CS 33 Note Guide

Taught by Prof. Glenn Reinman

Notes by Dylan Gunn

Spring Quarter 2020

SECTION 1: BITS AND BYTES

Representing Information as Bits

- Everything is Bits
  - Everything (data, instruction, storage, flow of storage) all in bits
    - Each bit is a 0 or 1 (ie. on or off state, really is high/low voltage)
    - Data & instructions encoded into bits
  - Why bits?
    - Electronic implementation of computer, are other implementations but focus on binary here
- What can/will we do?
  - Base 2 Number Representation
    - Rather than ones, tens, hundreds places, use ones, twos, fours, eights, etc.
  - Encoding Byte Values
    - Eight bits per byte, cluster of bits representing byte can represent:
      - Binary (00000000_2 to 11111111_2)
      - Decimal (0_{10} to 255_{10})
      - Hexadecimal (00_{16} to FF_{16})
        - Use characters 0 to 9 and A to F (A is 10, B is 12, etc.)
        - One hex digit represents four binary bits
  - Memory dumps in hex: relate address to contents at address in hex (16 pairs of hex digits, represents 16 bytes)
    - Addresses go up by 16 each time (0020 to 0030) for all 16 bytes
      - Specific bytes in each row addressed in ones place (001<u>4</u>)
- Bits don't only represent values; can represent characters (ASCII)
  - Depends on how interpreted; can be interpreted as digit or char
    - Normally, chosen using data types to understand

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	-	-	10/16
pointer	4	8	8

Bit-Level Manipulations

- Use boolean algebra, perform operations on the bit level
    - Operate on Bit Vectors
        - Operations applied bitwise (last bit in byte run through first bit)
        - and (&) or (|) xor (~) not (^)
    - Can apply these to any integral data type
        - long, int, short, char, unsigned
    - Contrast: DON't CONFUSE WITH LOGIC OPERATIONS
        - &&, ||, !
        - Return only a 0 or 1 (0x00, 0x01)
    - Shift Operations
        - Left Shift: x << y
            - Shift bit vector x left y positions, throw away extra left bits and fill 0's on right
        - Right Shift: x >> y
            - Shift bit vector x right y positions, throw away extra right bits and:
                - Logical shift: fill 0's on left
                - Arithmetic shift: replicate most significant (leftmost) bit

Argument x	**01100010**
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00***0**11000

Argument x	**10100010**
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11***1**01000

SECTION 2: INTEGERS

Encoding Integers

- Binary Form: Have a set of bits, 8 bits makes a byte
  - Goes up by power of 2 each time; base 2 set, normal is base 10
  - 00101010 is 2+8+32+128 = 170
  - Two's Complement Form: Allows for negatives. Most significant bit is negative, everything less significant is positive. 10001001 is 1+16-256 = -239
- Hexadecimal Form: Same, but base 16; uses 0-9 and A-F
  - 2 places in hex make up an entire byte
- Numeric Ranges
  - UMin to UMax
    - 00000000 00000000 to 11111111 11111111
    - 0 to 65535
  - TMin to TMax
    - 10000000 00000000 to 01111111 11111111
    - -32768 to 32767
  - Observations:
    - | TMin | = TMax + 1
    - UMax = 2*TMax + 1

Conversion & Casting

- Pos to pos is fine, as long as no 1 in most significant for unsigned. So can't do U2T or T2U if 1 in most significant; either too large of a positive (U2T) or neg to pos (T2U)
- Constants
  - By default, integers are signed
  - Unsigned if add "U" as a suffice, ie. 100U
- Casting
  - Explicit Casting: Same as U2T and T2U
    - int tx, ty;
      unsigned ux, uy
      tx = (int) ux;
      uy = (unsigned) ty;
  - Implicit Casting: Occurs via assignments and procedure calls
    - tx = ux;
      uy = ty;
  - Casting Surprises
    - Expression Evaluation
      - If there is a mix, signed are implicitly cast to unsigned

Ex: For W = 32 (TMin = -2,147,483,648; TMax = 2,147,483,647)

Constant_1	Constant_2	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Expanding & Truncating

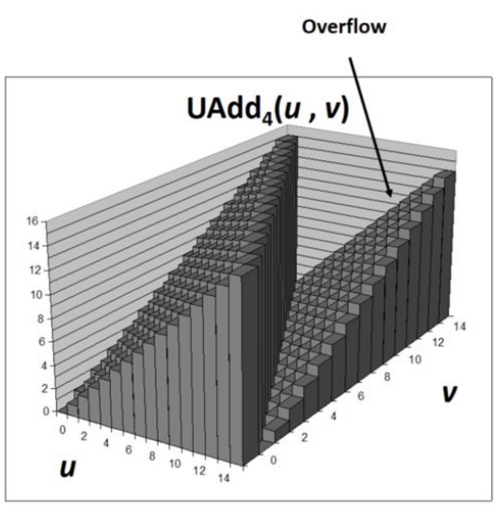
- Sign Extension (Expanding, short int to int)
    - Unsigned: Add 0's to the left side
    - Signed: Add the bits to the left side, replicating most significant bit

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Truncating (unsigned to unsigned short)
    - For any, bits truncated and result reinterpreted
    - Unsigned: Equivalent modulo operation with most significant bit truncating to
        - 32 bit to 16 bit is done by essentially doing val % 16
    - Signed is similar, but some diffs because of negative values

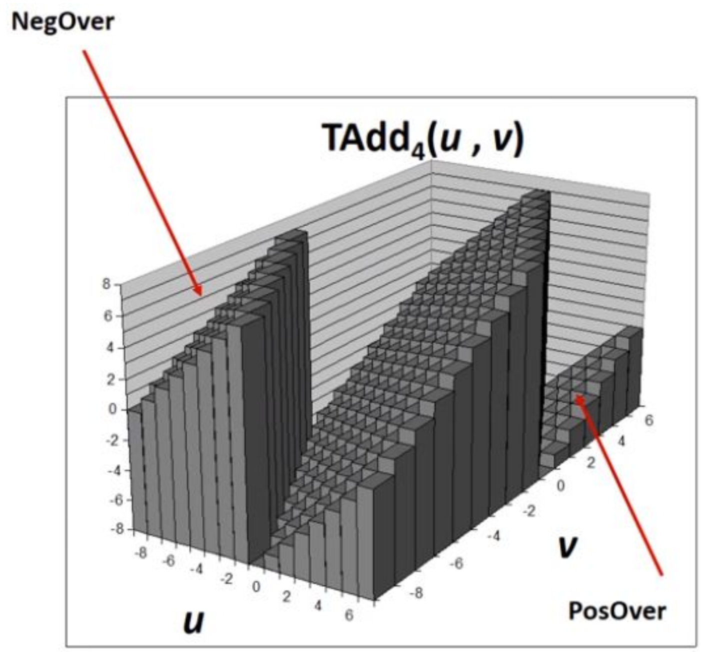
Addition, Negation, Multiplication, Shifting

- Addition/Subtraction
    - If adding two operands with w bits, result would be w+1 bits (True Sum)
    - Standard addition ignores carry output
        - UAdd(u,v) = u + v mod 2^w



For a 4 bit unsigned integer, max is 15; here we see u + v mod 16

- For signed integers, will overflow in both positive and negative directions
    - Can only overflow once per operation



For a 4 bit signed integer, min is -8 & max is 7; this displays expected behavior

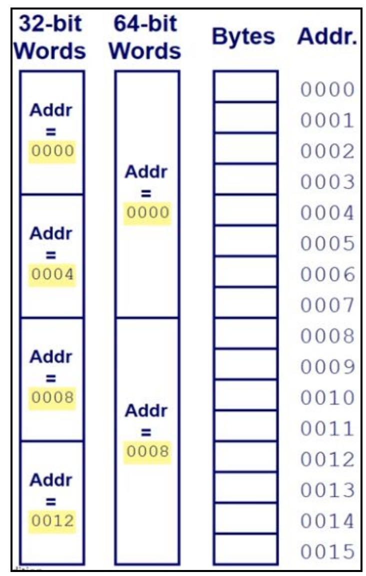
- Multiplication
    - Number of Bits
        - Unsigned: Would need up to 2w bits (True Product)
        - Two's Complement min (neg): Up to 2w-1 bits (True Product)
        - Two's Complement max (pos): Up to 2w bits (True Product)
        - Normally don't expand, but if done in software use "arbitrary precision"
    - Unsigned Multiplication
        - At most 2w with arbitrary precision, w without
        - Standard ignores high order w bits, modular arithmetic: u * v mod 2^w
    - Signed Multiplication
        - At most 2w bits with arbitrary precision, w without
        - Standard ignores high order w bits, can cause overflow
    - Power-of-2 Multiply with Shift
        - Multiplication very expensive, cheaper to shift
        - u << k give u * 2^k
        - Both signed and unsigned
        - Strength reduction switches to shifts and adding/subtracting
            - Ex: u << 3 == u * 8
            - Ex: (u << 5) - (u << 3) == u * 24
- Division
    - Very similar in method but no potential for overflow
    - Floor Function: performs via right shift, so only int values maintained
        - Ex: 11 >> 1 == 11 / 2 == 5

When to use Unsigned

- Very easy to make mistakes and go out of range
- Also may accidentally compare an int to unsigned, now unsigned result
- So why should you?
    - When Performing Modular Arithmetic
        - Multiprecision arithmetic
    - When Using Bits to Represent Sets
        - Logical right shift, no sign extension

