it possible for local variables (on the stack) to be on 16 byte boundaries,
  - a requirement for some SSE(Streaming SIMD Extensions) and AVX(Advanced Vector Extension) instructions.
- Conforming functions generally start with "`push rbp`" which re-establishes the 16 byte bounding temporarily **botched by the function call**
  - Remember, the `call` operation pushes a 8 byte value onto the stack (the return address)
- Following that, conforming functions subtract multiples of 16 from `rsp` to allocate stack space or push pairs of 8 byte values
  - Even if this means over allocation

Hello world, at last

```
        section .data
msg:    db      "Hello World!",0x0a,0

        section .text
        global  main
        extern  printf
main:
        push    rbp
        mov     rbp, rsp    ; will explain shortly
        mov     rdi, msg    ; parameter 1 for printf
        mov     rax, 0      ; 0 floating point parameters
        call    printf
        mov     rax, 0      ; return 0
        mov     rsp, rbp    ; will explain shortly (NIB)
        pop     rbp
        ret
```

Stack frames

- Stack frames are used by the `gdb` debugger to trace backwards through the stack to inspect calls made in a process
- If we start and end each function like:

```
push    rbp
mov     rbp, rsp

...
mov rsp, rbp
pop rbp
ret
```

- We are in effect constructing a link list of all of the stack frames.
- All non-leaf functions must have the stack frame set up and destruction to conform to the ABI and be properly c/c++ compatible.

Stack frames

	In base function:	In Function L1:	In Function L2:	In Function L3:	In Function L4:
rbp	0x0	StackPointer(0)	StackPointer(1)	StackPointer(2)	StackPointer(3)
rsp	StackPointer(0)	StackPointer(1)	StackPointer(2)	StackPointer(3)	StackPointer(4)
0x7ffffffff0	x	Ret Adr to Base	Ret Adr to Base	Ret Adr to Base	Ret Adr to Base
0x7fffffffe8	x	0x0	0x0	0x0	0x0
0x7fffffffe0	x	x	Ret Adr to L1	Ret Adr to L1	Ret Adr to L1
0x7fffffffd8	x	x	StackPointer(0)	StackPointer(0)	StackPointer(0)
0x7fffffffd0	x	x	x	Ret Adr to L2	Ret Adr to L2
0x7fffffffc8	x	x	x	StackPointer(1)	StackPointer(1)
0x7fffffffc0	x	x	x	x	Ret Adr to L3
0x7fffffffb8	x	x	x	x	StackPointer(2)
0x7fffffffb0	x	x	x	x	x

*assuming no local varibles are stored on the stack. If the local variables were stored where would they be?

Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.

Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.

Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.
    - We only **have** to subtract 8 bytes.
    - But we **should** maintain the 16 byte boundary.

```
push    rbp
mov     rbp, rsp
sub     rsp,16
```

Stack frames

- If you require space for local variables you simple need to subtract an amount from the stack pointer `rsp`.
  - The subtraction should always maintain the 16 byte boundary.
  - For example, say we wish to have a single quadword as a local variable.
    - We only **have** to subtract 8 bytes.
    - But we **should** maintain the 16 byte boundary.

```
push    rbp
mov     rbp, rsp
sub     rsp,16
```

- if we allocate local variables we **must** use

```
mov rsp, rbp
pop rbp
ret
```

Just popping will not work.

Stack frames

- If you prefer you can utilize

  ```
  leave
  ret
  ```

  instead of

  ```
  mov rsp, rbp
  pop rbp
  ret
  ```

print max example

```
main:
   push rbp
   mov  rbp, rsp
;   print_max ( 100, 200 );
   mov  rdi, 100    ; first parameter
   mov  rsi, 200    ; second parameter
   call print_max
   mov  rax, 0     ; to return 0
   leave
   ret
```

print max example

```
; void print_max ( long a, long b )
; {
a   equ  0
b   equ  8
max equ 16
print_max:
   push rbp
   mov  rbp, rsp
   sub  rsp, 32         ; leave space for a, b and max
   mov  [rsp+a], rdi    ; save a
   mov  [rsp+b], rsi    ; save b
   mov  [rsp+max], rdi ; max = a;
   cmp  rsi, rdi        ; if ( b > max ) max = b
   jng  skip
   mov  [rsp+max], rsi
```

print max example

```
skip:
segment .data
   fmt db   "max(%ld,%ld) = %ld",0xa,0
segment .text
   mov  rdi, fmt        ; address of format string
   mov  rsi, [rsp+a]    ; first  %ld
   mov  rdx, [rsp+b]    ; second %ld
   mov  rcx, [rsp+max] ; third  %ld
   mov  rax, 0          ; zero floating point param
   call printf
   leave
   ret
```

Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?

Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.

Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.
- Solution?

Accessing Local Variables

- Can you see the potential issue with the way we are accessing our local variables?
  - If anything is pushed onto the stack our offsets become incorrect.
- Solution?
  - Simply use `rbp` as the base address. Just remember that since rbp holds the value of rsp before the subtraction you need to use subtract from rbp and not add (think about what the offset values should be).
  - or just don't use push and pop other than for the stack frame setup and destruction

Recursive Functions

- Recursive algorithms serve as a good example for why we need stack based storage.
- Often times we can get away with utilizing registers that are preserved across function calls.
- However consider the case where we a have a recursive algorithm. On the first level we decide to use `r15` to store a value. But now on the second level `r15` is already in use...

Recursive Functions

- Consider again the following recursive Fibonacci function

```
// assuming fib(1)=1
long fib(long n)
{
   if(n<2)
      return 1;
   return fib(n-1)+fib(n-2);
}
```

Recursive Functions

- ```
  fib:  push rbp
        mov rbp, rsp
        sub rsp, 16
  N   equ 0
  nM1 equ 8
        mov rax, 1          ;base case return value
        cmp rdi, 2
        jl .end             ;first parameter<2 (base case)
        dec rdi             ;recall rdi is the first parameter (n)
        mov [rsp+N], rdi    ;save N-1
        call fib
        mov [rsp+nM1], rax
        mov rdi, [rsp+N]    ;load N-1
        dec rdi
        call fib
        add rax, [rsp+nM1]
        .end
        leave
        ret
  ```
  ```

Function Implementation: Correct Practice

- If a function is a **non-leaf** function you **must** set up and destroy a stack frame.
- If you utilize a register in your function that should be preserved across function calls (like `r15`) you **must** restore it to its original value.
- Example

```
sub rsp, 16
mov [rsp], r15
mov [rsp+8], r14
....
....
mov r14, [rsp+8]
mov r15, [rsp]
add rsp, 16
```

Function Implementation: Correct Practice

- or if you make use of no other stack based memory.

```
push r15
push r14
....
....
pop r14
pop r15
```

Floating Point Instructions

Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.

Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.
- x86-64 CPUs have 16 floating point registers (128 or 256 bits)
  - These registers can be used for single data instructions or single instruction multiple data instructions (SIMD)

Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.
- x86-64 CPUs have 16 floating point registers (128 or 256 bits)
  - These registers can be used for single data instructions or single instruction multiple data instructions (SIMD)
- We will focus on these newer registers
  - The older instructions tended to start with the letter "`f`" and referenced the stack using register names like `ST0`
  - The newer instructions reference using registers with names like "`xmm0`", and "`ymm0`". (`zmm0 in new CPUs as well`)

Floating point instructions

- There are 16 floating point registers.
- `ymm0` to `ymm15` (AVX registers)
- Each one is 256 bits.
- The lower half (128 bits) of `ymm-` is refereed to at `xmm-`
- `xmm0` to `xmm15` (SSE registers)
- The full 256 bit register are available from the Core i series.

Floating point instructions

- There are 16 floating point registers.
- `ymm0` to `ymm15` (AVX registers)
- Each one is 256 bits.
- The lower half (128 bits) of `ymm-` is refereed to at `xmm-`
- `xmm0` to `xmm15` (SSE registers)
- The full 256 bit register are available from the Core i series.
- We will mainly discuss `xmm` registers, all operations are the same for `ymm` registers, we just append a `v` in front of the instruction.

Moving scalars to or from floating point registers

Moving floating point numbers

- The two instructions available are `movss` and `movsd`
- `movss` moves a single 32 bit floating point value to or from an `xmm` register (float/single)
- `movsd` moves a single 64 bit floating point value (double)

Moving scalars to or from floating point registers

Moving floating point numbers

- The two instructions available are `movss` and `movsd`
- `movss` moves a single 32 bit floating point value to or from an `xmm` register (float/single)
- `movsd` moves a single 64 bit floating point value (double)
- It should be noted that there is no implicit data conversion - unlike the old instructions which converted floating point data to an 80 bit internal format

Moving scalars to or from floating point registers

- The instructions follow the standard pattern of having at most one memory address

