Week 4 Discussion

Today’s Agenda

- Homework 2
- Lab 2
- Assembly Instructions
    - Arithmetic & Logical
    - Control
        - If statements
        - Loops
        - Switch statements
    - Procedure Calls
- Arrays, Structs, & Unions
- Stack Corruption

Homework 2

Any questions?

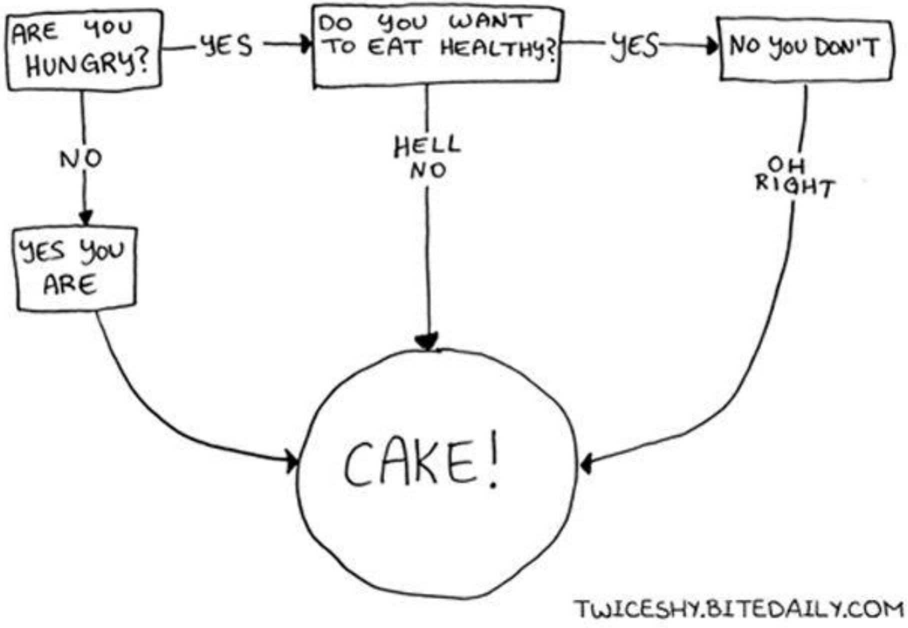
Lab 2

Any questions?

Arithmetic and Logical Operations

leal	memory,	register	load effective address
inc	register or memory		increment
dec	register or memory		decrement
neg	register or memory		negate
not	register or memory		complement
add	memory or register,	register	add
sub	memory or register,	register	subtract
imul	memory or register,	register	integer multiply
idiv	memory or register		integer divide (divides RDX:RAX by source)
xor	memory or register,	register	bitwise exclusive or
or	memory or register,	register	bitwise or
and	memory or register,	register	bitwise and
sal	immediate or one byte register,	memory or register	left arithmetic shift
shl	immediate or one byte register,	memory or register	left logical shift (sal)
sar	immediate or one byte register,	memory or register	right arithmetic shift
shr	immediate or one byte register,	memory or register	right logical shift

Control


ARE YOU HUNGRY?
YES
DO YOU WANT TO EAT HEALTHY?
YES
NO YOU DON'T
NO
HELL NO
OH RIGHT
YES YOU ARE
CAKE!
TWICESHY.BITEDAILY.COM

Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)
- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: `addl/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)`
-

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpl/cmpq` *Src2*, *Src1*
  - `cmpl b,a` like computing `a-b` without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **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)`

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - **`testl`/`testq`** *Src2*, *Src1*

    `testl b,a` like computing **`a&b`** without setting destination

  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask

  - **ZF set** when **`a&b == 0`**
  - **SF set** when **`a&b < 0`**

Reading Condition Codes

SetX Instructions

- Set low-order byte to 0 or 1 based on combinations of condition codes
- Does not alter remaining 3 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)

Jumping

jX Instructions

- Jump to different 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)

Conditional Branch Example (If Statement)

