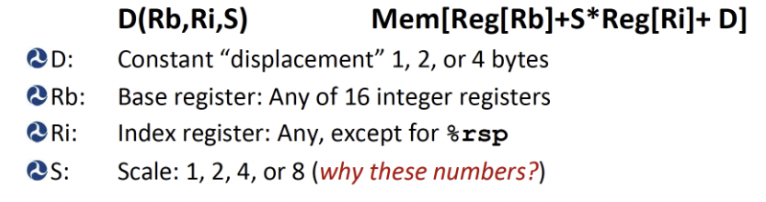
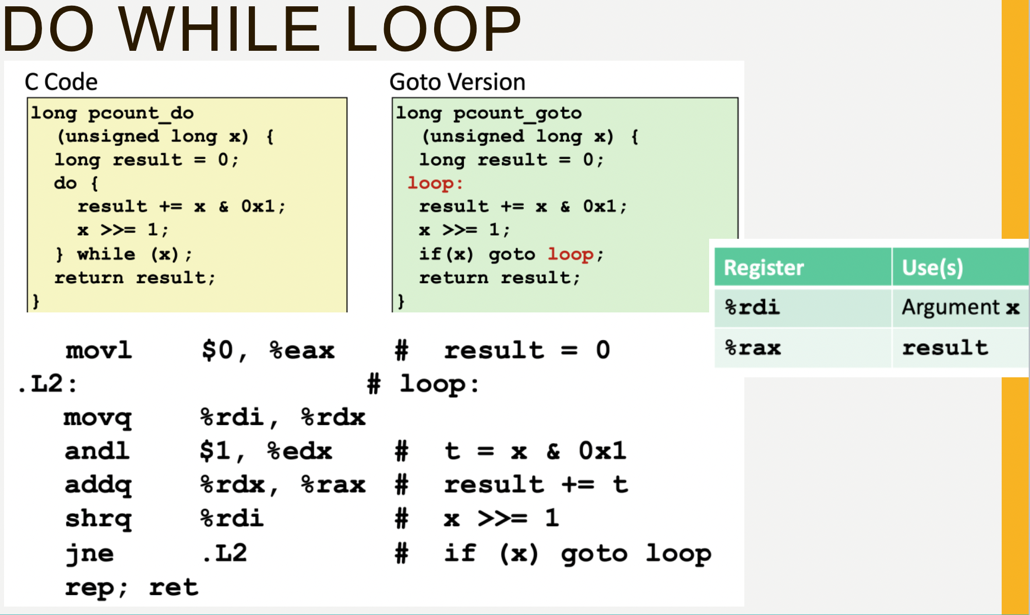


When call to func x occurs the stack grows (decrements) and stores the address of the next instr (after the call) so we know where to return to after func x. The rsp changes to point at the lower address bc the stack grew.

**Memory Model:**

- Basic unit is a bit (0 or 1)

- 8 bits make up a byte

- Interpreting a series of bits is usually done either in base 2 (binary) OR base 16 (hex)

**Binary**:

- base 2

- multiply number at a single spot by 2^n where n is that number's position from the left starting from 0

- ex: 0101 --> 0\*2^3 + 1\*2^2 + 0\*2^1 + 1\*2^0 = 0 + 4 + 0 + 1 = 5

- in C: 0b[binary]

**Hex**:

- a way of viewing a series of 4 bits as one hex-digit since it's base 16 and a 4 bit binary sequence can hold 16 digits

- represented with 0-9 A-F (where 0-9 represent the numbers 0-9 and A-F represent the numbers 10-15)

- converting from hex to decimal: same as binary but instead of 2^n, we use 16^n, AND instead of just multiplying each hex digit by 1 or 0, you multiply it by whatever the hex at that spot is as a number

- ex: 1D --> 1\*16^1 + 13\*16^0 = 16 + 13 = 29

- in C: 0x[hex]

- ex: 0b0101 --> 0x5

- ex: 0b1111 1010 --> 0xFA

**Unary Operators:**

- NOT [~]:

- flips all bits in a binary sequence

- ex: ~11110000 == 00001111

- AND [&]:

- used on 2 binary sequences; conducts a bit-wise operation on the sequence

- useful for masking to get certain values: create a sequence with a certain number of 1s, & it with another value, and you get part of the other value

- ex: 10010101 & 00001111 --> 00000101 only the last 4 digits of the initial value

- ex: 10010101 & 11101111 --> 10000101 we got rid of the 5th bit

- OR [|]:

- used on 2 binary sequences like "&"

- if there's at least one 1, it will give a 1; if both are 0, it gives a 0

- XOR [^]:

- also used on 2 binary sequences

- if the 2 bits are the same (0,0 or 1,1), it will give a 0; if the bits are different (0,1 or 1,0), it will give a 1

**Shifting:**

- takes the sequence of bits & literally shifts it to the left or right

- left shift syntax: x << n # will shift variable x to the left n times

- right shift syntax: x >> n # will shift variable x to the right n times

- LEFT SHIFT [<<]:

- super straight forward; it will move the digits to the left, eating up the most significant digits and filling the right in with 0s

- ex: 10101010 >> 1 --> 01010100

- ex: 11001010 >> 3 --> 01010000

- really useful for multiplying: left shifting by n will multiply by **2^n**

- you can use combinations of multiplications to get a specific multiplication (ex: x << 3 - x << 1 == x\*6)

- be careful about overflowing (if most significant digit is 1, it will overflow with a left shift)

- RIGHT SHIFT [>>]:

- arithmetic right shift: used for signed representations (int); the new most significant digit after the shift is whatever the most significant digit before the shift was

- ex: 1000 >> 1 --> 1100

- ex: 0111 >> 2 --> 0001

- logical right shift: used for unsigned representations; the new most significant digit is always 0

- the type of shift is auto-determined by the compiler based on what you're shifting

- right shifting is useful for dividing by 2^n (similar to how left shifting multiplies by 2^n except right shift divides)

- your shift amount (n) cannot be < 0 OR >= #bits in data type

**Integers:**

- for the rest of this, assume that the representation is w bits long

- Unsigned:

- represented in binary

- U\_min = 0

- U\_max = 2^w - 1 = 111...11

- unsigned values CANNOT be negative (watch out for ux <= 0 comparisons as they will ALWAYS be true)

- constants are unsigned by default

- to create an unsigned, you can cast it with (unsigned) x OR have a "U" at the end of the binary

- ex: 1010U & 0101u are unsigned

- Signed (Two's Complement):

- represented in binary with the most significant digit being viewed as a negative number

- T\_min = -2^(w-1) = 100...00

- T\_max = 2^(w-1) - 1 = 011...11

- -1 = 111...11

- think of the most sig digit as a massive negative bias, and the rest of the digits are smaller positive biases that you add

- (-x) = ~x + 1

- \***note**: this negation formula will NOT work with T\_min because |T\_min| = |T\_max| + 1

- U\_max = 2\*T\_max + 1

- representing U(x) & T(x) in hex is the same (just convert the decimal to hex)

- Multiplication & division work the same with left/right shifts; just make sure you account for overflow (too negative --> positive)

- Converting between U(x) & T(x) & casting:

- nothing changes in terms of the binary, but the digits are just interpreted using the other method

- ex: (int) 0b1010 = -6; (unsigned) 0b1010 = 10

- when doing operations with both signed and unsigned, the int is auto-casted to an unsigned & the **result is unsigned**

**Byte Ordering:**

- Big Endian: stores the bytes in the way you would normally read it from the start of memory to the end of memory (most significant byte first)

- Little Endian: stores the bytes in the opposite way you would read it from the start of memory to the end of memory (least significant byte first)

**Registers:**

- 16 main registers used; denoted with a "%" sign before the register name. Used for really fast access memory; 64 bits long (8 bytes)

- ex: %rdi

- %rax is usually used to store the return value of the function being executed

- Sub-registers:

- the "r" prefix --> 64 bit register (the full thing)

- ex: %rdi

- "e" prefix --> 32 bit sub-registers (the lower-significance half of the "r" register that they belong to)

- example: %edi

- for the number registers (i.e. %r8 - %r15): it's the register name followed by "d"