```
    segment .data
x:    dd 12.35 ; float/single
y:    dq 14.36 ; double

    movss    xmm0, [x]    ; move the float value at x into xmm0
    movsd    [y], xmm1    ; move double value from xmm1 to y
    movss    xmm2, xmm0   ; move from xmm0 to xmm2
```

Moving packed data

- The XMM registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The YMM registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)

Moving packed data

- The `XMM` registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The `YMM` registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)
- But how do we load them?
  - There are two types of packed move instructions available.
  - An **aligned** version and an **unaligned** version.
  - **aligned** move requires the data to be of a 16 byte boundary, but is faster in general.
  - **unaligned** move is slower, though more so on older CPUs.

Moving packed data

- The `XMM` registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The `YMM` registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)
- But how do we load them?
  - There are two types of packed move instructions available.
  - An **aligned** version and an **unaligned** version.
  - **aligned** move requires the data to be of a 16 byte boundary, but is faster in general.
  - **unaligned** move is slower, though more so on older CPUs.
- If you try an use a aligned move on unaligned data you will get a segmentation fault.

Moving packed data

Actual packed move instructions:

- `movaps` moves 4 floats to/from a memory address aligned at a 16 byte boundary
- `movups` does the same task with unaligned memory addresses
- `movapd` moves 2 doubles to/from a memory address aligned at a 16 byte boundary
- `movupd` does the same task with unaligned memory addresses

```
   segment .data
x: dd 12.3, 9.3, 123.2, 0.1
a: dq 0, 0
....
   movups  xmm0, [x]    ; move 4 floats to xmm0
   movupd  [a], xmm15   ; move 2 doubles to a
```

Moving packed data

If you wish to use the aligned move:

```
   segment .data
align 16
x: dd 12.3, 9.3, 123.2, 0.1
....
   movaps  xmm0, [x]    ; move 4 floats to xmm0
```

Floating point addition

- `addss` adds a scalar float (single precision) to another
- `addsd` adds a scalar double to another
- `addps` adds 4 floats to 4 floats - pairwise addition
- `addpd` adds 2 doubles to 2 doubles
- There are 2 operands: destination and source
- The source can be memory or an `XMM` register
- The destination must be an `XMM` register
- Flags are unaffected

```
movss   xmm0, [a]   ; load a
addss   xmm0, [b]   ; add b to a
movss   [c], xmm0   ; store sum in c
```

And

```
movapd  xmm0, [a]   ; load 2 doubles from a
addpd   xmm0, [b]   ; add a[0]+b[0] and a[1]+b[1]
movapd  [c], xmm0   ; store 2 sums in c
```

Floating point subtraction

- subss subtracts the source float from the destination
- subsd subtracts the source double from the destination
- subps subtracts 4 floats from 4 floats
- subpd subtracts 2 doubles from 2 doubles

```
movss   xmm0, [a]   ; load a
subss   xmm0, [b]   ; add b from a
movss   [c], xmm0   ; store a-b in c
```

And

```
movapd  xmm0, [a]   ; load 2 doubles from a
subpd   xmm0, [b]   ; add a[0]-b[0] and a[1]-b[1]
movapd  [c], xmm0   ; store 2 differences in c
```

Basic floating point instructions

instruction	effect
`addsd`	add scalar double
`addss`	add scalar float
`addpd`	add packed double
`addps`	add packed float
`subsd`	subtract scalar double
`subss`	subtract scalar float
`subpd`	subtract packed double
`subps`	subtract packed float
`mulsd`	multiply scalar double
`mulss`	multiply scalar float
`mulpd`	multiply packed double
`mulps`	multiply packed float
`divsd`	divide scalar double
`divss`	divide scalar float
`divpd`	divide packed double
`divps`	divide packed float

Conversion to a different length floating point

- `cvtss2sd` converts a scalar single (float) to a scalar double
- `cvtps2pd` converts 2 packed floats to 2 packed doubles
- `cvtsd2ss` converts a scalar double to a scalar float
- `cvtpd2ps` converts 2 packed doubles to 2 packed floats

```
cvtss2sd    xmm0, [a]  ; get a into xmm0 as a double
addsd       xmm0, [b]  ; add a double to a
cvtsd2ss    xmm0, xmm0 ; convert to float
movss       [c], xmm0
```

Converting floating point to/from integer

- `cvtss2si` converts a float to a double word or quad word integer by rounding
- `cvtsd2si` converts a float to a double word or quad word integer by rounding
- `cvttss2si` and `cvttsd2si` convert by truncation
- `cvtsi2ss` converts an integer to a float in an `XMM` register
- `cvtsi2sd` converts an integer to a double in an `XMM` register
- When converting integers from memory a size qualifier is needed

```
cvtss2si    eax, xmm0   ; convert to dword integer
cvtss2si    rax, xmm0   ; convert to qword integer
cvtsi2sd    xmm0, rax   ; convert qword to double
cvtsi2sd    xmm0, dword [x] ; convert dword integer
```

Unordered versus ordered comparisons

- In the IEEE-754 floating point standard there are two types of NaNs (not a number)
- QNaN or SNaN
  - QNaN means "quiet, not a number"
  - SNaN means "signalling, not a number"
  - Both have all exponent field bits set to 1
  - QNaN has its top fraction bit equal to 1

Unordered versus ordered comparisons

- Floating point comparisons can cause exceptions
- Ordered comparisons cause exceptions on QNaN or SNaN
- An unordered comparison causes exceptions only for SNaN
- gcc uses unordered comparisons
- If it's good enough for gcc, it's good enough for us
- ucomiss compares floats
- ucomisd compares doubles
- The first operand **must** be an XMM register
- They set the zero flag, parity flag and carry flags

```
movss   xmm0, [a]
mulss   xmm0, [b]
ucomiss xmm0, [c]
jbe  less_eq   ; jmp if a*b <= c
```

Conditional floating point jumps

instruction	meaning	aliases	flags
`jb`	jump if below	`jc jnae`	CF=1
`jbe`	jump if below or equal	`jna`	ZF=1 or CF=1
`ja`	jump if above	`jnbe`	ZF=0 or CF=0
`jae`	jump if above or equal	`jnc jnb`	CF=0
`je`	jump if equal	`jz`	ZF=1
`jne`	jump if not equal	`jnz`	ZF=0

c= carry flag set

z= zero flag set

Mathematical functions

- 8087 had sine, cosine, arctangent and more
- The newer instructions omit these operations on `XMM` registers
- Instead you are supposed to use efficient library functions
- There are instructions for
  - Minimum
  - Maximum
  - Rounding
  - Square root
  - Reciprocal of square root

Minimum and maximum

- `minss` and `maxss` compute minimum or maximum of scalar floats
- `minsd` and `maxsd` compute minimum or maximum of scalar doubles
- The destination operand must be an `XMM` register
- The source can be an `XMM` register or memory
- `minps` and `maxps` compute minimum or maximum of packed floats
- `minpd` and `maxpd` compute minimum or maximum of packed doubles
- `minps xmm0, xmm1` computes 4 minimums and places them in `xmm0`

```
movss   xmm0, [x]    ; move x into xmm0
maxss   xmm0, [y]    ; xmm0 has max(x,y)
movapd  xmm0, [a]    ; move a[0] and a[1] into xmm0
minpd   xmm0, [b]    ; xmm0[0] has min(a[0],b[0])
                     ; xmm0[1] has min(a[1],b[1])
```

Rounding

- `roundss` rounds 1 float
- `roundps` rounds 4 floats
- `roundsd` rounds 1 double
- `roundpd` rounds 2 doubles
- The first operand is an `XMM` destination register
- The second is the source in an `XMM` register or memory
- The third operand is a rounding mode

mode	meaning
0	round, giving ties to even numbers
1	round down
2	round up
3	round toward 0 (truncate)

Square roots

- `sqrtss` computes 1 float square root
- `sqrtps` computes 4 float square roots
- `sqrtsd` computes 1 double square root
- `sqrtpd` computes 2 double square roots
- The first operand is an `XMM` destination register
- The second is the source in an `XMM` register or memory

Distance in 3D

$$d = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2)}$$

```
distance3d:
    movss   xmm0, [rdi]      ; x from first point
    subss   xmm0, [rsi]      ; subtract x from second point
    mulss   xmm0, xmm0       ; (x1-x2)^2
    movss   xmm1, [rdi+4]    ; y from first point
    subss   xmm1, [rsi+4]    ; subtract y from second point
    mulss   xmm1, xmm1       ; (y1-y2)^2
    movss   xmm2, [rdi+8]    ; z from first point
    subss   xmm2, [rsi+8]    ; subtract z from second point
    mulss   xmm2, xmm2       ; (z1-z2)^2
    addss   xmm0, xmm1       ; add x and y parts
    addss   xmm0, xmm2       ; add z part
    sqrtss  xmm0, xmm0
    ret
```

Dot product in 3D

$$d = x_1 x_2 + y_1 y_2 + z_1 z_2$$