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

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Setup

Before if

if x > y

else

Finish

Conditional Move

cmov

Added to avoid branch prediction errors. Move only if true

Instruction	Synonym		
cmove	cmovz	ZF	Equal / zero
cmovne	cmovnz	~ZF	Not equal / not zero
cmovs		SF	Negative
cmovns		~SF	Nonnegative
cmovg	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	cmovnge	SF ^ OF	Less (signed <)
cmovle	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb	cmovnae	CF	Below (unsigned <)
cmovbe	cmovna	CF |	ZF below or equal (unsigned <=)

Carefully read pages 208-212.

Conditional Assignments

Compilation

```
int absdiff( int x, int y )
  {
  return x < y ? y-x : x-y ;
  }

  400493 <+10>:   mov   -0x4(%rbp),%eax
  400496 <+13>:   cmp   -0x8(%rbp),%eax          // x:y
  400499 <+16>:   jge   0x4004a9 <absdiff+32>
  40049b <+18>:   mov   -0x4(%rbp),%eax          // true: x
  40049e <+21>:   mov   -0x8(%rbp),%edx          // y
  4004a1 <+24>:   mov   %edx,%ecx
  4004a3 <+26>:   sub   %eax,%ecx
  4004a5 <+28>:   mov   %ecx,%eax
  4004a7 <+30>:   jmp   0x4004b5 <absdiff+44>
  4004a9 <+32>:   mov   -0x8(%rbp),%eax          // false: y
  4004ac <+35>:   mov   -0x4(%rbp),%edx          // x
  4004af  <+38>:  mov   %edx,%ecx
  4004b1 <+40>:   sub   %eax,%ecx
  4004b3 <+42>:   mov   %ecx,%eax
  4004b5 <+44>:   leaveq                      // done
  4004b6 <+45>:   retq
```

Unoptimized Conditional Branching

```
-O
  400474 <+0>:    mov   %esi,%eax
  400476 <+2>:    sub   %edi,%eax    // x-y
  400478 <+4>:    mov   %edi,%edx
  40047a <+6>:    sub   %esi,%edx    // y-x
  40047c <+8>:    cmp   %esi,%edi    // compare x:y
  40047e <+10>:   cmovge %edx,%eax
  400481 <+13>:   retq
```

Optimized Conditional Move

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Loops
Loops
Loops
Loops
Loops
Loops
Loops
Loops
Loops
Loops

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

"Do-While" Loop Example

C Code

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

Goto Version

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

- Count number of 1's in argument x ("popcount")
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

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

```
    movl $0, %ecx       #   result = 0
.L2:                    # loop:
    movl %edx, %eax
    andl $1, %eax       #   t = x & 1
    addl %eax, %ecx     #   result += t
    shrl %edx           #   x >>= 1
    jne  .L2            #   If !0, goto loop
```

General "While" Translation

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" Loop Form

General Form

for (*Init*; *Test*; *Update*)

Body

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```

"For" Loop → While Loop

For Version

```
for (Init; Test; Update)

    Body
```

While Version

```
Init;

while (Test) {

    Body

    Update;

}
```

"For" Loop → … → Goto

For Version

```
for (Init; Test; Update )

    Body
```

While Version

```
Init;

while (Test ) {

    Body

    Update;

}
```

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

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

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

Switch Statement Example

- **Multiple case labels**
  - 5 & 6
- **Fall through cases**
  - 2
- **Missing cases**
  - 4

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Approximate Translation

```
target = JTab[x];
goto *target;
```

Jump Table

jtab:

Targ0
Targ1
Targ2
•
•
•
Targ*n*-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targ*n*-1: Code Block *n*–1

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump Table

```
.section .rodata
    .align 4
.L4:
    .long    .L8  # x = 0
    .long    .L3  # x = 1
    .long    .L5  # x = 2
    .long    .L9  # x = 3
    .long    .L8  # x = 4
    .long    .L7  # x = 5
    .long    .L7  # x = 6
```