- ex: %r12d is the 32 bit sub-register of %r12

- just the base of the register's name (no "r" or "e") --> 16 bit subregister (the lower-significance half of the "e" sub-register)

- example: %di

- for the number registers (i.e. %r8 - %r15): it's the register name followed by "w"

ex: %r12w is the 16 bit sub-register of %r12

- just the first letter of the base register's name followed by "h" or "l" to denote the higher/lower-significance half of the base registers --> 8 bit sub-register

- ex: %dh --> upper half of %di

- ex: %dl --> lower half of %di

- for the number registers (i.e. %r8 - %r15): it's the register name followed by "b"

ex: %r12b is the 8 bit sub-register of %r12 (lower half)

- Special Registers:

- %rax: always stores the return value of a function

- %rsp: stack pointer

- %ebp: base pointer

**Register Operations:**

- Usually, operations take 2 arguments: Src & Dest in that order (for the most part)

- There are 3 things that an operation can take as arguments:

- Immediates AKA constant integer data (i.e. 5, 0xFF, 111...11, etc)

- immediates are usually denoted by a "$" before the actual number/binary

- ex: $-533

- Registers: literally whatever is in one of the 16 64 bit registers

- denoted by the register name

- ex: %edi

- Memory: 8 consecutive bytes of memory whose address is stored in a register

- the register stores the literal memory location of the data we want (think of it as a pointer)

- denoted by the register which stores the memory address in parenthesis (think of the parenthesis as dereferencing a pointer)

- ex: (%rax) --> go to the memory address stored in %rax & use the 8 bytes of data that exists there

**Simple & Complete Memory Addressing Modes:**

- Normally, you would access memory by having parenthesis around a register to get the data at the location "pointed to" by that register

- Simple memory addressing mode is just a way to access data at a memory location stored at a location displaced by a constant:

- denoted by a constant before the memory:

- ex: 8(%rdx) --> go to the memory stored 8 bytes after the memory stored in the location stored in %rdx (follow the %rdx pointer, and use the data in memory 8 bytes after that)

- D(%R) == Mem[Reg[R] + D]

- Complete Memory Addressing Mode:

- Another way to access a certain item in memory but with more ways to displace a base register's address

- Displacement is denoted the same way (an immediate before the parenthesis)

- D: 1, 2, or 4 bytes

- There is a base register (the first argument in the parenthesis)

- There is an index register (can be anything EXCEPT %rsp) which is multiplied by the scale (the third argument, an immediate) that's added to the base register

- S: 1, 2, 4, or 8 bytes

- D(Rb, Ri, S) == Mem[Reg[Rb] + S\*Reg[Ri] + D]

- All of these fields do NOT have to be filled out; it can work without a bunch of them (if they don't exist, it's just 0)

**Operations:**

- all operations can be performed on different numbers of bits, usually denoted by the last letter in the operation (similar to the different clarities of the same register)

- q: does the operation on the whole 8 bytes/64 bits

- l: 4 bytes/32 bits

- w: 2 bytes/16 bits

- b: 1 byte/8 bits

movq Src, Dest:

- copies the value in Src and puts it in Dest; at the end, both Src & Dest store the thing that was initially in Src

- you cannot move to an immediate (should be obvious; the Dest cannot be immediate)

- you cannot move from memory to memory directly; you have to move from memory to registers to memory

leaq Src, Dest:

- Src is an address mode expression

- The memory address in Src is copied into Dest; NOT the contents of the memory address, but the actual address

- the machine code equivalent of: p = &x[i]

- useful for computing arithmetic operations of form x + k\*x where k = 1, 2, 4, or 8 (the scale factor in complete memory addressing mode)

- ex: leaq (%rdi, %rdi, 2), %rax <--> %rax = %rdi + 2\*%rdi

Arithmetic Operations:

- addq Src,Dest Dest = Dest + Src

- subq Src,Dest Dest = Dest - Src

- salq Src,Dest Dest = Dest << Src # AKA shlq (it's a left shift so logical vs arithmetic doesn't matter)