```
dot_product:
        movss   xmm0, [rdi]
        mulss   xmm0, [rsi]
        movss   xmm1, [rdi+4]
        mulss   xmm1, [rsi+4]
        addss   xmm0, xmm1
        movss   xmm2, [rdi+8]
        mulss   xmm2, [rsi+8]
        addss   xmm0, xmm2
        ret
```

Arrays

Arrays

- An array is a **contiguous** collection of memory cells of a specific type
- The start address of an array is the address of the first element
  - The start address is associated with the label given before a data definition in the data segment or a data reservation in the bss segment.
  - Unless the array is in allocated memory.
- The first index of an array in C/C++ and assembly is 0
- Some high level languages use different or user-selectable starting indices for arrays
  - Fortran defaults to 1

Array address computation

- Array elements all have the same size: 1, 2, 4 and 8 are common
  - If I have a pointer in my struct. The heap allocated memory does not explicitly contribute to the size of the object.
  - The pointer itself contributes to the size of the object not the heap allocated memory
  - A pointer on 64bit linux. Is 64 bits.
- Suppose an array has elements of size 4 and starts at address `0x10000`
  - The first element (at index 0) is at `0x10000`
  - The second element (at index 1) is at `0x10004`
  - The third element (at index 2) is at `0x10008`
  - Element number `k` is at address `0x10000 + k*4`

General pattern for memory references

`[label]`	the value contained at label
`[label+ind]`	the value contained at the memory address obtained by adding the label and index register
`[label+2*ind]`	the value contained at the memory address obtained by adding the label and index register times 2
`[label+4*ind]`	the value contained at the memory address obtained by adding the label and index register times 4
`[label+8*ind]`	the value contained at the memory address obtained by adding the label and index register times 8

General pattern for memory references

Consider:

```
    segment .data
c: dq  4,1,5,2,7,8
```

Then,

```
mov rax, [c];
```

moves 4 into `rax`. And

```
mov rcx,2
mov rax, [c+8*rcx];
```

moves 5 into `rax`.
What would the following misguided move load?

```
mov rcx,1
mov rax, [c+4*rcx];
```

General pattern for memory references

`[reg]`	the value contained at the memory address in the register
`[reg+k*ind]`	the value contained at the memory address obtained by adding the register and index register times k
`[label+reg+k*ind]`	the value contained at the memory address obtained by adding the label, the register and index register times k
`[n+reg+k*ind]`	the value contained at the memory address obtained by adding n, the register and index register times k

General pattern for memory references

Consider:

```
    segment .data
a: dq  123
c: dq  4,1,5,2,7,8
```

Then

```
lea rcx,[c]    ; or mov rcx, c
mov rax,[rcx]
```

will load 4 into `rax`. And

```
lea rcx,[a]
mov rdi,4
mov rax,[8 +rcx + 8*rdi]
```

General pattern for memory references

Consider:

```
    segment .data
a: dq  123
c: dq  4,1,5,2,7,8
```

Then

```
lea rcx,[c]    ; or mov rcx, c
mov rax,[rcx]
```

will load 4 into rax. And

```
lea rcx,[a]
mov rdi,4
mov rax,[8 +rcx + 8*rdi]
```

- will load 7 into rax.

Memory references

- For items in the data and bss segments we can use a label
- For arrays passed into functions the address is passed in a register
- Soon we will be allocating memory using `malloc`
    - This address will typically be stored in memory
    - Later to use the data, we must load the address from memory into a register
    - Then we can use a register form of memory reference
- The use of a number or a label is equivalent to the computer
    - Both use the same instruction and place the number or label value into the same field of the instruction
    - Using multipliers of 2, 4 or 8 are essentially "free" with index registers

Copy dword array example

- In the function below the first parameter is the address of the first dword of a destination array (rdi)
- The second parameter is the address of the source array (rsi)
- The third parameter is the number of dwords to copy (rdx)
- It would generally be faster to use "rep movsd"

```
copy_array:
        xor     ecx, ecx            ; index=0
more:   mov     eax, [rsi+4*rcx]    ; move src[index] to temp
        mov     [rdi+4*rcx], eax    ; move to dst[index]
        inc     rcx                 ; ++index
        cmp     rcx, rdx
        jne     more
        xor     eax, eax
        ret
        ; if rdx=0 bad things happen
```

Allocating arrays

If we wish to directly allocate heap storage in assembler we have two options.

- We can make use of the `brk` and `sbrk` system calls which allow us a means of altering the heap boundary.
- Or the more modern approach using `mmap`.

In this course we will however make use of the C `malloc` function.

- If malloc is not fast enough, your time would be better served rewriting a version of `malloc` for your purposes (maybe in ASM) rather than using the system calls directly all the time.
- A nice guide can be found at `https://moss.cs.iit.edu/cs351/slides/slides-malloc.pdf` (this is only if you are interested)

Allocating arrays

- We will allocate arrays using the C `malloc` function

```
void *malloc ( long size );
```

- The parameter to `malloc` is the number of bytes to allocate
- `malloc` returns the address of the array or 0
- Data allocated should be freed

```
    void free ( void *ptr );
```

Code to allocate an array

- The code below allocates an array of 1 billion bytes
- It saves the pointer to the new array in memory location named `pointer`

```
extern  malloc
...
mov     rdi, 1000000000
call    malloc
mov     [pointer], rax
```

Advantages for using allocated arrays

- The array will be the right size
- There are size limits of about 2 GB in the data and bss segments
- The assembler phase is very slow with large arrays and the program is large
- Assembling a program with a 2 GB array in the data segment took about 100 seconds
- The executable was over 2 GB
- Using `malloc` the program assembles in less than 1 second and the executable as about 10 KB

Processing arrays

- We present an application which creates an array
- Fills the array with random data by calling `random`
- Prints the array if the size is small
- Determines the minimum value in the array.
- Only the helper functions will be discussed in the lecture.

Creating an array

- This function allocates an array of **double words**
- The number of **double words** is the only parameter
- Note the use of a stack frame to avoid any problems of stack misalignment

```
;       array = create ( size );
create:
        push    rbp
        mov     rbp, rsp
        imul    rdi, 4
        call    malloc
        leave
        ret
```

Filling the array with random numbers

```
fill: ; void fill(int* array,long size) \\ assumes size>=1
.array  equ     0
.size   equ     8
.i      equ     16
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     [rsp+.array], rdi
        mov     [rsp+.size], rsi
        mov     rcx, 0
.more   mov     [rsp+.i], rcx
        call    rand ;rand returns an integer (int=32 bits)
        mov     rcx, [rsp+.i]
        mov     rdi, [rsp+.array]
        mov     [rdi+rcx*4], eax
        inc     rcx
        cmp     rcx, [rsp+.size]
        jl      .more
        leave
        ret
```

Printing the array

```
;       void print (int* array, long size);
print:
.array  equ     0
.size   equ     8
.i      equ     16
push    rbp
mov     rbp, rsp
sub     rsp, 32
mov     [rsp+.array], rdi
mov     [rsp+.size], rsi
mov     rcx, 0
mov     [rsp+.i], rcx
```

Printing the array

```
    segment .data
.format:
    db      "%10d",0x0a,0
    segment .text
.more
    lea     rdi, [.format]
    mov     rdx, [rsp+.array]
    mov     rcx, [rsp+.i]
    mov     esi, [rdx+rcx*4]
    mov     rax, 0
    call    printf
    mov     rcx, [rsp+.i]
    inc     rcx
    mov     [rsp+.i], rcx
    cmp     rcx, [rsp+.size]
    jl      .more
    leave
    ret
```

Finding the minimum value in the array

- This function calls no other function
- There is no need for a stack frame
  - but there is no real harm in having one)
- A conditional move is faster than branching

```
;       x = min ( a, size ); int min(int* array, long size)
;       assumes size>=1
min:
        mov     eax, [rdi]          ; start with a[0]
        mov     rcx, 1
.more   mov     r8d, [rdi+rcx*4]    ; get a[i]
        cmp     r8d, eax
        cmovl   eax, r8d            ; move if smaller
        inc     rcx
        cmp     rcx, rsi
        jl      .more
        ret
```

Command line parameter array

- The first argument to `main` is the number of command line parameters
- The second argument is the address of an array of character pointers, each pointing to one of the parameters
- Below is a C program illustrating the use of command line parameters

```
#include <stdio.h>

int main ( int argc, char *argv[] )
{
    int i;
    for ( i = 0; i < argc; i++ ) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Assembly program listing command line parameters

```
        segment .data
format  db      "%s",0x0a,0
        segment .text
        global  main        ; let the linker know about main
        extern  printf      ; resolve printf from libc
main:   push    rbp         ; prepare stack frame for main
        mov     rbp, rsp
        sub     rsp, 16
        mov     rcx, rsi    ; move argv to rcx
        mov     rsi, [rcx]  ; get first argv string
start_loop:
        lea     rdi, [format]
        mov     [rsp], rcx  ; save argv
        call    printf
        mov     rcx, [rsp]  ; restore argv
        add     rcx, 8      ; advance to next pointer in argv
        mov     rsi, [rcx]  ; get next argv string
        cmp     rsi, 0
        jnz     start_loop  ; end with NULL pointer
end_loop:
```

System Calls

System calls

- A system call is how a program requests a service from an operating system's kernel.
- A user process cannot do privileged instructions
  - No direct access to a hard drive
  - No changing of CPU mapping registers
- Instead a user process makes a system call
- The system call is a part of the kernel of the operating system
  - It verifies that the user should be allowed to do the requested action and then does the action.