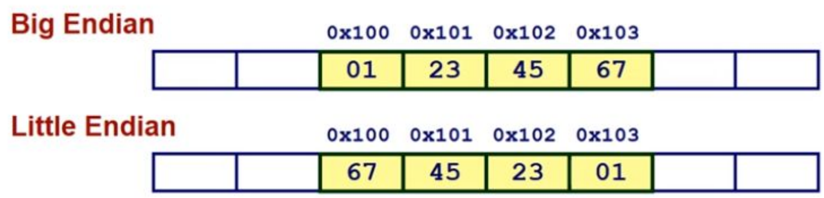
Representation in Memory, Pointers, & Strings

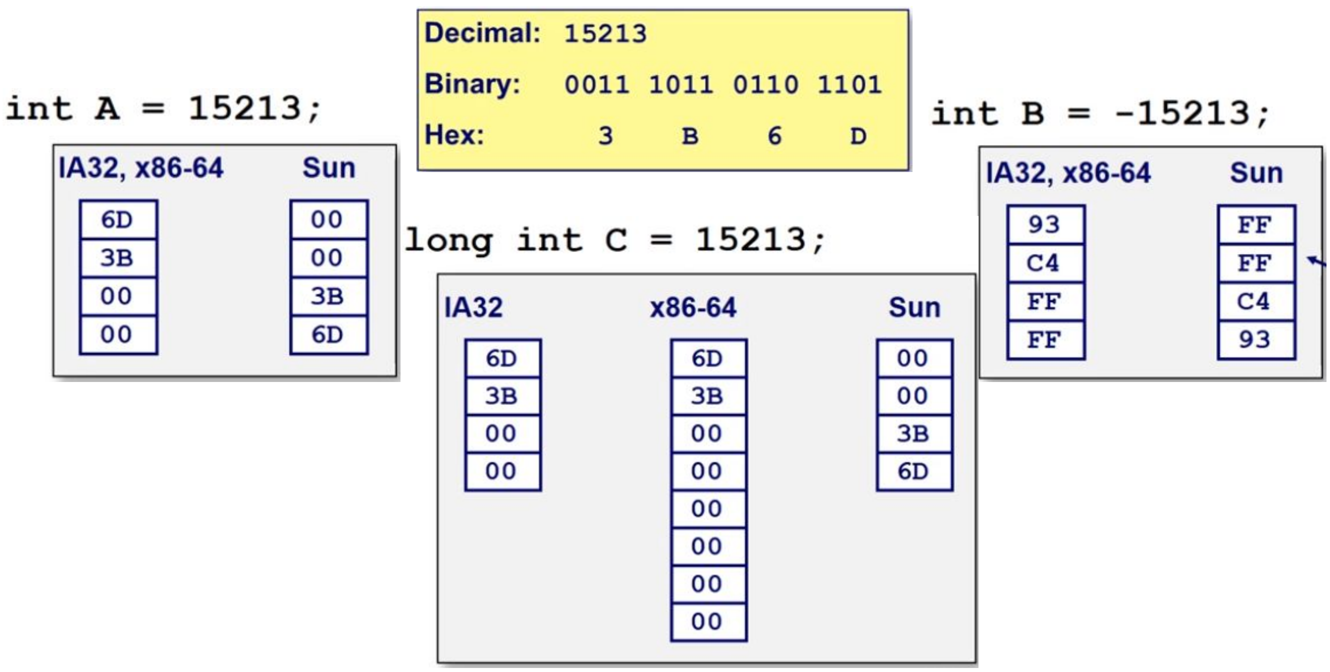
- Programs refer to data by address; bytes
    - Can conceptually imagine as a large array of bytes
    - An address is like an index of that array and a pointer stores an address
- System provides private address spaces to each "process"
    - Process is program being executed
    - Keeps it so program cannot modify data of other programs
- Machine Words
    - Any given computer has a "Word Size"
        - Nominal size of integer-valued data and of addresses (64-bit, 32-bit)
    - Most machines ran 32 bits (4 bytes) as word size
        - Limits addresses to 4GB (2^{32} bytes)
    - Increasingly today, we run 64-bit word size
        - Potentially 18 EB (exabytes) of addressable memory
        - $18.4 * 10^{18}$ bytes
    - Bits are all contiguous (one after another)
    - Machines still support multiple data formats



Byte Ordering

- Bytes are ordered in their multi-byte words in two conventional ways
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
    - The way you would think of it
  - Little Endian: x86, ARM processors running Android, iOS, Windows
    - Least significant byte has lowest address
    - Intuitively backwards by byte





- Strings
  - Strings are stored the same in both big-endian and little-endian
    - str[0] always at lowest address, str[i] always at str + i address
  - Each element in the array is stored according to its endian-ness
  - Represented as an array of characters, each in ASCII format
  - Should be null-terminated (ie. final character = 0)

Integer C Puzzles

- if x < 0, then x * 2 < 0

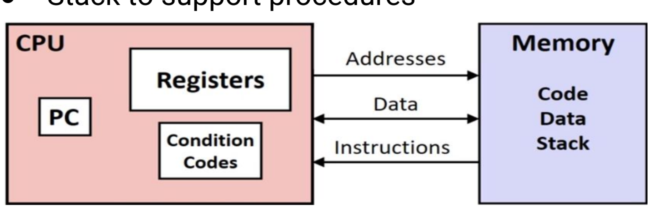
Initialization

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

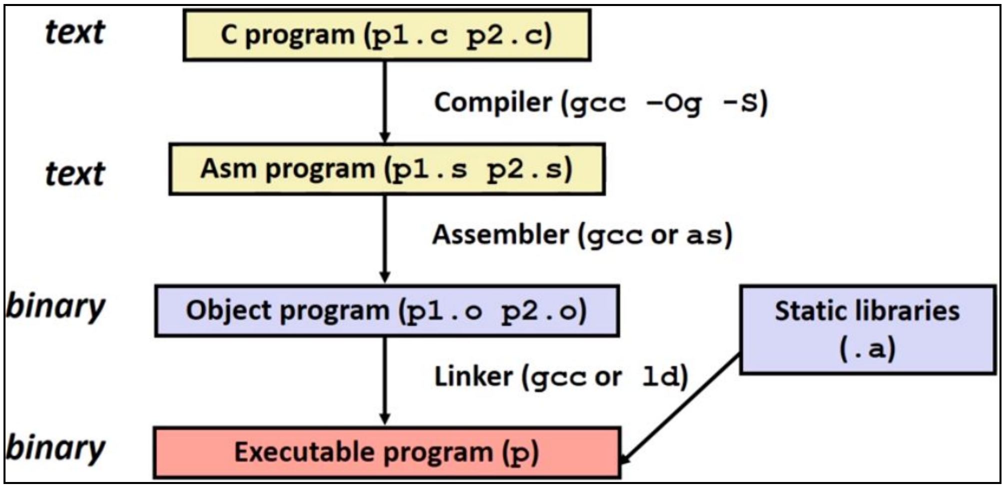
If x < 0, then x * 2 < 0	False, could overflow
ux >= 0	True
If x & 7 == 7, then (x << 30) < 0	True
ux > -1	False, unsigned comparison
If x > y, then -x < -y	False, if y is TMin then -y is TMin again because -x = ~x + 1
x * x > 0	False, can overload
If x > 0 && y > 0, then x + y > 0	False, overload
If x >= 0, then -x <= 0	True
If x <= 0, then -x >= 0	False, TMin
x | -x >> 31 == -1	False, 0
ux >> 3 == ux/8	True
x >> 3 == x/8	False, negative rounds down
x & (x - 1) != 0	False, 0

SECTION 3: MACHINE LEVEL PROGRAMMING
PART 1: BASICS
History of Intel Processors & Architectures
- Intel x86 Processors
  - Evolutionary Design
    - Backwards compatible up until 8086, introduced in 1978
    - Added features as time has gone on, improved speed and efficiency
  - Complex Instruction Set Computer (CISC)
    - Many different instructions with diff formats
      - Only small subset encountered with Linux programs
    - Hard to match performance of Reduced Instruction Set Computers (RISC)
      - Intel has done this in terms of speed, now as much for low power

C, Assembly, & Machine Code
- Definitions
  - Instruction Set Architecture (ISA): Parts of a processor design that one needs to understand to write assembly/machine code
    - Memory (place for values)
    - Instructions (moves from one state to the next)
    - Program (set of instructions; put in memory)
    - Execution Model (when do we execute each instruction)
  - Microarchitecture: Implementation of the architecture
    - Cache sizes, core frequency
  - Code Forms
    - Machine Code: Byte-level programs that a processor executes
    - Assembly Code: A text representation of machine code
- Assembly/Machine Code View
  - Programmer-Visible State
    - PC: Program Counter
      - Address of next instruction
      - Called RIP, IP meaning instruction pointer (x86-64)
    - Register File
      - Heavily used program data
    - Condition Codes
      - Store status information about most recent arithmetic/logical operation
      - Used for conditional branching
    - Memory
      - Byte addressable array
      - Code (individual instructions) and user data sections
      - Stack to support procedures



- Turning C into Object Code
  - Code in text files
  - Compile using "gcc -Og -s p1.c p2.c -o p
    - Use basic optimizations (-Og) (turns off advanced advanced optimizations, allows assembly program to look more like C for debugging purposes)
    - -s creates assembly program text
    - Then machine-readable binary is created (object program)
    - Object files are combined to make complete machine level code
    - Put resulting binary in file *p*