- imulq Src,Dest Dest = Dest \* Src

- sarq Src,Dest Dest = Dest >> Src

- shrq Src,Dest Dest = Dest >> Src

- xorq Src,Dest Dest = Dest ^ Src

- andq Src,Dest Dest = Dest & Src

- orq Src,Dest Dest = Dest | Src

One Operand Operations:

- incq Dest Dest = Dest + 1

- decq Dest Dest = Dest - 1

- negq Dest Dest = -Dest

- notq Dest Dest = ~Dest

**Condition Codes:**

- 1 bit flags set by arithmetic operations every time they're performed (NOT set by leaq)

- CF: set to 1 if UNSIGNED overflow occurred

- ZF: set if the Dest is set to 0 (when the arithmetic operation results in an answer of 0)

- SF: set if the Dest is < 0 (when the result of the arithmetic operation results in a negative number)

- OF: set if there's a SIGNED overflow (when the result of the arithmetic operation has a different sign than the original Dest & Src)

**Compare (cmp):**

- explicitly sets the condition codes (usually used right before conditional jumps)

- cmp Src,Dest # performs Dest - Src for the sole purpose of setting flags; doesn't store the result anywhere

- CF: set if there's overflow viewing Src & Dest as UNSIGNED integers

- ZF: set if Dest-Src = 0 (i.e. Src == Dest)

- SF: set if Dest-Src < 0 (i.e. Src > Dest)

- OF: set if there's a SIGNED overflow

**Test (test):**

- explicitly sets condition codes similar to cmp, but performs Src&Dest

- useful when using masks

- test Src,Dest # performs Dest & Src for the sole purpose of setting flags; doesn't store the result anywhere

- ZF: set if Src&Dest == 0

- SF: set if Src&Dest < 0

**Jump (jX):**

- jump instructions literally jump to other parts of the code

- used a LOT with if statements, for loops, etc

- generally used right after a cmp or test instruction

- cmp Src,Dest:

- jmp # unconditional jump (1)

- je # jump if Dest == Src (ZF)

- jne # jump if Dest != Src (~ZF)

- js # jump if Dest-Src < 0 (SF)

- jns # jump if Dest-Src >= 0 (~SF)

- jg # jump if Dest > Src (~(SF^OF) & ~ZF)

- jge # jump if Dest >= Src (~(SF^OF))

- jl # jump if Dest < Src (SF^OF)

- jle # jump if Dest <= Src ((SF^OF) | ZF)

- ja # jump if Dest > Src (unsigned) (~CF & ~ZF)

- jb # jump if Dest < Src (unsigned) (CF)

**Goto (goto):**

- literally goes to the position given by the label (i.e. goto Here will go to Here)

- conditional expressions (if statements) with goto:

- Expression: val = Test ? Then\_Expression : Else\_Expression;

- Pseudocode:

ntest = !Test;

if (ntest) goto Else;

val = Then\_Expression;

goto Done;

Else:

val = Else\_Expression;

Done:

. . .

**Conditional Moves (if statements):**

- Most optimal way to do this is to avoid jumps when possible

- sets value to one "branch" and checks condition; switches value if necessary & continues

- Expression: val = Test ? Then\_Expression : Else\_Expression;

- pseudocode:

result = Then\_Expression;

temp\_eval = Else\_Expression;

nt = !Test;

if (nt) result = temp\_eval;

return result;

- example in actual C & assembly:

- C code:

long absdiff (long x, long y)

{

long result;

if (x > y)

result = x-y;

else

result = y-x;

return result;

}

- Assembly:

absdiff:

movq %rdi, %rax # x

subq %rsi, %rax # result = x-y

movq %rsi, %rdx

subq %rdi, %rdx # eval = y-x

cmpq %rsi, %rdi # x:y

cmovle %rdx, %rax # if <=, result = eval

ret

**While, Do-While, & For loops:**

- all of these are optimized the same way to use an initial test, and then the loop to minimize the number of jumps

- pseudocode:

if (!Test) # checks condition at the beginning (while & for loops need this test; do-while loops can have it anyways)

goto done;

loop: # the meat of the loop

Body;

if (Test) # check for condition

goto loop;

done:

...