32 bit Linux system calls

- Each system call is identified by an integer defined in "`/usr/include/asm/unistd_32.h`" (varies slightly by distro)
- The system call number is placed in `eax`
- Parameters are placed in registers `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp`
- Processes use the software interrupt number 0x80 to make the system call
- Return value in `eax`

```
        segment .data
hello:  db      "Hello world!",0x0a
        segment .text
        ...
        mov     eax, 4          ; syscall 4 is write
        mov     ebx, 1          ; file descriptor
        lea     ecx, [hello]    ; array to write
        mov     edx, 13         ; write 13 bytes
        int     0x80
```

64 bit Linux system calls

- System call number defined in "/usr/include/asm/unistd_64.h"(varies slightly by distro)
- System call number is placed in rax
- Parameters rdi, rsi, rdx, r10, r8 and r9.
- Process uses syscall instruction
- Return value in rax.

```
        segment .data
hello:  db      "Hello world!",0x0a
        segment .text
        global  _start
_start: mov     eax, 1          ; syscall 1 is write
        mov     edi, 1          ; file descriptor
        lea     rsi, [hello]    ; array to write
        mov     edx, 13         ; write 13 bytes
        syscall
        mov     eax, 60         ; syscall 60 is exit
        xor     edi, edi        ; exit(0)
        syscall
```

C wrapper functions

- Every system call is available through a C “wrapper function”
- A wrapper function might do very little other than shuffle registers
- Some wrappers offer a little extra convenience
- Wrapper functions are described in section 2 of the on-line manual
  - Use “`man 2 write`” to learn about the write system call

```
        segment .data
msg:    db      "Hello World!",0x0a ; String to print
len:    equ     $-msg               ; Length of the string
        segment .text
        global  main
        extern  write, exit
main:
        mov     edi, 1              ; Arg 1 is the fd
        lea     rsi, [msg]          ; Arg 2 is the array
        mov     edx, len            ; Arg 3 is the length
        call    write
        xor     edi, edi            ; 0 return = success
        call    exit
```

$-msg

```
segment .data
msg:    db      "Hello World!",0x0a ; String to print
len:    equ     $-msg               ; Length of the string
```

- Think of the $ as the current assembly address (byte addressable).
- Then $-msg, is the current assembly address minus the address of the label msg.
- As a result len is set to equal to the number of bytes in the message.
- Better then manually counting.

Open system call

```
int open ( char *pathname, int flags [, int mode ] );
```

- `pathname` is a null-terminated string
- `flags` is a collection of options or'ed together about how the file is opened.

flags	meaning
0	read-only
1	write-only
2	read and write
0x40	create if needed
0x200	truncate to size 0
0x400	append

- `mode` is the permissions to grant if a file is created

Permissions for files

- There are 3 basic permissions: read, write and execute
- There are 3 categories of users: user (owner), group and other
- Each of the 3 categories gets a 0 or 1 for each basic permission
- Octal works well for permissions
- For example 640o
    - 110 100 000
    - rw- r-- ---
    - read and write permission to the user.
    - read permission to the group.
    - no permission to others.

Code to open a file

- Open system call returns a small non-negative integer identifying the opened file (file descriptor)
- It returns -1 on error and sets errno

```
        segment .data
fd:     dd      0
name:   db      "sample",0
        segment .text
        extern  open
        lea     rdi, [name] ; pathname
        mov     esi, 0x42   ; read-write | create
        mov     edx, 600o   ; read-write for me
        call    open
        cmp     eax, 0
        jl      error       ; failed to open
        mov     [fd], eax
```

Read and write system calls

```
int read ( int fd, void *data, long count );
int write ( int fd, void *data, long count );
```

- `fd` is the file descriptor returned by `open`
- `data` is a pointer to some memory to send or receive data
- `count` is the number of bytes to read or write
- The data can be any type
- These functions return the number of bytes read or written
- `read` returns 0 on end-of-file
- They both return -1 on errors and set `errno`
- Use `perror` to print a text description based on `errno`

Lseek system call

```
long lseek ( int fd, long offset, int whence );
```

- `offset` is a byte offset
- If `whence` is 0, `offset` is the byte position from the start of the file
- If `whence` is 1, `offset` is relative to the current position
- If `whence` is 2, `offset` is relative to the end of the file
- `lseek` returns the current position
- Using `whence` = 2 and `offset` = 0, `lseek` returns the file size

The close system call

```
int close ( int fd );
```

- You should make a habit of closing files when no longer needed
- They will be closed when the process ends
- No data is buffered in the user process, so data written to unclosed files will not have been lost on a close call.
- Closing will reduce overhead in the kernel
- There is a per-process limit on open files
- Use "`ulimit -a`" to see your limits

Example

- Let use write a simple encrypt program.
- The program will rotate each byte right by 1
- We will use the following helper functions.
  - long file_size(int fd)
  - void encrypt(void* data, long size)

file_size

```
;long file_size(int fd)
file_size:
   push rbp
   mov rbp, rsp
   sub rsp, 16
.fd    equ 0
.size  equ 4
   mov [rsp+.fd],edi
   mov rsi,0              ;offset=0
   mov rdx,2              ;whence=2
   call lseek
   mov [rsp+.size],rax
   mov edi, [rsp+.fd]
   mov rsi,0              ;offset=0
   mov rdx,0              ;whence=0
   call lseek
   mov rax,[rsp+.size]
   leave
   ret
```

encrypt

```
;void encrypt(void* data, long size)
encrypt:
   xor rcx,rcx        ; i=0
.while:
   cmp rcx,rsi        ;i<size
   jge .ewhile
   mov al, [rdi+rcx]  ;al=data[i]
   ror al, 1
   mov [rdi+rcx],al   ;data[i]=al
   inc rcx            ;++i
   jmp .while
.ewhile
  ret
```

main program

```
   segment .data
filename:   db "test.txt",0
fd:         dd 0
fsize:      dq 0
datap:      dq 0

   segment .text
   global  main
   extern open,lseek,malloc,read,write

main:
   push rbp
   mov rbp, rsp
```

main program

```
mov rdi, filename
mov rsi, 2 ;read-write
call  open

mov [fd], eax ; save fd
mov edi, eax
call file_size

mov [fsize],rax
mov rdi, rax
call malloc
mov [datap],rax
```

main program

```
mov edi,[fd]
mov rsi,[datap]
mov rdx,[fsize]
call read

mov edi,[fd]
mov rsi,0
mov rdx,0
call lseek

mov rdi, [datap]
mov rsi, [fsize]
call encrypt
mov edi,[fd]
mov rsi,[datap]
mov rdx,[fsize]
call write
leave
ret
```

Structs

Structs

- A struct is a compound object

```
struct Customer {
    int  id;
    char name[71];
    char address[71];
    int  balance;
};
```

- How big is this structure?
- There is the easy, but wrong answer
- and the somewhat more complex answer

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.
- An `int` is 4 bytes. We have 2 of these in our structure, so that's 8 bytes

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.
- An `int` is 4 bytes. We have 2 of these in our structure, so that's 8 bytes
- A `char` is 1 byte and we have 2 arrays of 71 characters each, so this is another 142 bytes

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.
- An `int` is 4 bytes. We have 2 of these in our structure, so that's 8 bytes
- A `char` is 1 byte and we have 2 arrays of 71 characters each, so this is another 142 bytes
- So 150 in total.

Structs

If we want to allocate space for our struct, a simple call to `malloc` will suffice.

```
mov     rdi, 150          ; size of a Customer
call    malloc
mov     [c], rax          ; save the address
```

So c now holds a pointer to our allocated structure.

- But how do we use the struct?

Structs

If we want to allocate space for our struct, a simple call to `malloc` will suffice.

```
mov     rdi, 150          ; size of a Customer
call    malloc
mov     [c], rax          ; save the address
```

So c now holds a pointer to our allocated structure.