- Assembly Characteristics: Data Types
  - "Integer" data of 1, 2, 4, 8 bytes
    - Data values
    - Addresses (untyped pointers)
  - Floating point data of 4, 8, 10 bytes
  - Code: Byte sequences encoding series of instructions
    - CISC type, so may have instructions made up of single bytes and others of multiple
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory
- Assembly Characteristics: Operations
  - Perform arithmetic function on register data, memory data, or literals (constants parts of individual instruction)
  - Transfer data between memory and register
    - Load data into memory from register
    - Store register data into memory
  - Transfer Control
    - Unconditional jumps to/from procedures
    - Conditional branches

- Object Code
    - Assembler
        - Translates .s into .o
        - Binary encoding of each instruction
        - Nearly-complete image of executable code
        - Missing linkages between code in diff files
    - Linker
        - Resolves references between files
        - Combines with static run-time libraries
            - eg. code for malloc, printf
        - Some libraries are dynamically linked
            - Linking occurs when program begins execution
- Machine Instruction Example

```
*dest = t;
```

- C Code
    - Store value t where designated by dest

```
movq %rax, (%rbx)
```

- Assembly
    - Move 8-byte value to memory (make copy of location)
        - Quad words in x86-64 parlance
    - Operands
        - t: Register %rax
        - dest: Register %rbx
        - *dest: Memory M[%rbx]

```
0x40059e:   48 89 03
```

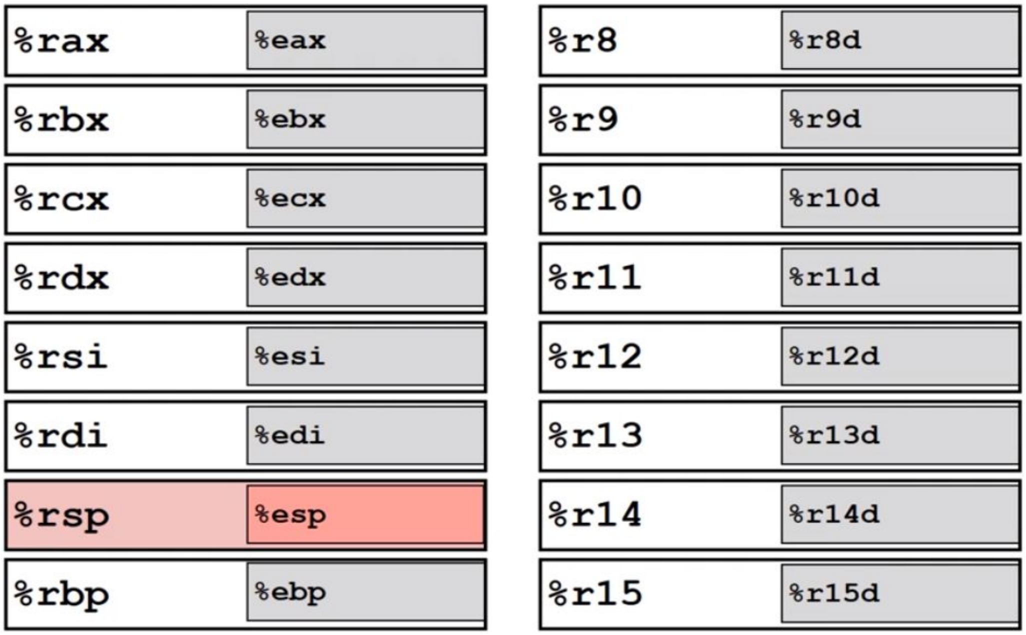
- rax is t, dest is an address, rbx holds address of dest
- Parentheses indicate dereferencing of the pointer

- Disassembling Object Code
    - Disassembler:
      `objdump -d sum`
        - Useful for examining object code
        - Analyzes bit pattern of series of instructions
        - Produces approximate rendition of assembly code
        - Can be run on either a.out (complete executable) or .o file
    - Within gdb Debugger:
      `gdb sum`
      `disassemble sumstore`
        - Disassemble procedure, doesn't show individual bytes

      `x/14xb sumstore (x is examine, / is formatter: 14 bytes in hexadecimal (x) in terms of bytes (b))`
        - Examine the 14 bytes starting at sumstore
- What Can be Disassembled
    - Technically anything, but things like Microsoft products would break EULA

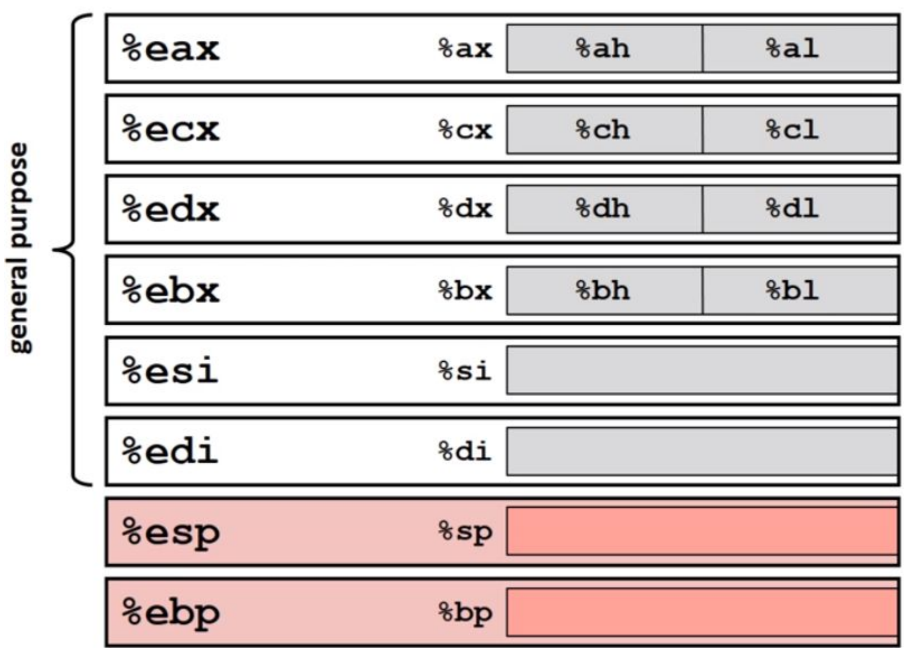
Assembly Basics: Registers, Operands, and Move

x86-64 Integer Registers



In 32 bit systems, use the grey 4 byte registers (lower 4). In 64 bit systems, use the entire 8 bytes. Can still access individual 4 byte registers if necessary

- Most are general purpose registers
- %rsp is a stack pointer, will focus on it later
- Some History: IA32 Registers
  - Each of the 8 byte registers also had 16 bit system 4 byte registers
  - 64 bit rax has 32 bit eax has 16 bit ax has two 8 bit ah and al
  - Keep these 16-bit virtual registers for backwards compatibility



- Moving Data

  `movq source,destination`

  - Move 16 bit quantities
  - Operand Types
    - Immediate (aka a literal): Constant integer data, part of the instruction
      - Example: `$0x400, %-533`
      - Like C constant, but prefixed with '$'
      - Encoded with 1, 2, or 4 bytes
    - Register: One of 16 integer registers
      - Example: `%rax, %r13`
      - But `%rsp` for special use
      - Others have special uses for particular instructions
    - Memory: 8 consecutive bytes of memory at address given by register
      - Simplest example: `(%rax)`
        - Two levels of indirection: indicates register file, so first there, then use 64-bit value in `%rax` as address into memory and pull out 8 consecutive bytes
      - Various other "address modes"

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

- Simple Memory Addressing Modes
  - Normal: `(R)` `--->` `Mem[Reg[R]]`

    `movq (%rcx),%rax`

    - Take value in memory `%rcx` and place value into register `%rax`
  - Displacement: `D(R)` `--->` `Mem[Reg[R]+D]`

    `movq 8(%rbp),%rdx`

    - Take value in memory `%rbp`, add 8, place value into `%rdx`
    - Register R specifies start of memory region
    - Constant displacement D specifies offset
  - Example: Swap Function

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

- Complete Memory Addressing Modes
  - Most General Form

    `D(Rb,Ri,S)     --->     Mem[Reg[Rb]+S*Reg[Ri]+D]`

    - D: Constant “displacement”; 1, 2, or 4 bytes
    - Rb: Base register - Any of 16 integer registers
    - Ri: Index register - Any, except %rsp
    - S: Scale - 1, 2, 4, 8 (power of 2, easy to perform multiplication with shifting)
  - Special Cases

```
(Rb,Ri)      --->     Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)     --->     Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)    --->     Mem[Reg[Rb]+S*Reg[Ri]]
```

Examples:

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Arithmetic & Logical Operations

- Address Computation Instruction

  `leaq Source,Destination`

    - `Source` is address mode expression
    - Set `Destination` to address denoted by expression
  - Difference with move: memory is not dereferenced in `leaq`
  - Uses
    - Computing addresses without a memory reference
      - ie. translation of `p = &x[i];`
    - Computing arithmetic expressions of the form x + k*y
      - k = 1, 2, 4, or 8

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

- Some Arithmetic Operations (-q suffix bc 64-bit values)