**Switch Statements:**

- Overview: they create a jump table that stores all the addresses of the instructions for each possible value of the switch

- switch values (i.e. the variable you're comparing) will be manipulated to start at 0 and go to (Your\_Max\_Value - Your\_Min\_Value)

- ex: if your switch checks the values 1, 3, & 5, the jump table will have jumps for n = 0 to n = 4

- Jump Table:

- you can think of this as an array that stores pointers (not really pointers) to each condition of the switch statement

- the switch statement will find the correct place to go to by using \*.JumpTable(, %rdi, 8) where %rdi stores your truncated input (0 to Max-Min), .JumpTable is the start of the jump table & 8 is the spacing between each value of the jump table

- the effective address of the correct body is .JumpTable + %rdi\*8

- fallthroughs are handled with either a goto at the end of the specific condition that takes control to the next condition OR by just allowing control to fall through by not having a ret instruction at the end of the condition & making sure the next instruction is the one you're falling through to

- default cases are handled with a jump to the a specific condition that handles the default case (look for multiple jumps to the same tag in the jump table; this will probably be the location of the default)

**The Stack:**

- the stack is a group of memory with the largest addresses

- stores local variables

- grows downward (just like a normal stack) & the top is kept track of with %rsp (the stack pointer register)

- push Src: will first decrement %rsp by 8 bytes & then copy Src to the top (or bottom in terms of memory addresses) of the stack

- pop Dest: will first copy the top 8 bytes on the stack into Dest & then increment %rsp by 8 bytes

**Control Flow (call):**

- call label: will move control (kept track of with %rip AKA the instruction pointer) to point to label

- function arguments are stored in the following registers in order:

%rdi # first parameter

%rsi # second parameter

%rdx # third parameter

%rcx # fourth parameter

%r8 # fifth parameter

%r9 # sixth parameter

- additional parameters are stored on the stack with parameter 7 being on the top, 8 under 7, 9 under 8...

- %rax stores the return value of the function

- the first thing pushed onto the stack when making a function call is the return address (this will be in the caller's scope), followed by any parameters and then free space for local variables

- the return address & additional parameters are automatically pushed by the call instruction

- CALLER SAVED REGISTERS:

%rax # the caller needs to save its own return value before calling another function which will return it's own value

%rdi, %rsi,..., %r8, %r9 # all the arguments need to be saved by the caller; cannot assume callee won't modify them

%r10 & %r11 # temp variables that can be used by callees; caller must save these if it's using them

- CALLEE SAVED REGISTERS (all of these must be saved & restored by the callee before returning):

%rbx, %r12 - %r14, %rbp, %rsp

**Arrays:**

- 1D arrays:

- length = L, datatype T, size of each element = K

- total size = K\*L

- stored sequentially in memory

- the array variable will just be a pointer to the start of the array in memory with another register storing the index

- accessing elements (%rdi stores the pointer to the start of the array & %rsi stores the index):

(rdi, rsi, S)

- ex: accessing 4th element of an int array: (rdi, 3 /\* index \*/, 4 /\* size of each element \*/)

- to iterate through the array, just iterate %rsi (or whatever register is storing the index)

- Multidimensional Arrays:

- type T, # of rows = R, # of columns = C, size of each element = K

- total size = K\*R\*C

- row major ordering: each row is stored sequentially & each (i.e. the sub-arrays are stored as a normal array & all the sub arrays are stored one after another as a 1D array of arrays)

- accessing Arr[i][j] --> Arr + (i\*C + j)\*K

- ex:

# %rdi stores i, %rsi stores j, arr is the pointer to the start of the array

leaq (%rdi,%rdi,4), %rax # 5\*i

addl %rax, %rsi # 5\*i+j

movl arr(,%rsi,4), %eax # %rax = (arr + 4\*(5\*i+j)) [%rax stores the actual value at arr[i][j]]

**Structs:**

- memory for structs is stored in the order it's declared

- struct alignment = alignment of the largest element in the struct

- char: 1 byte

- short: 2 bytes (0)

- int, float: 4 bytes (00)

- double, long, char\*: 8 bytes (000)

- long: 16 bytes (0000)

- struct size & start address have to be multiples of the size of the largest primitive in the struct

- saving space: put the largest elements first