- But how do we use the struct?
- By using offsets.

```
struct Customer {
   int  id;                      [rax]------->id
   char name[71];                [rax+4]----->name
   char address[71];             [rax+75]---->address
   int  balance;                 [rax+146]--->balance
};
```

Filling in a C struct

```
char * strcpy ( char * destination, const char * source );
```

```
   segment .data
name    db "Bob",0
address db "22 Duncun street",0
balance dd 123
....................
        mov     [rax], dword 7  ; set the id
        lea     rdi, [rax+4]    ; name field
        lea     rsi, [name]     ; name to copy to struct
        call    strcpy
        mov     rax, [c]
        lea     rdi, [rax+75]   ; address field
        lea     rsi, [address]  ; address to copy
        call    strcpy
        mov     rax, [c]
        mov     edx, [balance]
        mov     [rax+146], edx
```

Assembly struct

- Using the yasm struc pseudo-op we can define a Customer

```
        struc   Customer
id      resd    1
name    resb    71
address resb    71
balance resd    1
        endstruc
```

- id, name, address and balance are globals
- It's almost the same as doing 4 equates
- The size is Customer_size

Assembly struct

- Using the yasm struc pseudo-op we can define a Customer

```
        struc   Customer
id      resd    1
name    resb    71
address resb    71
balance resd    1
        endstruc
```

- id, name, address and balance are globals
- It's almost the same as doing 4 equates
- The size is Customer_size
- But, you could not have id in 2 structs

Assembly struct

- One alternative is to prefix field names with dots

```
         struc    Customer
.id      resd     1
.name    resb     71
.address resb     71
.balance resd     1
         endstruc
```

- Then you would have to use `Customer.id`
- Another alternative is to use an abbreviated prefix

```
        struc    Customer
c_id      resd     1
c_name    resb     71
c_address resb     71
c_balance resd     1
        endstruc
```

Program to allocate and fill a struct - data segment

```
        segment .data
name    db      "Calvin", 0
address db      "12 Mockingbird Lane",0
balance dd      12500
        struc   Customer
c_id      resd    1
c_name    resb    71
c_address resb    71
c_balance resd    1
        endstruc
c       dq      0        ; to hold a Customer pointer
```

Program to allocate and fill a struct - part of text segment

```
mov     rdi, Customer_size
call    malloc
mov     [c], rax     ; save the pointer
mov     [rax+c_id], dword 7
lea     rdi, [rax+c_name]
lea     rsi, [name]
call    strcpy
mov     rax, [c]     ; restore the pointer
lea     rdi, [rax+c_address]
lea     rsi, [address]
call    strcpy
mov     rax, [c]     ; restore the pointer
mov     edx, [balance]
mov     [rax+c_balance], edx
```

Size Discrepancy

- Now the hard question. How big would the same C++ struct be?

Size Discrepancy

- Now the hard question. How big would the same C++ struct be?
- 152 **bytes**

C Alignment, and Struct padding

- This happens because C/C++ enforces primitives to have specific alignment based on its size.
- In effect C/C++ is padding the struct to achieve this alignment.

C Alignment, and Struct padding

- This happens because C/C++ enforces primitives to have specific alignment based on its size.
- In effect C/C++ is padding the struct to achieve this alignment.
- Certain data types have specific alignment requirements.
- The ones relevant to us in 64-bit linux are:
  - chars(1 byte) have no alignment requirement.
  - shorts(2 bytes) must start on an even address (multiple of 2).
  - int,float(4 bytes) must start on an multiple of 4
  - long,double(8 bytes) must start on a multiple of 8
  - pointer must start on a multiple of 8
- Furthermore alignment must still be preserved across struct elements in an array.

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char *p;      // 8 bytes
   char  c;      // 1 byte
   int   x;      // 4 bytes
}
```

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char *p;      // 8 bytes
   char  c;      // 1 byte
   int   x;      // 4 bytes
}
```

- Will actually be stored as

```
struct example
{
   char *p;      // 8 bytes
   char  c;      // 1 byte
   char  pad[3]; // 3 bytes
   int   x;      // 4 bytes
}
```

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char  c;      // 1 byte
   char *p;      // 8 bytes
}
```

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char  c;      // 1 byte
   char *p;      // 8 bytes
}
```

- Will actually be stored as

```
struct example
{
   char  c;      // 1 byte
   char  pad[7];// 7 bytes
   char *p;      // 8 bytes
}
```

C Alignment, and Struct padding

- The padding also has to be applicable for aligning multiples of the same struct (arrays)

```
struct example
{
   int  e;  //4 bytes
   char c;  //1 byte
}
```

C Alignment, and Struct padding

- The padding also has to be applicable for aligning multiples of the same struct (arrays)

```
struct example
{
   int  e;  //4 bytes
   char c;  //1 byte
}
```

- Will actually be stored as

```
struct example
{
   int  e;       //4 bytes
   char c;       //1 byte
   char pad[3];  //3 bytes
}
```

C Alignment, and Struct padding

- ```
  struct example
  {
     long  e;  //8 bytes
     char  c;  //1 byte
  }
  ```
  ```

C Alignment, and Struct padding

- ```
  struct example
  {
     long  e;  //8 bytes
     char  c;  //1 byte
  }
  ```
- Will actually be stored as

  ```
  struct example
  {
     long  e;        //8 bytes
     char c;         //1 byte
     char pad[7];    //7 bytes
  }
  ```
  ```

C Alignment, and Struct padding

- ```
  struct example
  {
     int a;  //4 bytes
     long b; //8 bytes
     char c; //1 byte
  }
  ```
  ```

C Alignment, and Struct padding

- ```
  struct example
  {
      int a;  //4 bytes
      long b; //8 bytes
      char c; //1 byte
  }
  ```
- Will actually be stored as

  ```
  struct example
  {
      int a;       //4 bytes
      char pad[4]; //4 bytes
      long b;      //8 bytes
      char c;      //1 byte
      char pad2[7];//7 bytes
  }
  ```

  Why 7 and not 3?
  ```

Allocating a slightly more complex array of customers

```
            segment .data
            struc   Customer
c_id        resd    1      ; 4 bytes in total
c_name      resb    65     ; 69 bytes in total
c_address   resb    65     ; 134 bytes in total
            align   4      ; aligns to 136
c_balance   resd    1      ; 140 bytes in total
c_rank      resb    1      ; 141 bytes in total
            align   4      ; aligns to 144
            endstruc
customers   dq      0
            segment .text
            mov   rdi, 100 ; for 100 structs
            imul   rdi, Customer_size
            call  malloc
            mov   [customers], rax
```

Printing an array of customers

```
        segment .data
format  db    "%s %s %d",0x0a,0
        segment .text
        push rbp
        mov rbp, rsp
        push  r15
        push  r14
        mov   r15, 100        ; counter saved through calls
        mov   r14, [customers]; pointer saved through calls
more    lea   rdi, [format]
        lea   rsi, [r14+c_name]
        lea   rdx, [r14+c_address]
        mov   ecx, [r14+c_balance]
        mov   rax, 0
        call  printf
        add   r14, Customer_size
        dec   r15
        jnz   more
        pop   r14
        pop   r15
        leave
```

Data Structures

Data structures

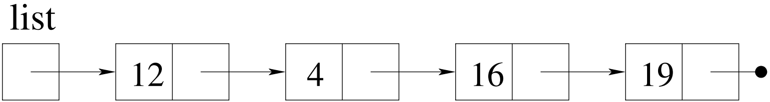
- In this lecture we will not learn any *new* assembler instructions/concepts.
- But rather, we will utilize the knowledge we have gained so far to build some common data structures.
  - One obvious downside to using ASM to implement a data structure, is that generality via templatization is not possible. (unless you write part of a complier for your own tweaked version of assembler)
  - However given ASM's low level nature we can utilize the specific details of the intended type to our advantage. Namely in terms of
    - Speed
    - Size

Outline

The data structures we will be focusing on are as follows

- Linked Lists
- Double Link Lists
- Hash Tables
- Binary and K-Aray trees
- Directed Graphs

Linked lists



- A simple linked list is constructed of a sequence of structs
- Each struct has some data and a pointer to the next item on the list
- The filled circle means a pointer equal to NULL (0)
- There needs to be some memory cell containing the first pointer
- This list has no obvious order to the keys
- It could be ordered by insertion time in two ways: by inserting at the front or the end
- It is easier to insert at the front, though the value of `list` will change with each insertion

List node struct definition

```
        struc   node
n_value resq    1
n_next  resq    1
        endstruc
```

Creating an empty list

- The only requirement will be to set the pointer to NULL
- Having a function makes it possible to change later with less impact on the rest of the program

```
newlist:
        mov rax, 0
        ret
        ...
        call    newlist
        mov     [list], rax ; where list is a dq
```

Inserting a number into a list

- A new `node` will be allocated and placed at the start
- We must pass the list pointer into the function
- We also must receive a new pointer back to store in `list`
- In C++ the prototype would be node* insert(node*, long) and we would use

```
list = insert ( list, k );
```

- In assembly we would insert k using

```
mov     rdi, [list] ; pass in the list pointer
mov     rsi, [k]
call    insert
mov     [list], rax ; we have a new list pointer
```

Insert code

```
insert:
.list   equ     0
.k      equ     8
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.list], rdi ; save list pointer
        mov     [rsp+.k], rsi    ; and k on stack
        mov     rdi, node_size
        call    malloc           ; rax will be node pointer
        mov     r8, [rsp+.list]  ; get list pointer
        mov     r9, [rsp+.k]     ; get k
        mov     [rax+n_next], r8 ; new node point to old list
        mov     [rax+n_value], r9; save k in node
        leave
        ret
```

Printing the list

```
 ;print(node* list)
print:
   push rbp
   mov rbp, rsp
   push r12
   mov r12,rdi ; save list
               ; we use r12 since since it
               ; is safe across function calls
   segment .data
.prt_frm: db "%ld ",0
.prt_nl:  db 0x0a,0
```