Setup:

```
switch_eg:
    pushl   %ebp# Setup
    movl %esp, %ebp   # Setup
    movl 8(%ebp), %eax     # eax = x
    cmpl $6, %eax # Compare x:6
    ja   .L8       # If unsigned > goto default
    jmp  *.L4(,%eax,4)     # Goto *JTab[x]
```

Indirect jump

Assembly Setup Explanation

Table Structure

- Each target requires 4 bytes
- Base address at **`.L4`**

Jumping

- **Direct:** **`jmp .L2`**
- Jump target is denoted by label **`.L2`**

- **Indirect:** **`jmp *.L4(,%eax,4)`**
- Start of jump table: **`.L4`**
- Must scale by factor of 4 (labels have 32-bits = 4 Bytes on IA32)
- Fetch target from effective Address **`.L4 + eax*4`**
  - Only for $0 \le \mathbf{x} \le 6$

Jump Table

```
.section .rodata
     .align 4
.L4:
     .long     .L8  # x = 0
     .long     .L3  # x = 1
     .long     .L5  # x = 2
     .long     .L9  # x = 3
     .long     .L8  # x = 4
     .long     .L7  # x = 5
     .long     .L7  # x = 6
```

Jump Table

Jump table

```
.section .rodata
    .align 4
.L4:
    .long    .L8 # x = 0
    .long    .L3 # x = 1
    .long    .L5 # x = 2
    .long    .L9 # x = 3
    .long    .L8 # x = 4
    .long    .L7 # x = 5
    .long    .L7 # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

Code Blocks (x == 1)

```
switch(x) {
case 1:     // .L3
      w = y*z;
      break;
 . . .
}
```

```
.L3:                           # x == 1
   movl   12(%ebp), %eax       # y
   imull  16(%ebp), %eax       # w = y*z
   jmp .L2                     # Goto done
```

Code Blocks (x == 2, x == 3)

```
long w = 1;
. . .
switch(x) {
. . .
case 2:
   w = y/z;
   /* Fall Through */
case 3:
   w += z;
   break;
. . .
}
```

```
.L5:                           # x == 2
  movl    12(%ebp), %eax # y
  cltd
  idivl   16(%ebp)             # y/z
  jmp     .L6                  # goto merge
.L9:                           # x == 3
  movl    $1, %eax             # w = 1
.L6:                           # merge:
  addl    16(%ebp), %eax # += z
  jmp     .L2                  # goto done
```

Code Blocks (x == 5, x == 6, default)

```
switch(x) {
  . . .
  case 5:  // .L7
  case 6:  // .L7
      w -= z;
      break;
  default: // .L8
      w = 2;
}
```

```
.L7:                              # x == 5, 6
  movl     $1, %eax               # w = 1
  subl     16(%ebp), %eax         # w -= z
  jmp      .L2                    # goto done
.L8:                              # default
  movl     $2, %eax               # w = 2
.L2:                              # done:
```

Switch Code (Finish)

```
return w;
```

```
.L2:          # done:
    popl    %ebp
    ret
```

- **Noteworthy Features**
  - Jump table avoids sequencing through cases
    - Constant time, rather than linear
  - Use jump table to handle holes and duplicate tags
  - Use program sequencing and/or jumps to handle fall-through
  - Don’t initialize w = 1 unless really need it

Procedure Calls

- **What happens when I make a function call?**
- How do we pass parameters?
- What executes after the return?
- Where does the return value go?
- How do we keep track of registers?

Register Saving Conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the ***caller***
  - `who` is the ***callee***

- **Can register be used for temporary storage?**

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $18213, %edx
    • • •
    ret
```