Format	Computation
`addq  Src,Dest`	`Dest = Dest + Src`
`subq  Src,Dest`	`Dest = Dest - Src`
`imulq Src,Dest`	`Dest = Dest * Src`
`salq  Src,Dest`	`Dest = Dest << Src`
`sarq  Src,Dest`	`Dest = Dest >> Src`
`xorq  Src,Dest`	`Dest = Dest ^ Src`
`andq  Src,Dest`	`Dest = Dest & Src`
`orq   Src,Dest`	`Dest = Dest | Src`
`incq  Dest`	`Dest = Dest + 1`
`decq  Dest`	`Dest = Dest - 1`
`negq  Dest`	`Dest = -Dest`
`notq  Dest`	`Dest = ~Dest`

Important Review

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rdx`	Argument `z`
`%rax`	`t1`, `t2`, `rval`
`%rdx`	`t4`
`%rcx`	`t5`

SECTION 3: MACHINE LEVEL PROGRAMMING
PART 2: CONTROL
Control: Condition Codes

- Implicit Setting
    - Single Bit Registers
        - CF: Carry Flag (for unsigned)
        - SF: Sign Flag (for signed)
        - ZF: Zero Flag
        - OF: Overflow Flag
    - Implicitly Set (think of as side effect) by arithmetic operations
      Ex: `addq Src,Dest` <---> `t = a+b`
        - CF set if carry out from most significant bit (unsigned overflow)
        - ZF set if t == 0
        - SF set if t < 0 (as signed)
        - OF set if two's complement (signed) overflow
          `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>0)`
    - Not set by `leaq` instruction
        - Does not use memory dereferencing OR condition codes
- Explicit Setting: Compare
    - `(cmpq b,a)` like computing `(a-b)` without setting destination
        - CF set if carry out from most significant bit (used for unsigned comparison)
        - ZF set if `a == b`
        - SF set if `(a-b) < 0` (as signed)
        - OF set if two's complement (signed) overflow
          `(a>0 && b>0 && (a-b)<0) || (a<0 && b<0 && (a-b)>0)`
- Explicit Setting: Test
    - `(testq b,a)` like computing `(a&b)` without setting destination
    - Sets condition codes based on value of Src1 & Src2
        - ZF set if `a&b == 0`
        - SF set if `a&b < 0`
    - Useful to have one of the operands be a mask
- Reading Condition Codes: SetX Instructions
    - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
    - Does not alter remaining 7 bytes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use `movzbl` to finish job
    - 'z' means to pad the rest with 0, if did 's' would pad with sign extension
    - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
cmpq   %rsi, %rdi    # Compare x:y
setg   %al           # Set when >
movzbl %al, %eax     # Zero rest of %rax
ret
```

Conditional Branches

- Jumping
  - jX Instructions: Jump to diff part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

- General Conditional Expression Translation (Using Branches)

  C Code:

  ```
  val = Test ? Then_Expr : Else_Expr
  Ex) val = x>y ? x-y : y-x;
  ```

  Using goto:

  ```
    ntest = !Test;
    if (ntest) goto Else;
    val = Then_Expr;
    goto Done;
  Else:
    val = Else_Expr;
  Done:
    . . .
  ```

- Using Conditional Moves
  - Will do move, but only if a test passes
  - Useful bc avoids need to branch, or change, instruction flow in any way
  - Checks for data value written rather than conditional branch, which checks whether an instruction pointer is changed
  - Why?
    - Processors consist of pipelines
    - In order to keep efficient, continuously feed to pipeline
    - Conditional branches need to be evaluated before they're fed
    - Conditional moves don't require control transfer

Using goto:

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Conditional Move Example

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
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
```

- Bad Cases for Conditional Move
  - Expensive Computations
    `val = Test(x) ? Hard1(x) : Hard2(x);`
    - Both values computed
    - Only makes sense when computations very simple
  - Risky Computations
    `val = p ? *p : 0;`
    - Both values computed, would try to dereference p even if p is null
    - May have undesirable results
  - Computations with Side Effects
    `val = x>0 ? x*=7 : x+=3`
    - Both values get computed, both will be computed even though only want one
    - Must be side-effect free

Loops

- Do-While

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Example:

Register	Use(s)
`%rdi`	Argument **`x`**
`%rax`	`result`

```
    movl    $0, %eax     #  result = 0
.L2:                     # loop:
    movq    %rdi, %rdx
    andl    $1, %edx     #  t = x & 0x1
    addq    %rdx, %rax   #  result += t
    shrq    %rdi         #  x >>= 1
    jne     .L2          #  if (x) goto loop
    rep; ret
```

- While (Without Optimization, used with -Og):

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

- While (Optimized, used with -O1)

While version

```
while (Test)
  Body
```

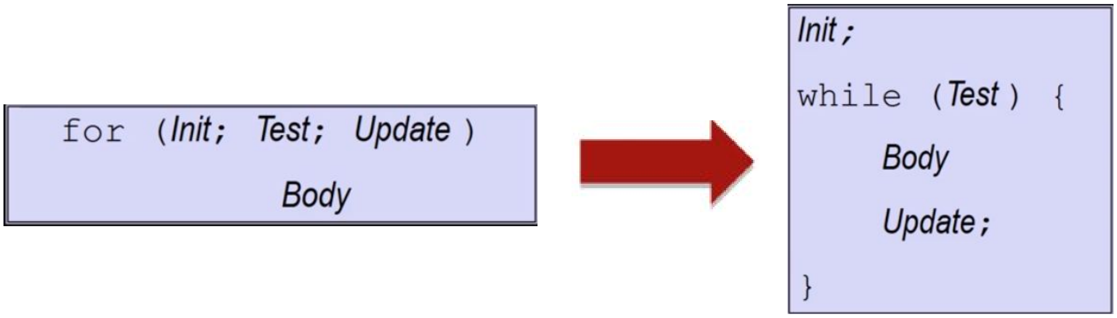
Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

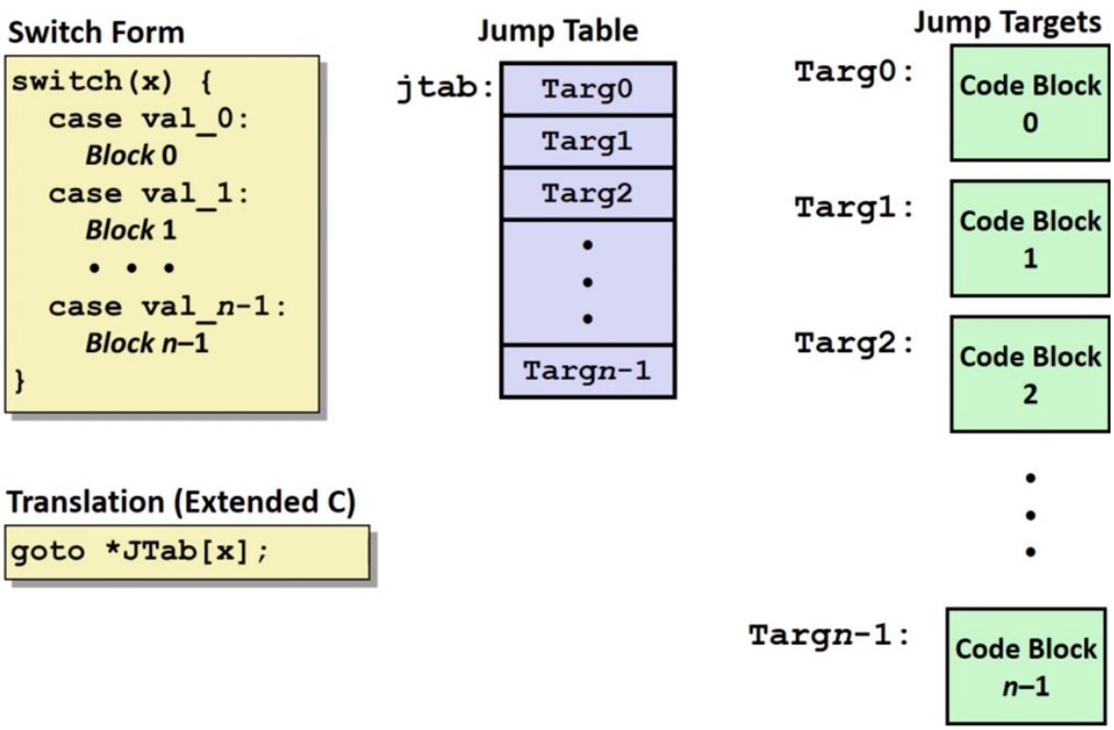
```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

- For
  - Use For-While Conversion

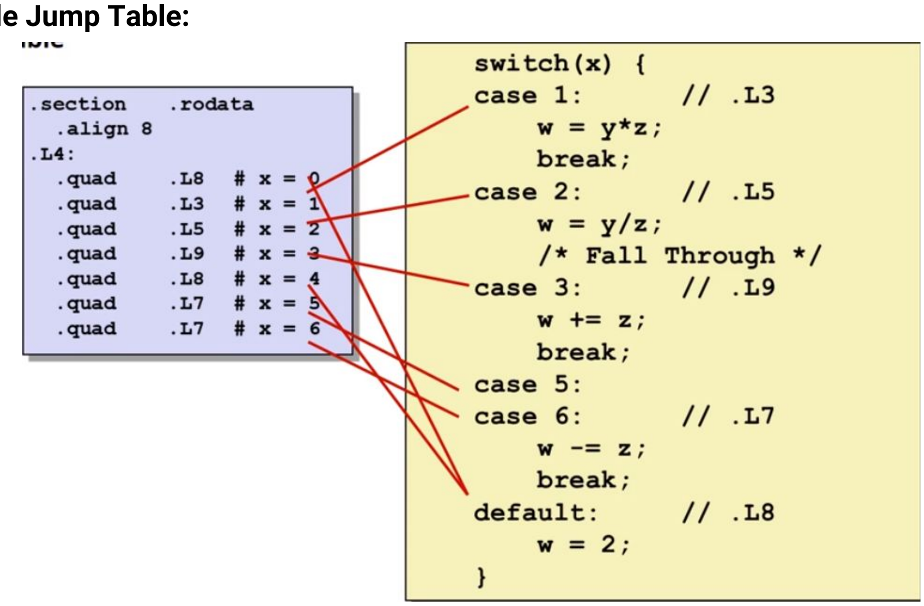


Switch Statements

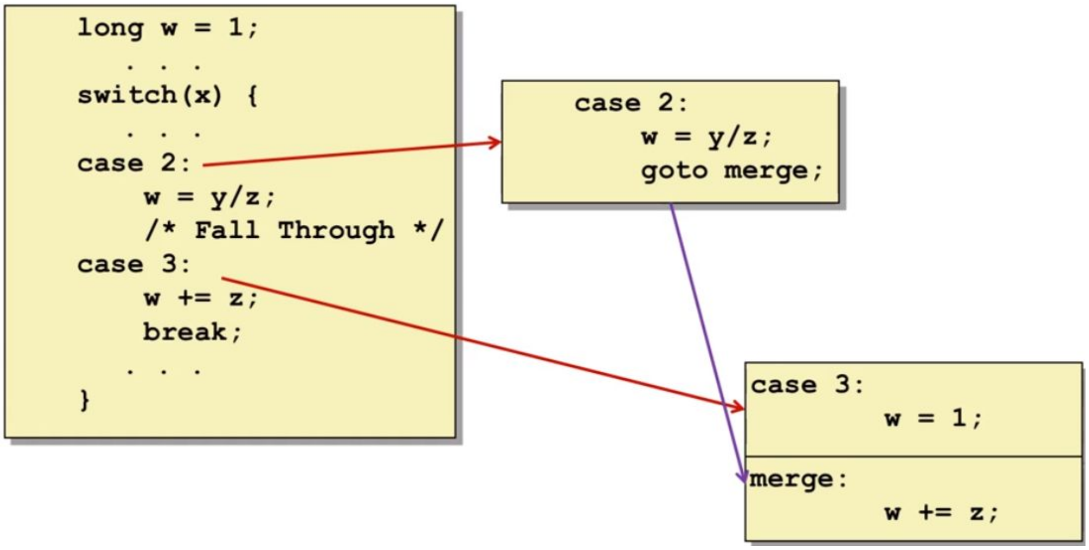
- Jump Table Structure
  - Switch statements are implemented with jump tables



Example Jump Table:



- When needs to carry through, will use a merge



Complete Example