Printing the list cont.

```
segment .text
   .while:
      cmp r12,0
      je .ewhile
      mov rdi, print.prt_frm
      mov rsi, [r12+n_value]
      xor eax,eax
      call printf
      mov r12,[r12+n_next]
      jmp .while
   .ewhile:

   mov rdi, print.prt_nl
   xor eax,eax
   call printf

   pop r12
   leave
   ret
```

Main program to build a list, part 1

```
main:
.list equ 0
.k    equ 8
segment .data
.scanf_fmt: db "%ld",0
segment .text
```

Main program to build a list, part 2

```
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        call    newlist
        mov     [rsp+.list], rax  ; .list equal to 0
.more   lea     rdi, [.scanf_fmt] ; .scanf_fmt - "%ld",0
        lea     rsi, [rsp+.k]     ; .k equal to 8
        xor     eax, eax          ; no floating point value parameters
        call    scanf
        cmp     rax, 1            ; quit if scanf does not return 1
        jne     .done
        mov     rdi, [rsp+.list]  ; Get the list pointer
        mov     rsi, [rsp+.k]     ; Get k
        call    insert
        mov     [rsp+.list], rax  ; Save new list pointer
        mov     rdi, rax          ; Move the pointer to be a parameter
        call    print
        jmp     .more             ; Try to read another number
.done   leave
        ret
```

Example

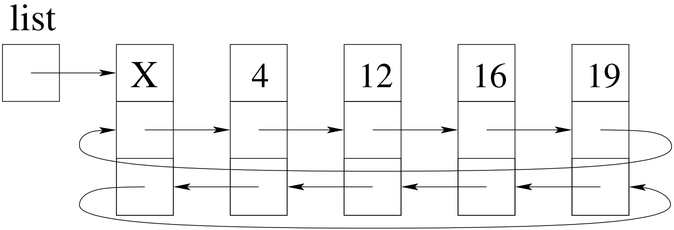
```
4
4
5
5 4
7
7 5 4
8
8 7 5 4
1
1 8 7 5 4
2
2 1 8 7 5 4
6
6 2 1 8 7 5 4
4
4 6 2 1 8 7 5 4
1
1 4 6 2 1 8 7 5 4
2
2 1 4 6 2 1 8 7 5 4
f
```

Ordered Insert

Homework:

- Implement a deallocate method for the list.
- Implement an insert at the end of the list.
- Implement an ordered insert, using insert as a helper function (where we assume the list is ordered before the call).

Doubly linked lists

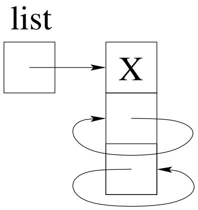


- This list uses forwards and backwards pointers to make a cycle
- Also the first node is not used, so an empty list will have one node and will be circular
- The first node is called a "head" node
- Using a head node and a circular list makes insertion trivial
- You can also insert and remove from either end easily

Doubly linked list `node` struct

```
        struc   node
n_value resq    1
n_next  resq    1
n_prev  resq    1
        endstruc
```

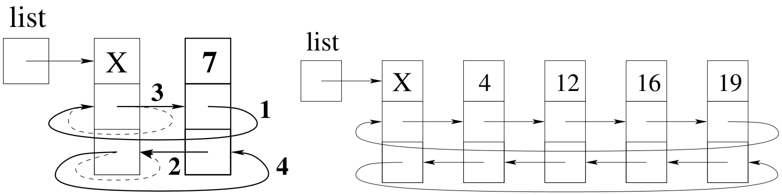
- An “empty” list is still circular
- There are no special cases to consider



Doubly linked list node :new list

```
; list=newlist()
newlist:
    push rbp
    mov rbp, rsp
    mov rdi, node_size
    call malloc
    mov [rax+n_next], rax
    mov [rax+n_prev], rax
    leave
    ret
```

Inserting at the front of a doubly linked list



- Make the new node's n_next point to the head cell's next
- Make the new node's n_prev point to the head cell
- Make the head's next cell point to the new cell
- Make the new node's next node's prev cell point backward to the new node

The original links are dashed lines

Insertion function

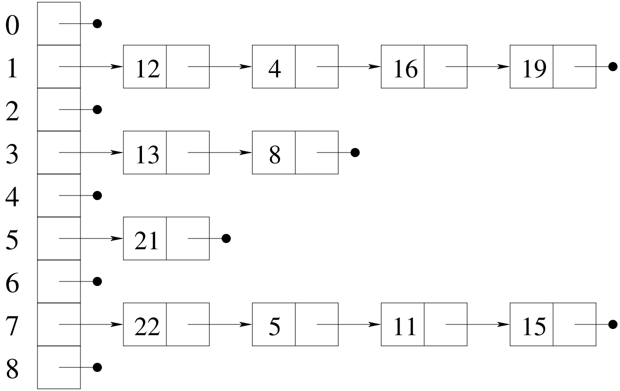
```
;        insert ( node* list, long k );
insert: push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.list], rdi   ; save list pointer, .list equ 0
        mov     [rsp+.k], rsi      ; and k on stack, .k equ 8
        mov     rdi, node_size
        call    malloc             ; rax will be node pointer
        mov     r8, [rsp+.list]    ; get list pointer
        mov     r9, [r8+n_next]    ; get head's next
        mov     [rax+n_next], r9   ; set new node's next
        mov     [rax+n_prev], r8   ; set new node's prev
        mov     [r8+n_next], rax   ; set head's next
        mov     [r9+n_prev], rax   ; set new node's next's prev
        mov     r9, [rsp+.k]       ; get k
        mov     [rax+n_value], r9 ; save k in node
        leave
        ret
```

List traversal

```
;         print ( node* list );
print:    push      rbp
          mov       rbp, rsp
          sub       rsp, 16
          mov       [rsp+.r12], r12    ; save r12, .r12 equ 0
          mov       [rsp+.list], rdi   ; save list, .list equ 8
          mov       r12, [rdi+n_next] ; skip the head node
          cmp       r12, [rsp+.list]   ; is the list empty? (head.next==head)
          je        .done
.more     lea       rdi, [.print_fmt] ; .print_fmt: db "%ld ",0
          mov       rsi, [r12+n_value]
          xor       eax,eax
          call      printf             ; print the node's value
          mov       r12, [r12+n_next] ; advance to the next node
          cmp       r12, [rsp+.list]   ; have we reached the head cell?
          jne       .more
.done     lea       rdi, [.newline]    ; .newline:  db 0x0a,0
          xor       eax,eax
          call      printf
          mov       r12, [rsp+.r12]    ; restore r12
          leave
          ret
```

Hash tables

- For each key, compute a hash value. $A[hash(key)]$
- The hash value defines an index in an array to store a value associated with a given key.
- Collisions occur when 2 different keys hash to the same index
- The simplest collision resolution is to use a linked list



A hash function for integers

- A good hash function spreads the keys around
- Using $(k \mod t)$ as a hash, were t is the table size, is sufficient for our purposes
- A good recommendation is to make t prime, but
- in this example, $t = 256$, so using **and** works

```
;       i = hash ( n );
hash    mov     rax, rdi
        and     rax, 0xff
        ret
```

- For example $-4 \rightarrow 111..100$ & $0xff = 0..011111100 \rightarrow 252$

A hash function for strings

- The code below uses the characters of the string as coefficients of a polynomial
- The polynomial is evaluated at 191 (a prime)
- Then a mod is done with 100000 to get the hash value
- Homework: write this in assembler. We use Horner's Method to calculate the polynomial quicker.

```
int hash ( unsigned char *s )
{
    unsigned long h = 0;
    int i = 0;
    while ( s[i] ) {
        h = h*191 + s[i];
        i++;
    }
    return h % 100000;
}
```

Hash node structure and array of pointers

- Our hash table that uses integer keys has only 256 entries/pointers
- Usually the array would be larger and a creation function needed

```
        segment .data
table   times 256 dq    0   ; All NULL pointers
        struc  node
n_value resq    1
n_next  resq    1           ; Singly linked list
        endstruc
```

Function to find a key

```
;       p = find ( n ); where p is a pointer to the matched node.
;       p = 0 if not found
find:   push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp], rdi              ; save key
        call    hash
        mov     rax, [table+rax*8]      ; get pointer
        mov     rdi, [rsp]              ; get key
        cmp     rax, 0                  ; empty list?
        je      .done
.more   cmp     rdi, [rax+n_value]      ; key match?
        je      .done
        mov     rax, [rax+n_next]       ; advance on the collision list
        cmp     rax, 0                  ; end of list
        jne     .more
.done   leave
        ret
```

Function to insert a key

```
insert: push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.n], rdi       ; save n, .n equ 0
        call    find
        cmp     rax, 0              ; Is n already there?
        jne     .found
        mov     rdi, [rsp+.n]       ; compute hash(n)
        call    hash
        mov     [rsp+.h], rax       ; save hash value, .h equ 8
        mov     rdi, node_size      ; allocate a node
        call    malloc
        mov     r9, [rsp+.h]        ; use r9 as index register
        mov     r8, [table+r9*8]    ; get old pointer from table
        mov     [rax+n_next], r8    ; make new node point to old
        mov     r8, [rsp+.n]        ; get n from the stack
        mov     [rax+n_value], r8   ; set the node value
        mov     [table+r9*8], rax   ; make new node first on its list
.found  leave
        ret
```

Binary trees

- A binary tree is a hierarchy of nodes
- There is a root node (or not, for an empty tree)
- Each node can have a left child and a right child
- The node structure has 2 pointers
- Either or both pointers could be NULL
- Binary trees usually impose an ordering on the nodes.
  - We will assume keys less than the current node's key will be stored down the left branch
  - $Leftchild < Current < Rightchild$
- Such a tree is a "binary search tree"
- We do not store duplicate keys.