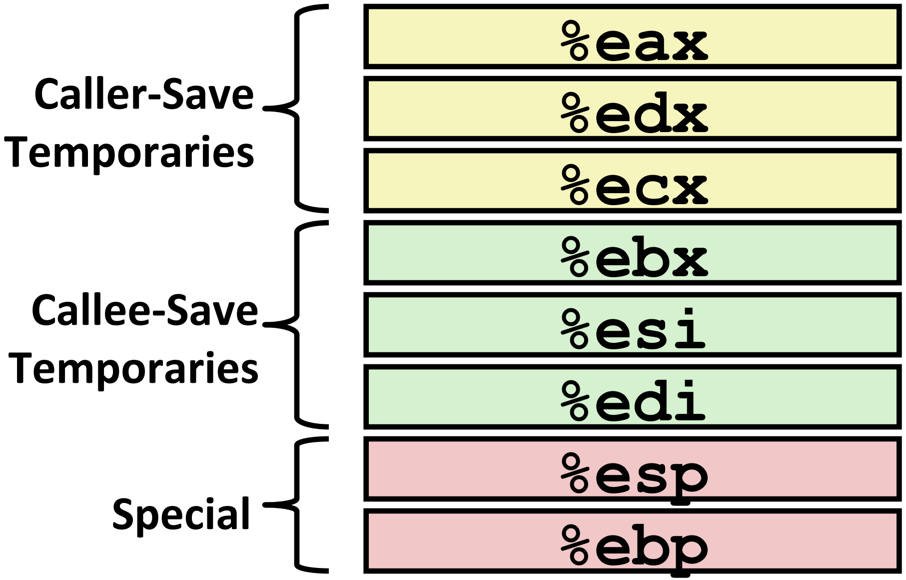
  - Contents of register `%edx` overwritten by `who`
  - This could be trouble ➙ something should be done!
    - Need some coordination

Register Saving Conventions

- **When procedure `yoo` calls `who`:**
  - `yoo` is the ***caller***
  - `who` is the ***callee***
- **Can register be used for temporary storage?**
- **Conventions**
  - ***"Caller Save"***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Save"***
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

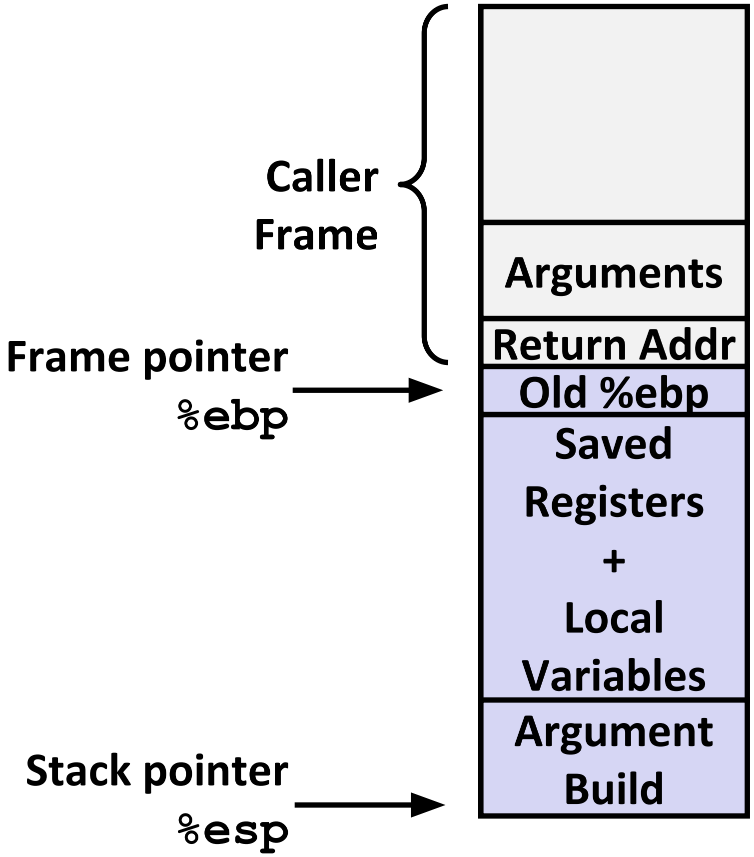
IA32/Linux+Windows Register Usage

- `%eax, %edx, %ecx`
  - Caller saves prior to call if values are used after call returns
- `%eax`
  - also used to return integer value
- `%ebx, %esi, %edi`
  - Callee saves if wants to use them
- `%esp, %ebp`
  - special form of callee save
  - Restored to original values upon exit from procedure