```c
    long w = 1;
      . . .
    switch(x) {
      . . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L5:                        # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx         #  y/z
   jmp     .L6          #  goto merge
.L9:                        # Case 3
   movl    $1, %eax     #  w = 1
.L6:                        # merge:
   addq    %rcx, %rax   #  w += z
   ret
.L7:                      # Case 5,6
  movl  $1, %eax     #  w = 1
  subq  %rdx, %rax   #  w -= z
  ret
.L8:                      # Default:
  movl  $2, %eax     #  2
  ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

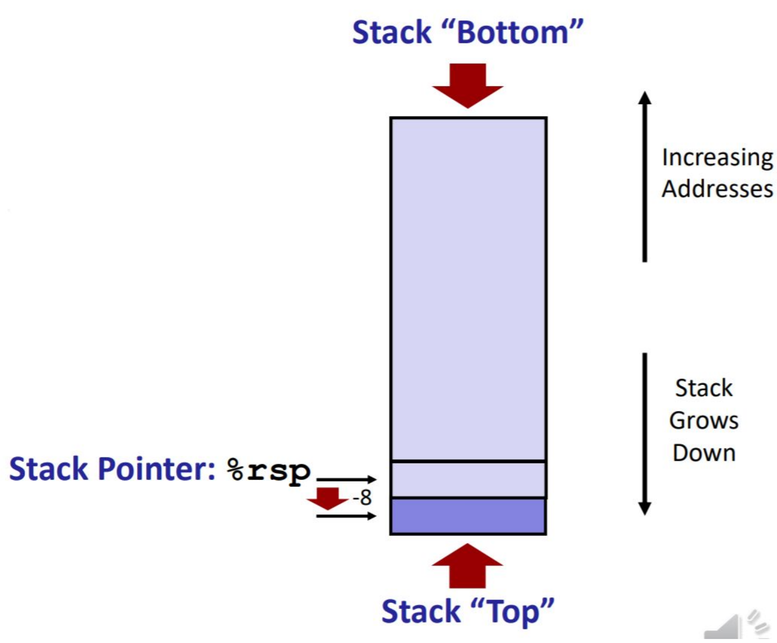
Section 3: Machine Level Programming
Part 3: Procedures

- Passing Control
  - To beginning of procedure code, back to return point
- Passing Data
  - Procedure arguments, return value
- Memory Management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

Stack Structure

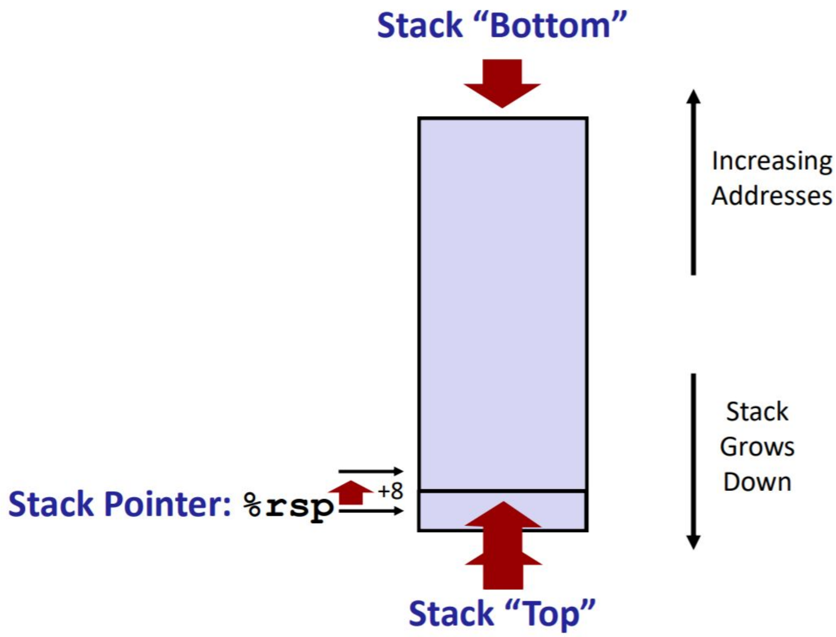
- x86-64 Stack
  - Region of memory managed with stack discipline
  - Grows toward lower addresses
  - Register `%rsp` contains lowest stack address
    - Address of "top" element
- Push

  `pushq Src`
  - Fetch operand at Src
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



- Pop

  `popq Dest`
    - Read value at address given by `%rsp`
    - Increment `%rsp` by 8
    - Store value at `Dest` (must be register)

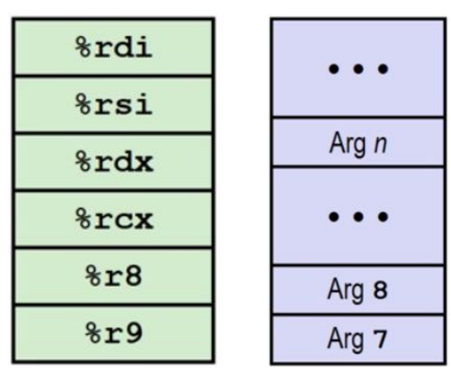


Calling Conventions: Passing Control

- Procedure Control Flow
    - Stack support procedure call and return by providing local variables, notion of scope, and some mechanism to return and store register values
    - Procedure Call: `call (callq in x86-64) label`
        - Push return address onto stack
        - Jump to label (changes instruction pointer)
    - Return Address
        - Address of the next instruction right after call
    - Procedure Return: `ret`
        - Pop address from stack
        - Jump to address