Node Structure

```
    struc node
n_value resq    1
n_left  resq    1
n_right resq    1
    endstruc
```

A structure for the tree

- We could represent an empty tree as a NULL pointer
- This introduces special cases
  - specifically any operation that may change the position of the root node, would need to return an updated pointer to the tree(new root node).
- Instead we implement a `tree` struct
- It contains the root pointer which can be NULL
- It also contains the count of nodes in the tree
- After creating a tree, we use the same pointer for all function calls

```
    struc tree
t_count resq    1
t_root  resq    1
    endstruc
```

Creating a new tree

- The `new_tree` function allocates a `tree` struct and sets it up as an empty tree

```
new_tree:
    push    rbp
    mov     rbp, rsp

    mov     rdi, tree_size
    call    malloc
    xor     edi, edi
    mov     [rax+t_root], rdi
    mov     [rax+t_count], rdi

    leave
    ret
```

Finding a node in a tree: p = find(tree,n)

```
find:   push    rbp
        mov     rbp, rsp
        mov     rdi, [rdi+t_root]
        xor     eax, eax
.more   cmp     rdi, 0
        je      .done
        cmp     rsi, [rdi+n_value]
        jl      .goleft
        jg      .goright
        mov     rax, rdi ; note, in the book this is rsi
        jmp     .done
.goleft:
        mov     rdi, [rdi+n_left]
        jmp     .more
.goright:
        mov     rdi, [rdi+n_right]
        jmp     .more
.done   leave
ret
```

Inserting a node into a tree

Overview:

- First you check to see if the key is already in the tree
- If not, then you create a new node and set its value to the given key and set its two children to NULL.
- There is a special case for an empty tree
  - specifically, you make the newly created node the root of the tree
- If the tree is not empty, then we must traverse down the tree, going sometimes left and sometimes right to find the correct place to insert the new node

Inserting a node into a tree: p=insert(tree,n)

```
insert:
.n      equ     0
.t      equ     8
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     [rsp+.t], rdi
    mov     [rsp+.n], rsi
    call    find
    cmp     rax, 0
    jne     .done
    mov     rdi, node_size
    call    malloc
    mov     rsi, [rsp+.n]
    mov     [rax+n_value], rsi
    xor     edi, edi
    mov     [rax+n_left], rdi
    mov     [rax+n_right], rdi
```

Inserting a node into a tree

```
mov     rdx, [rsp+.t]
mov     rdi, [rdx+t_count]
cmp     rdi, 0
jne     .findparent
inc     qword [rdx+t_count]
mov     [rdx+t_root], rax
jmp     .done
```

Inserting a node into a tree

```
.findparent:
    inc     qword [rdx+t_count]
    mov     rdx, [rdx+t_root]
.repeatfind:
    cmp     rsi, [rdx+n_value] ; Remember rsi=n
    jl      .goleft
    mov     r8, rdx
    mov     rdx, [r8+n_right]
    cmp     rdx, 0
    jne     .repeatfind
    mov     [r8+n_right], rax
    jmp     .done
.goleft:
    mov     r8, rdx
    mov     rdx, [r8+n_left]
    cmp     rdx, 0
    jne     .repeatfind
    mov     [r8+n_left], rax
.done      leave
ret
```

Printing the keys in order

- We first call a non-recursive function with the tree object
- It calls a recursive function with the root node

```
;       print(t);
print:
        push    rbp
        mov     rbp, rsp
        mov     rdi, [rdi+t_root]
        call    rec_print
segment .data
.print  db       0x0a, 0
segment .text
        lea     rdi, [.print]
        xor     eax, eax
        call    printf
        leave
        ret
```

Recursive print function: rec_print(t)

```
rec_print: push    rbp
           mov     rbp, rsp
           sub     rsp, 16              ; make room to save t
           cmp     rdi, 0               ; return if t is NULL
           je      .done
           mov     [rsp+.t], rdi        ; save t, .t equ 0
           mov     rdi, [rdi+n_left]    ; print the left sub-tree
           call    rec_print
           mov     rdi, [rsp+.t]        ; print the current node
           mov     rsi, [rdi+n_value]
           lea     rdi, [.print2]        ; .print2: bd "%ld ",0
           xor     rax,rax
           call    printf
           mov     rdi, [rsp+.t]        ; print the right sub-tree
           mov     rdi, [rdi+n_right]
           call    rec_print
.done      leave
           ret
```

Homework

- Deallocate a binary tree
- Print the content of the tree layer by layer (Breadth first traversal).
  Remember you actually have a stack available

High Performance ASM

High Performance ASM

We will now discuss a couple of common and effective ways of optimizing ASM (or code in general).

- But first, it should be noted, that it is not always viable to write the most efficient code for every part of a system. Rather spend the time and effort on the bottlenecks in the system.

High Performance ASM

We will now discuss a couple of common and effective ways of optimizing ASM (or code in general).

- But first, it should be noted, that it is not always viable to write the most efficient code for every part of a system. Rather spend the time and effort on the bottlenecks in the system.

Beating the compiler

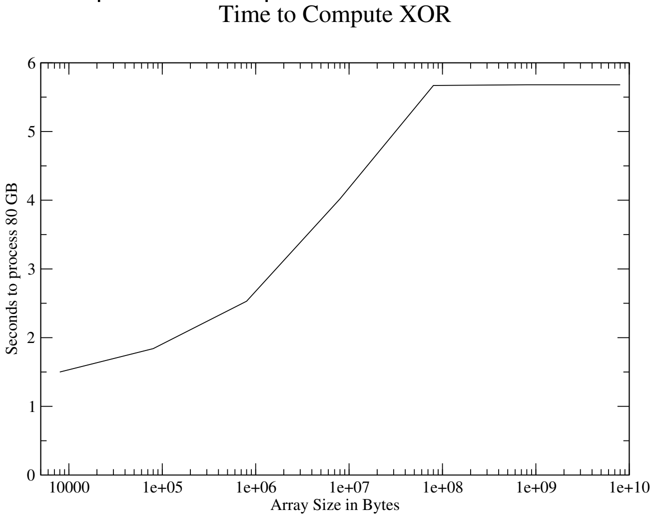
- An optimizing compiler will implement nearly all of the general optimizations
- It will do them tirelessly, missing nearly nothing
- You must usually find a non-obvious technique to get better performance than the compiler
- Learn from the complier
  - Use the `-S` option to get an assembly listing
  - Perhaps you can do the compiler's tricks better

Efficient use of cache

- Our aim is generally to keep a processor as busy as possible.
- If a CPU is working with cached data is doesn't wait around much.
- If the required data is in memory and not in the cache, a substantial amount of computational capacity is wasted while the data is retrieved.
- Organize your algorithm to exploit data locality.

Efficient use of cache: Example

- The plot below shows time versus array size for computing 10 billion exclusive or operations on quad words.



Efficient use of cache: Example

- Consider the task of matrix multiplication (with square matrices for simplicity)
-

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1N} \\ b_{21} & b_{22} & \dots & b_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N2} & \dots & b_{NN} \end{bmatrix} \tag{1}$$

$$= \begin{bmatrix} \sum_{k=1}^{N} a_{1k} b_{k1} & \sum_{k=1}^{N} a_{1k} b_{k2} & \dots & \sum_{k=1}^{N} a_{1k} b_{kN} \\ \sum_{k=1}^{N} a_{2k} b_{k1} & \sum_{k=1}^{N} a_{2k} b_{k2} & \dots & \sum_{k=1}^{N} a_{2k} b_{kN} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{N} a_{Nk} b_{k1} & \sum_{k=1}^{N} a_{Nk} b_{k2} & \dots & \sum_{k=1}^{N} a_{Nk} b_{kN} \end{bmatrix} \tag{2}$$

Efficient use of cache: Example

- Naive matrix multiplication can easily be implements with three nested loops, and is $O(N^3)$

```
for(int i=0; i<N; ++i)
    for(int j=0; j<N; ++j)
    {
        sum = 0.0;
        for(int k=0; k<N; ++k)
            sum+= a[i][k] * b[k][j]
        c[i][j]= sum;
    }
```