IA32/Linux Stack Frame

- **Caller Stack Frame**
  - Arguments for this call
  - Return address
    - Pushed by `call` instruction
- **Callee Stack Frame**
  - Old frame pointer
  - Saved register context
  - Local variables
    If can't keep in registers
  - "Argument build:"
    Parameters for next callee



Procedure Calls (Example 1)

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Compile:
  gcc -c example1.c -g

Disassemble obj file:
  objdump -d example1.o > example1.s

Build executable:
  gcc -o example1 example1.o

Generated IA32 Assembly

```
0000004c <sum>:
```

Set Up

```
4c:     pushl %ebp
4d:     movl %esp,%ebp
```

Body

```
4f:     movl 0x0c(%ebp),%eax
52:     addl 0x08(%ebp),%eax
```

Finish

```
55:     popl %ebp
57:     ret
```

Procedure Calls (Example 1)

Generated IA32 Assembly

```
0000004c <sum>:
4c:     pushl %ebp
4d:     movl %esp,%ebp

4f:     movl 0x0c(%ebp),%eax
52:     addl 0x08(%ebp),%eax

55:     popl %ebp
57:     ret
```

Save base pointer of the caller frame onto the stack

%eip = 4c

Procedure Calls (Example 1)

Generated IA32 Assembly

```
0000004c <sum>:
4c:      pushl %ebp
4d:      movl %esp,%ebp

4f:      movl 0x0c(%ebp),%eax
52:      addl 0x08(%ebp),%eax

55:      popl %ebp
57:      ret
```

Update base pointer to point to the current frame (sum)

%eip = 4d

Procedure Calls (Example 1)

Generated IA32 Assembly

```
0000004c <sum>:
4c:      pushl %ebp
4d:      movl %esp,%ebp

4f:      movl 0x0c(%ebp),%eax
52:      addl 0x08(%ebp),%eax

55:      popl %ebp
57:      ret
```

Grab the argument 2
%eax = argument 2

%eip = 4f

Procedure Calls (Example 1)

Generated IA32 Assembly

```
0000004c <sum>:
4c:      pushl %ebp
4d:      movl %esp,%ebp

4f:      movl 0x0c(%ebp),%eax
52:      addl 0x08(%ebp),%eax

55:      popl %ebp
57:      ret
```

Grab the argument 2
%eax = argument 2
%eax += argument 1

%eip = 52

Procedure Calls (Example 1)

Generated IA32 Assembly

```
0000004c <sum>:
4c:     pushl %ebp
4d:     movl %esp,%ebp

4f:     movl 0x0c(%ebp),%eax
52:     addl 0x08(%ebp),%eax

55:     popl %ebp
57:     ret
```

Restore the caller frame

%eip = 55

Procedure Calls (Example 1)

Generated IA32 Assembly

```
0000004c <sum>:
4c:      pushl %ebp
4d:      movl %esp,%ebp

4f:      movl 0x0c(%ebp),%eax
52:      addl 0x08(%ebp),%eax

55:      popl %ebp
57:      ret
```