Calling Conventions: Passing Data

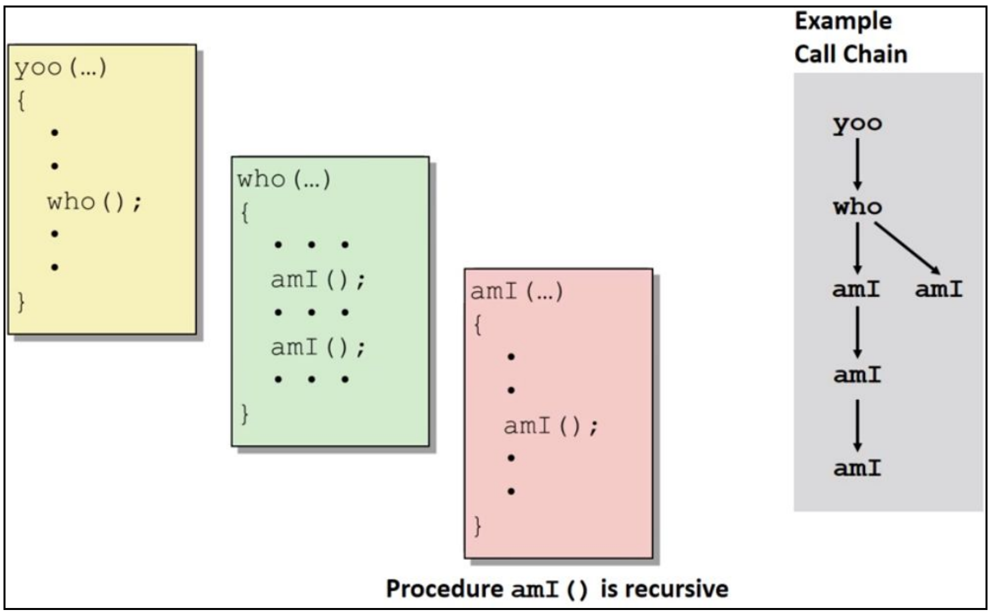
- Procedure Data Flow
    - Two places where can store parameters to pass from one procedure to another
        - Registers: First 6 arguments, much faster than memory
        - Stack: Only allocated when need it
            - Stored in reverse order
            - **`%rax` holds return value**



Calling Conventions: Managing Local Data

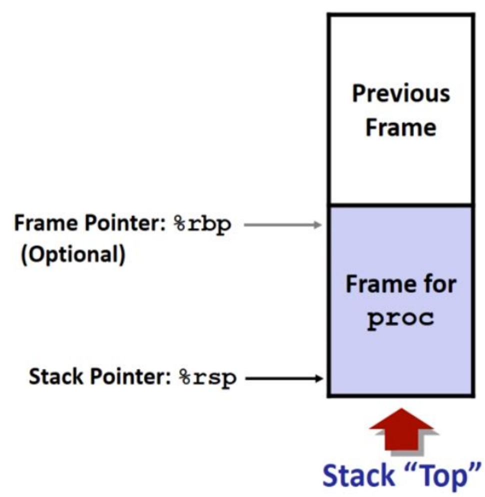
- Stack-Based Languages
  - Stack Discipline
    - Agreed upon set of constraints on assembly language
    - Order of storing arguments, variables, and how to clean up before and after procedure calls
  - Stack Allocated in Frames
    - Stack grows in large jumps and data added to stack is called frame
    - Frame: State for single procedure instantiation

Call Chain Example



yoo calls who, who calls amI, amI calls self two more times, second amI in who no recursion

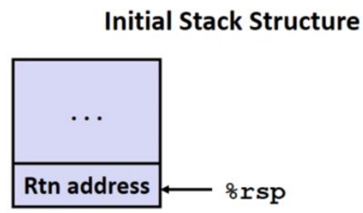
- Stack Frames
  - Contents
    - Return information
    - Local storage (if needed)
    - Temporary space (if needed)
  - Management
    - Space allocated when enter procedure
      - "Set-up" code
      - Includes push by `call` instruction
    - Deallocated when return
      - "Finish" code
      - Includes pop by `ret` function



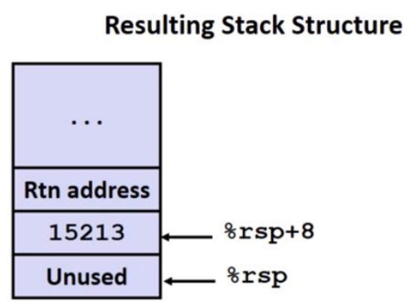
- x86-64/Linux Stack Frame
  - The caller (yoo) calls the callee (who)
  - Current (Callee) Stack Frame (“Top” to Bottom)
    - “Argument build:” Parameters if another function called inside
    - Local variables if can’t keep in registers
    - Saved register context
    - Old frame pointer (optional)
  - Caller Stack Frame
    - Return address
      - Pushed by `call` instruction
    - Arguments for this call

Example

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

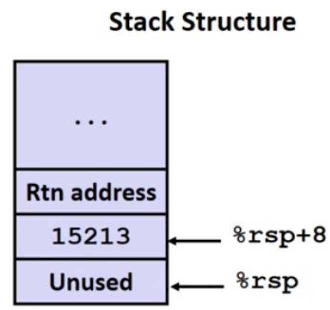



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



  - Subtract 16 from %rsp so enough space told hold return address and any temporary registers that could require
    - Allocate 16 bytes worth of space, create v1

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rdi	&v1
%rsi	3000

- Move 3000 into %esi (could have used %rsi too, but equivalent bc upper bits zero'd)
- Place 8+(address of %rsp) into %rdi (leaq, so storing location of 15213, not value)
- %rdi now holds &v1 [(%rsp)+8]
- Then, call of incr sets v1 [(%rsp)+8] to 18213
- Add the 18213 to the return value
- Pop from the stack
- Return

- Register Saving Conventions
    - When procedure yoo calls who:
        - yoo is caller
        - who is callee
    - Can register be used for temp storage?

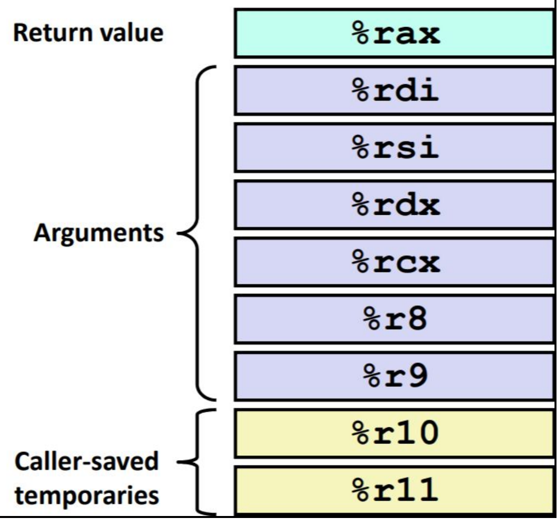
```
yoo:
   • • •
   movq $15213, %rdx
   call who
   addq %rdx, %rax
   • • •
   ret
```

```
who:
   • • •
   subq $18213, %rdx
   • • •
    ret
```

        - Contents of `%rdx` overwritten by `who`
            - Need some coordination
    - Conventions
        - Caller Saved
            - Caller saves temp values in its frame before the call
        - Callee Saved
            - Callee saves temp values in its frame before using
            - Callee restores them before returning to caller

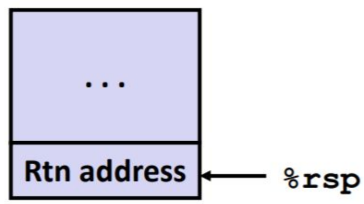
Caller Saved

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi`, ..., `%r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved
  - Can be modified by procedure



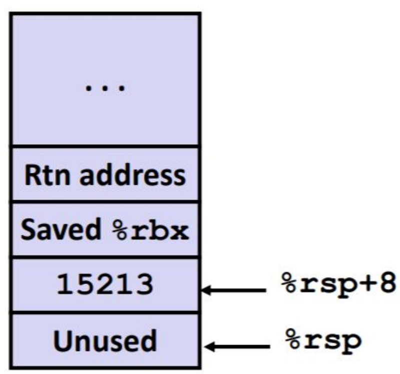

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Initial Stack Structure




```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

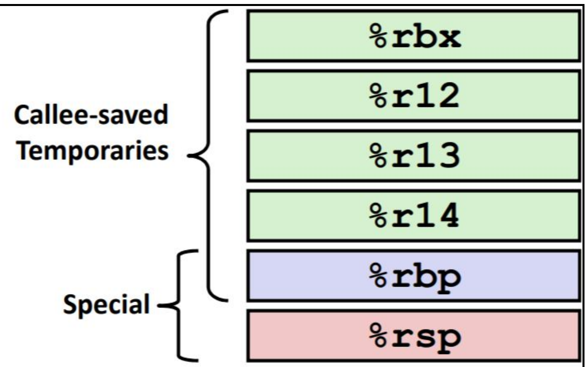
Resulting Stack Structure



s adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Callee Saved

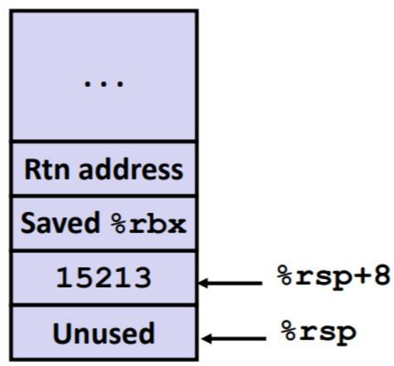
- `%rbx, %r12, %r13, %r14`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure




```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure

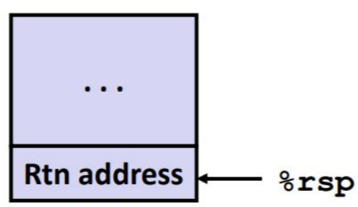


Illustration of Recursion

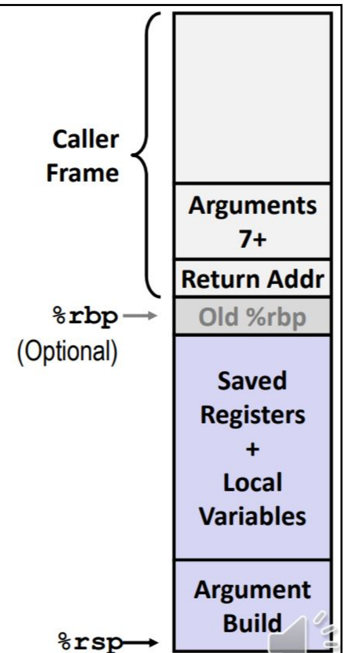
```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

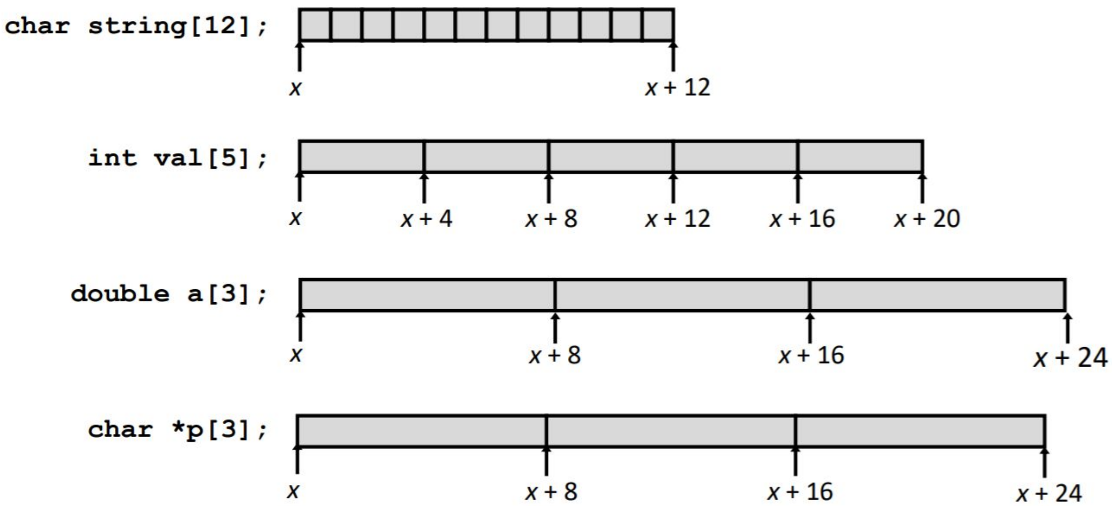
- Observations About Recursion
  - Handled without special consideration
    - Stack frames mean that each function call has private storage
      - Saved registers and local variables
      - Saved return pointer
    - Register saving conventions prevent one function call from corrupting another's data
      - Unless the C code explicitly does so
    - Stack discipline follows call / return pattern
      - If P calls Q, then Q returns before P
      - Last-In, First-Out
  - Also works for mutual recursion
    - P calls Q; Q calls P

Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion (& mutual recursion) handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax`
- Pointers are addresses of values
  - On stack or global



SECTION 3: MACHINE LEVEL PROGRAMMING
PART 4: DATA
Arrays: One-Dimensional
- Basic Principle
  `T A[L];`
  - Array of data type T and length L
  - Contiguously allocated region of `L*sizeof(T)` bytes in memory

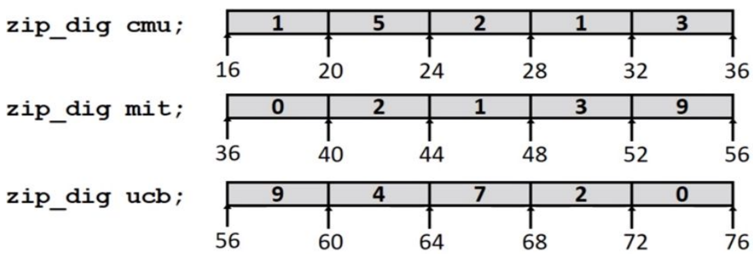


  - Identifier A can be used as pointer to array element 0: Type `T*`

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	x
`val+1`	`int *`	$x + 4$
`&val[2]`	`int *`	$x + 8$
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	$x + 4i$

Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration "`zip_dig cmu`" equivalent to "`int cmu [5]`"

```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

X86-64

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- `%rdi` hold first parameter, `%rsi` holds second

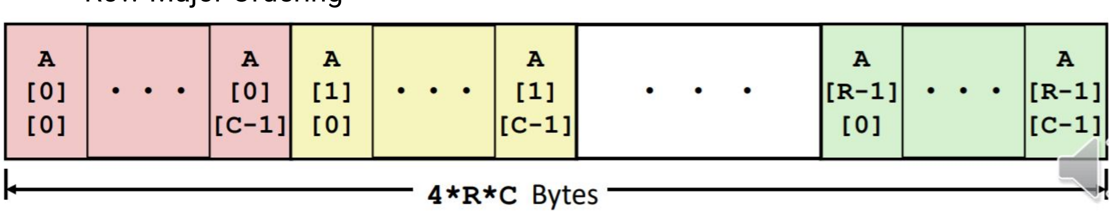
- Array Loop

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax          #   i = 0
  jmp     .L3               #   goto middle
.L4:                        # loop:
  addl    $1, (%rdi,%rax,4) #   z[i]++
  addq    $1, %rax          #   i++
.L3:                        # middle
  cmpq    $4, %rax          #   i:4
  jbe     .L4               #   if <=, goto loop
  ret
```

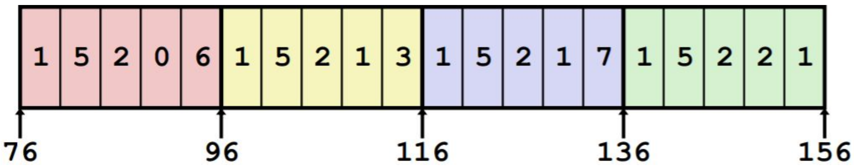
Arrays: Multi-Dimensional (Nested)

- Declaration
  `T A[R][C];`
  - 2D array of type T, has R rows and C columns
  - Type T requires K bytes
- Array Size: R*C*K
- Arrangement
  - Row-Major Ordering



Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Array of 4 elements, each element is an array of 5 ints

- Row Vectors
  - `A[i]` is array of C elements
  - Each element of type T requires K bytes
  - `Address A[i] = A + i*(C*K)`

Example: To return row vector `pgh[index]`

```
 # %rdi = index
  leaq (%rdi,%rdi,4),%rax # 5 * index
  leaq pgh(,%rax,4),%rax   # pgh + (20 * index)
```

- Elements
  - `A[i][j]` is array of C elements
  - Each element of type T requires K bytes
  - `Address A[i][j] = A + i*(C*K) + j*K`

Example: To return integer `pgh[index][dig]`

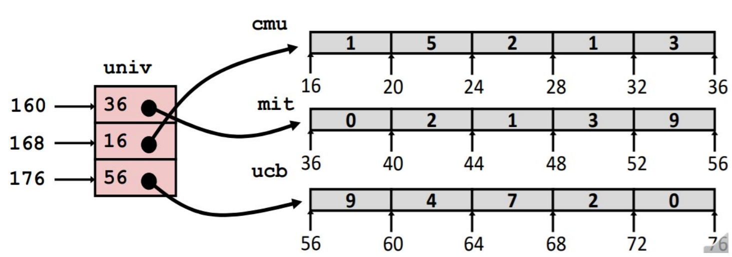
```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

Arrays: Multi-Level

- Arrays of Pre-Existing Arrays
  - Array of type T*, holds addresses of arrays
- Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```



- Element Access
  - Treat like 2D array

```
int get_univ_digit
  (size_t index, size_t digit)
{
  return univ[index][digit];
}
```

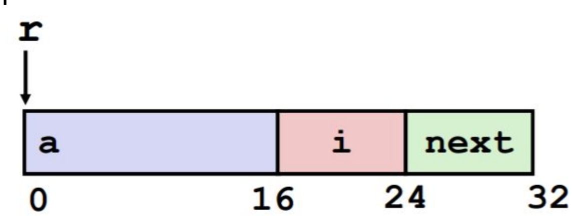
```
salq    $2, %rsi            # 4*digit
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax        # return *p
ret
```

- Computation
  - Address is `Mem[Mem[univ+8*index]+4*digit]`
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

Structures: Allocation

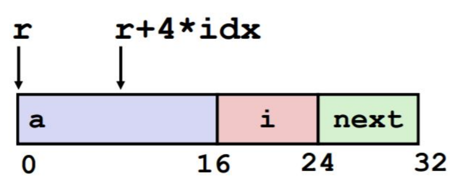
- Structure Representation
  - Represented as block of memory
  - Fields ordered according to declaration
  - Compiler determines overall size & position of fields

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



Structures: Access

- Generating Pointer to Struct Member
  - Array Element
    - Offset of each structure member determined at compile time
    - Compute as `r + 4*idx`

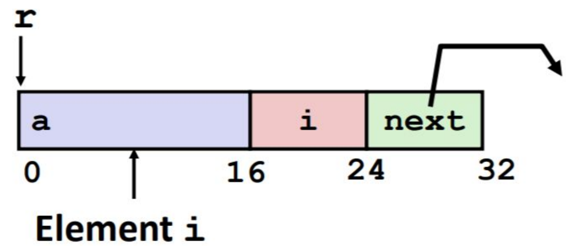



```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