Efficient use of cache: Example

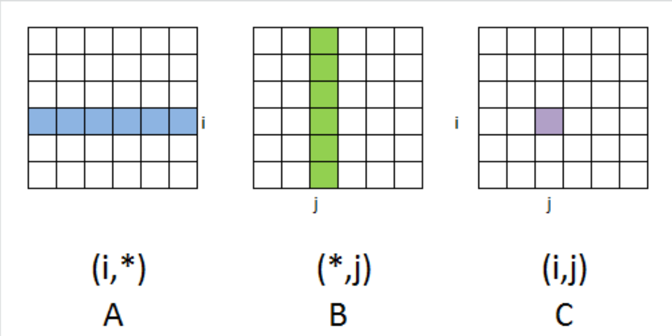
- Naive matrix multiplication can easily be implements with three nested loops, and is $O(N^3)$

```
for(int i=0; i<N; ++i)
    for(int j=0; j<N; ++j)
    {
        sum = 0.0;
        for(int k=0; k<N; ++k)
            sum+= a[i][k] * b[k][j]
        c[i][j]= sum;
    }
```

- There are three indices that are being iterated over. Which means there are at most $\binom{3}{2} = 6$ possible unique ways of iterating.
- Which way is best? We will consider three cases (which covers 6 performance profiles)

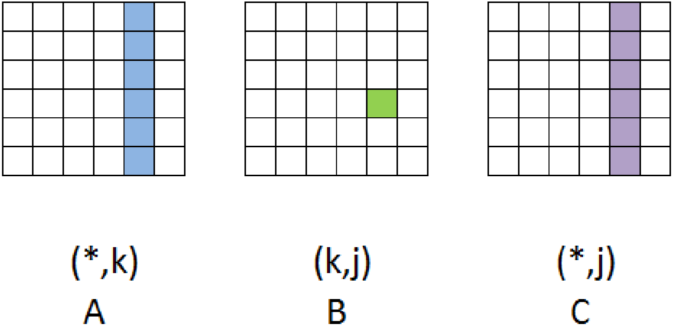
Matrix multiplication: IJK

```
for(int i=0; i<N; ++i)
    for(int j=0; j<N; ++j)
    {
        sum = 0.0;
        for(int k=0; k<n; ++k)
            sum+= a[i][k] * b[k][j];
        c[i][j]= sum;
    }
```



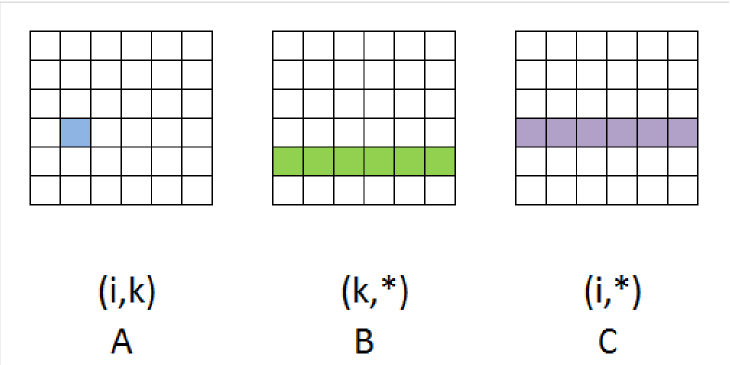
Matrix multiplication: JKI

```
for(int j=0; j<N; ++j)
    for(int k=0; k<n; ++k)
    {
        r =b[k][j];
        for(int i=0; i<N; ++i)
            c[i][j]+= a[i][k]*r;
    }
```



Matrix multiplication: IKJ

```
for(int i=0; i<N; ++i)
    for(int k=0; k<n; ++k)
    {
        r =a[i][k];
        for(int j=0; j<N; ++j)
            c[i][j]+= r*b[k][j];
     }
```



Common sub-expression elimination

- In compiler theory, common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyses whether it is worthwhile replacing them with a single variable holding the computed value.
- Simple example

```
a = b * c + g;
d = b * c * e;
```

could be replaced with

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

Strength reduction

- The idea is to rely on simpler and less computationally expensive operations.
- For example
    - Instead of dividing an integer by 8 we could shift it right 3 bits.
    - Rather than using `pow(x,3)` use $x * x * x$
    - Computer x^4 by computing x^2 and then squaring that
    - Avoid division by a floating point number x, rather computing $1/x$ and use multiplication instead
    - Getting a remainder after division by 1024, can be done using **and**.

$$x \quad and \quad (1024 - 1)$$

- It should be noted that sometimes the computationally cost of an operation is architecture dependent.

Use registers efficiently

- Place commonly-used values in registers
- When a value is used frequently it will be better to store it in a register as apposed to in memory.

Use fewer branches

- The problem with branches is they interrupt the instruction pipeline. Specifically it means a predictive pipelining might have to be undone.
- The compiler will frequently re-order blocks of code to reduce branches
- This is where conditional moves can help allot.

Convert loops to branch at the bottom

- The compiler generally does this to reduce the number of instructions in a loop and, especially, the number of branches
- Here is a C `for` loop

```
for ( i = 0; i < n; i++ )
{
    x[i] = a[i] + b[i];
}
```

- By adding an **if** at the start you can loop with a branch at the bottom
- Don't do this in C of C++ (the complier can handle this)

```
if ( n > 0 )
{
    i = 0;
    do {
     x[i] = a[i] + b[i];
     i++;
    } while ( i < n );
}
```

Unroll loops

- Use `-funroll-all-loops` to have `gcc` unroll loops
- Unrolling means repeated occurrences of the loop body with multiple parts of the data being processed
- Try to make each unrolling use different registers to reduce instruction dependence
- This frees up the CPU to do out-of-order execution
- It can do more pipelining and more parallel execution

Unroll loops

- Consider the following function that sums the element of quad words

```
add_array: ; long add_array(long* array, long size)
   xor rax,rax
   .add_qwords
      add rax,[rdi]
      add rdi, 8
      dec rsi
      jg .add_qwords
      ret
```

Unroll loops

- For simplicity let's consider a multiple of 4 sized array

```
add_array:  long add_array(long* array, long size)
   xor rax,rax
   xor r8,r8
   xor rcx,rcx
   xor rdx,rdx
   .add_qwords:
      add     rax, [rdi]
      add     r8,  [rdi+8]
      add     rcx, [rdi+16]
      add     rdx, [rdi+24]
      add     rdi, 32
      sub     rsi, 4
      jg      .add_words
   add     rcx, rdx
   add     rax, r8
   add     rax, rcx
   ret
```

Merge loops

- If 2 loops have some loop limits, consider merging the bodies
- There will be less loop overhead
- The following 2 loops can be profitably merged

```
for ( i = 0; i < 1000; i++ ) a[i] = b[i] + c[i];
for ( j = 0; j < 1000; j++ ) d[j] = b[j] - c[j];
```

- After merging values for `b[i]` and `c[i]` can be used twice

```
for ( i = 0; i < 1000; i++ )
{
   a[i] = b[i] + c[i];
   d[i] = b[i] - c[i];
}
```

Split loops

- If a loop is running over two disjoint pieces of data it might be profitable to spit the loops.
- The potential cache usage can add a greater speed increase than the extra overhead incurred by having two loops.
- Real world testing if often the best way to determine if the split is worth it.

Move loop-invariant code outside the loop

- Simple example

```
for(i=0;i<N;++i)
{
    myarray[i]*=pow (4.5,2.5)
}
```

Now if this is executed exactly as stated, the `pow` function will be called `N` times. Rather just compute it once, as reuse the result.

Remove recursion

- Eliminating tail-recursion is generally useful
- If you have to simulate a “stack” like recursion gives you, recursion will probably be faster

Eliminate stack frames

- Use `-fomit-frame-pointers` with `gcc`
- Use this for release code
- Using the `rbp` register is optional
- Leaf functions don't even need to worry about stack alignment
  - Unless you are using some local data requiring 16 byte alignment

Inline functions

- The compiler can do this painlessly
- In assembly you will make your code less readable
- Explore using macros

Reduce dependencies to allow super-scalar execution

- Use different registers to try to reduce dependencies
- The CPU has multiple computational units in 1 core
- You can benefit from out-of-order execution
- You can get more out of pipelines
- You can keep more computational units busy
- For example from our earlier add_array function

```
add     rax, [rdi]
add     r8,  [rdi+8]
add     rcx, [rdi+16]
add     rdx, [rdi+24]
```

  The order of execution does not matter, so super-scalar execution is possible.

Use specialized instructions

- The compiler will have a harder time doing this than you
- There are also SIMD integer instructions
- Use SIMD floating point instructions
- The AVX instructions are a new feature which allow twice as many floating point values in the SIMD registers

Use specialized instructions

Remember the question: Assume `A` and `B` have been defined in the data segment as quad words. Write the necessary ASM to calculate the hamming distance between `A` and `B` and store the result in the `rax` register.

- Using special instruction this is very easy

Use specialized instructions

Remember the question: Assume `A` and `B` have been defined in the data segment as quad words. Write the necessary ASM to calculate the hamming distance between `A` and `B` and store the result in the `rax` register.

- Using special instruction this is very easy
- ```
  mov rax, [A]
  xor rax, [B]
  popcnt rax,rax
  ```
  ```