Return to the caller's next instruction:
%eip = 0x04(%ebp) + $0x05

%eip = 57

Some Remarks

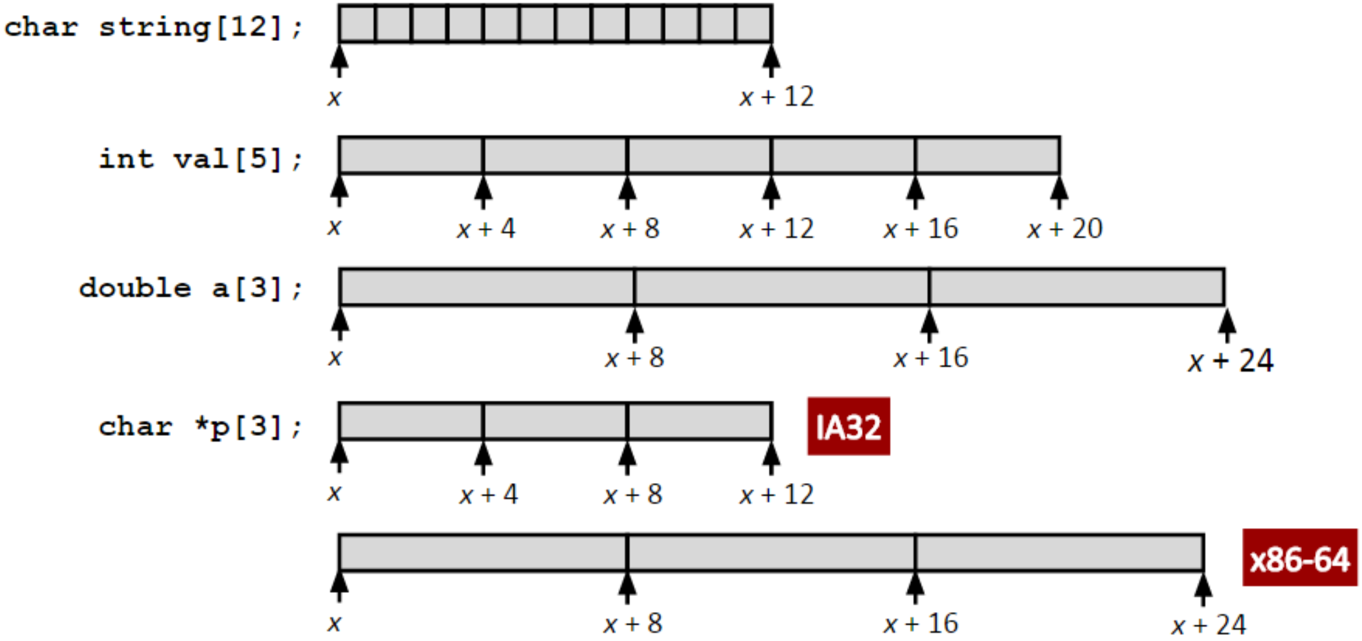
- Why does this code not allocate stack space?
- How does the caller get the return value?

Carnegie Mellon

Array Allocation

- Basic Principle

  T **A[*L*];**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory



Nested Array 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];

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]"**
  - Variable **pgh**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **int**'s, allocated contiguously
- **"Row-Major" ordering of all elements in memory**

Many Variations

```
struct newtype2
{
  int a ;
  struct inner
    {
    float b ;
    int c[10] ;
    } y ;
  int *d ;
} x ;
```

The scope of the variable name lies inside of the structure. That is the name is not known outside of the structure unless you refer to the variable with its qualification:

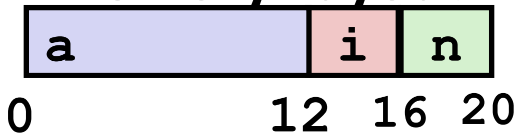
```
x.a,   x.y.c[5]
```

You cannot refer to c[5] without the qualification unless char c[] exists outside of the structure. This means that you can have c both inside and outside of the structure. Confusing!