```
# r in %rdi, idx in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

- Following Linked List

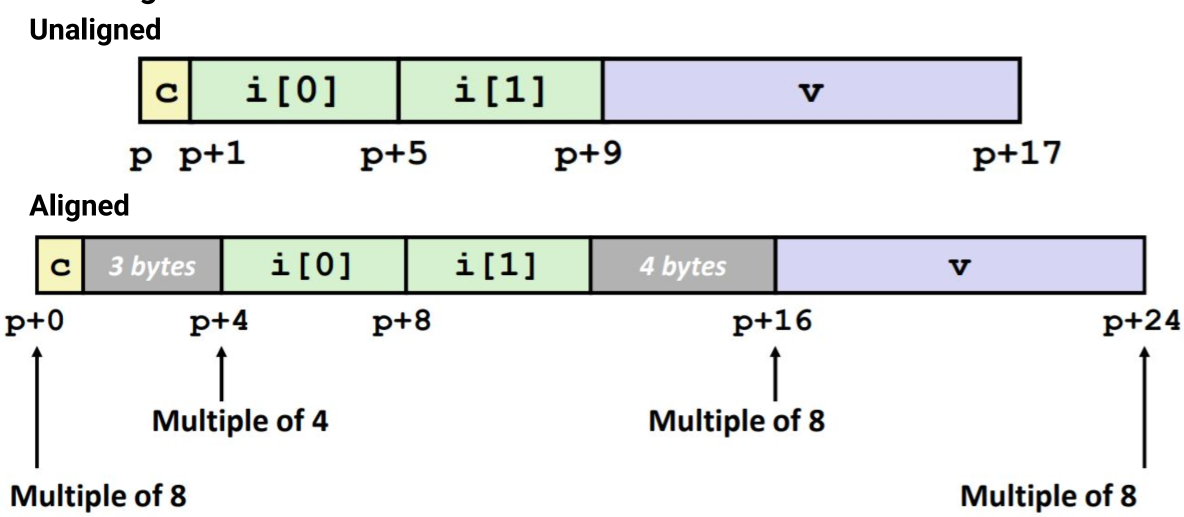
```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	r
%rsi	val

```
.L11:                           # loop:
  movslq  16(%rdi), %rax        #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)   #   M[r+4*i] = val
  movq    24(%rdi), %rdi        #   r = M[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

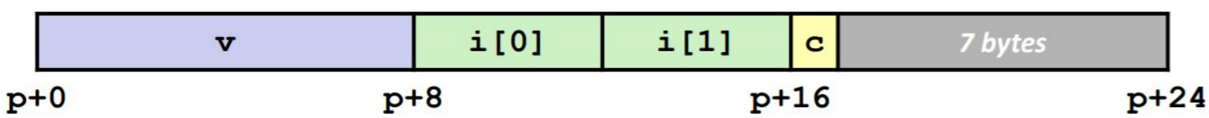
Structures: Alignment



- Alignment Principles
  - Aligned Data
    - Primitive data type requires K bytes
    - Address must be multiple of K
    - Required on some machines; advised on x86-64
  - Motivation for Aligning
    - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
      - Inefficient to load or store data that spans quad word boundaries
      - Virtual memory trickier when data spans 2 pages
  - Compiler
    - Inserts gaps in struct to ensure correct alignment of fields

- Specific Alignment Cases
  - 1 byte: char
    - No restrictions
  - 2 bytes: short
    - Lowest bit must be 0
  - 4 bytes: int, float
    - Lowest 2 bits must be 0
  - 8 bytes: double, long, pointer
    - Lowest 3 bits must be 0
  - 16 bytes: long double
    - Lowest 4 bits must be 0
- Satisfying Alignment with Structures
  - Within Structure
    - Must satisfy each element's alignment requirement
  - Overall Structure Placement
    - Each struct has alignment requirement **K**
      - **K** = largest alignment of any element
    - Initial address & struct length must be multiples of **K**

Example



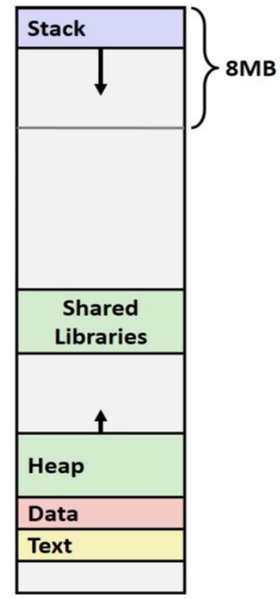
- Arrays of Structures
  - Overall structure length multiple of **K**
  - Satisfy alignment requirement for every element
- Accessing Array Elements
  - Compute array offset `sizeof(total struct)*i`
  - Add offset, will be offset `sizeof(data type)*j`
- Save Space by Putting Largest Data Types First

SECTION 3: MACHINE LEVEL PROGRAMMING

PART 5: ADVANCED TOPICS

Memory Layout

- x86-64
    - Stack
        - Runtime Stack (8MB limit)
        - ie. local variables
    - Heap
        - Dynamically allocated memory as needed
        - malloc(), calloc(), new()
    - Data
        - Statically allocated data
        - ie. global vars, static vars, string constants
    - Text/Shared Libraries
        - Executable machine instructions
        - Read-Only

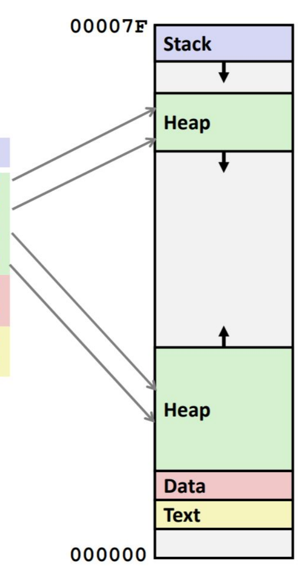


- Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8)
 /* Some print statement
}
```

address range ~2^{47}

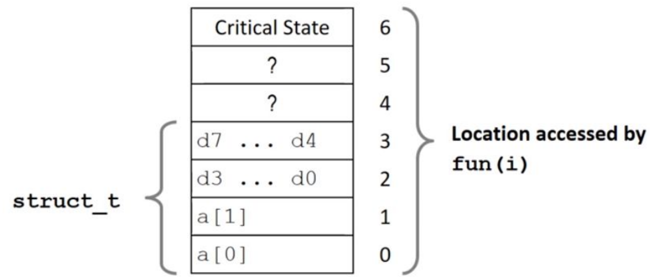
local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



Buffer Overflow: Vulnerability

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  ▯    3.14
fun(1)  ▯    3.14
fun(2)  ▯    3.1399998664856
fun(3)  ▯    2.00000061035156
fun(4)  ▯    3.14
fun(6)  ▯    Segmentation fault
```



- These are a Big Deal
  - "Buffer Overflow"
    - Exceeding memory size allocated for an array
  - Why is it a big deal?
    - #1 technical cause of security vulnerabilities
  - Most common form
    - Unchecked lengths on string inputs
- String Library Code
  - Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

    - No way to specify limit on number of chars to read
  - Similar with other library functions

- Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

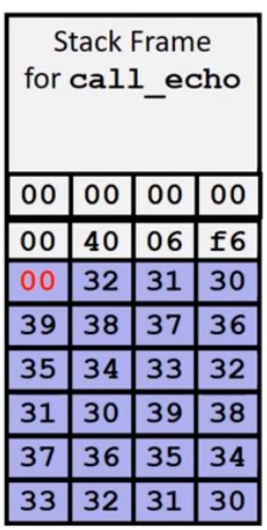
```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

echo:

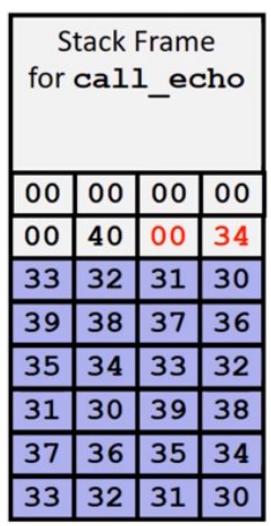
```
00000000004006cf <echo>:
 4006cf:   48 83 ec 18            sub    $0x18,%rsp
 4006d3:   48 89 e7               mov    %rsp,%rdi
 4006d6:   e8 a5 ff ff ff         callq  400680 <gets>
 4006db:   48 89 e7               mov    %rsp,%rdi
 4006de:   e8 3d fe ff ff         callq  400520 <puts@plt>
 4006e3:   48 83 c4 18            add    $0x18,%rsp
 4006e7:   c3                     retq
```

call_echo:

```
 4006e8:   48 83 ec 08            sub    $0x8,%rsp
 4006ec:   b8 00 00 00 00         mov    $0x0,%eax
 4006f1:   e8 d9 ff ff ff         callq  4006cf <echo>
 4006f6:   48 83 c4 08            add    $0x8,%rsp
 4006fa:   c3                     retq
```



String Fits



String Overflows, modifies return of call_echo

- Code Injection Attacks
    -

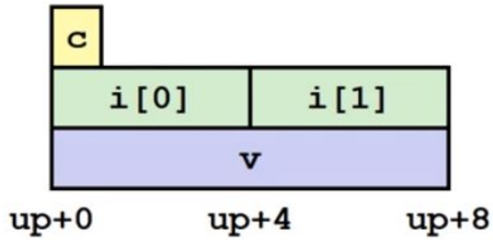
Buffer Overflow: Protection

- Binary Form

Unions

- Union Allocation
    - Allocate according to largest element
    - Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



-
    - vs a struct with the same data