Structure Allocation

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```

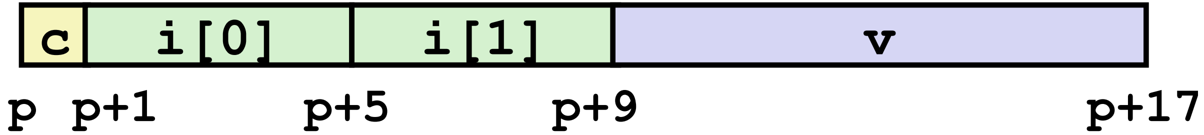
Memory Layout



- **Concept of structures in C**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

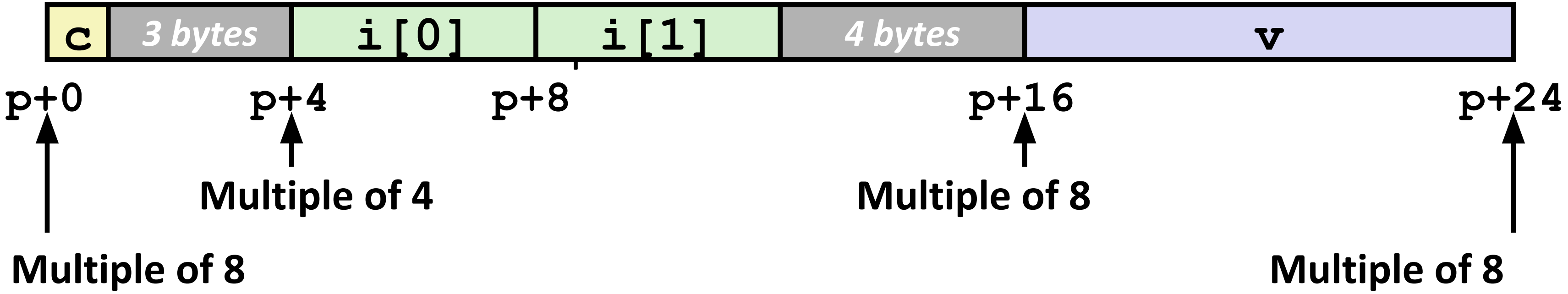
Structures & Alignment

- **Unaligned Data**




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- **Aligned Data**
  - Primitive data type requires *K* bytes



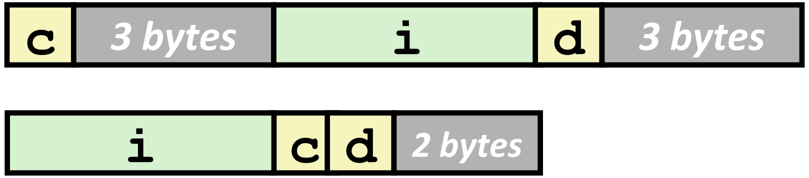
Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)



Unions

Look like structures but..

In unions, the offset is always 0. This means that each variable overlays or occupies the same storage as the other variables:

```
union u
  {
   int i ;
   unsigned char c[4] ;
   float a ;
   } examine_endian ;
```

Sound familiar? Pointers are not needed here! But it is dangerous.

Accessing examine_endian.a overwrites what is in examine_endian.i

Memory Corruption

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

Vulnerable Buffer Code

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

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

```
unix>./bufdemo
Type a string:0123456789a
0123456789a
```

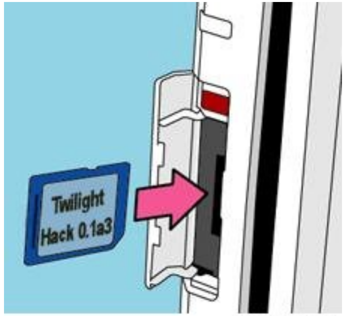
```
unix>./bufdemo
Type a string:0123456789ab
Segmentation Fault
```

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines***
- **Distressingly common in real progams**
  - Programmers keep making the same mistakes ☹
- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” between MSN and AOL (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more

Twilight hack on Wii

- Legend of Zelda: Twilight Princess hack on Nintendo Wii
- The game let's you give Epona (Link's horse) a custom name, which manually only allows a certain number of characters, but when loading from a file, has no restrictions.
- A specially crafted save file has a small program just outside of the horse name buffer, which happens to be where the next line of code should execute.
- You have full control at this point to run whatever you have on the SD card.



gbatemp.net

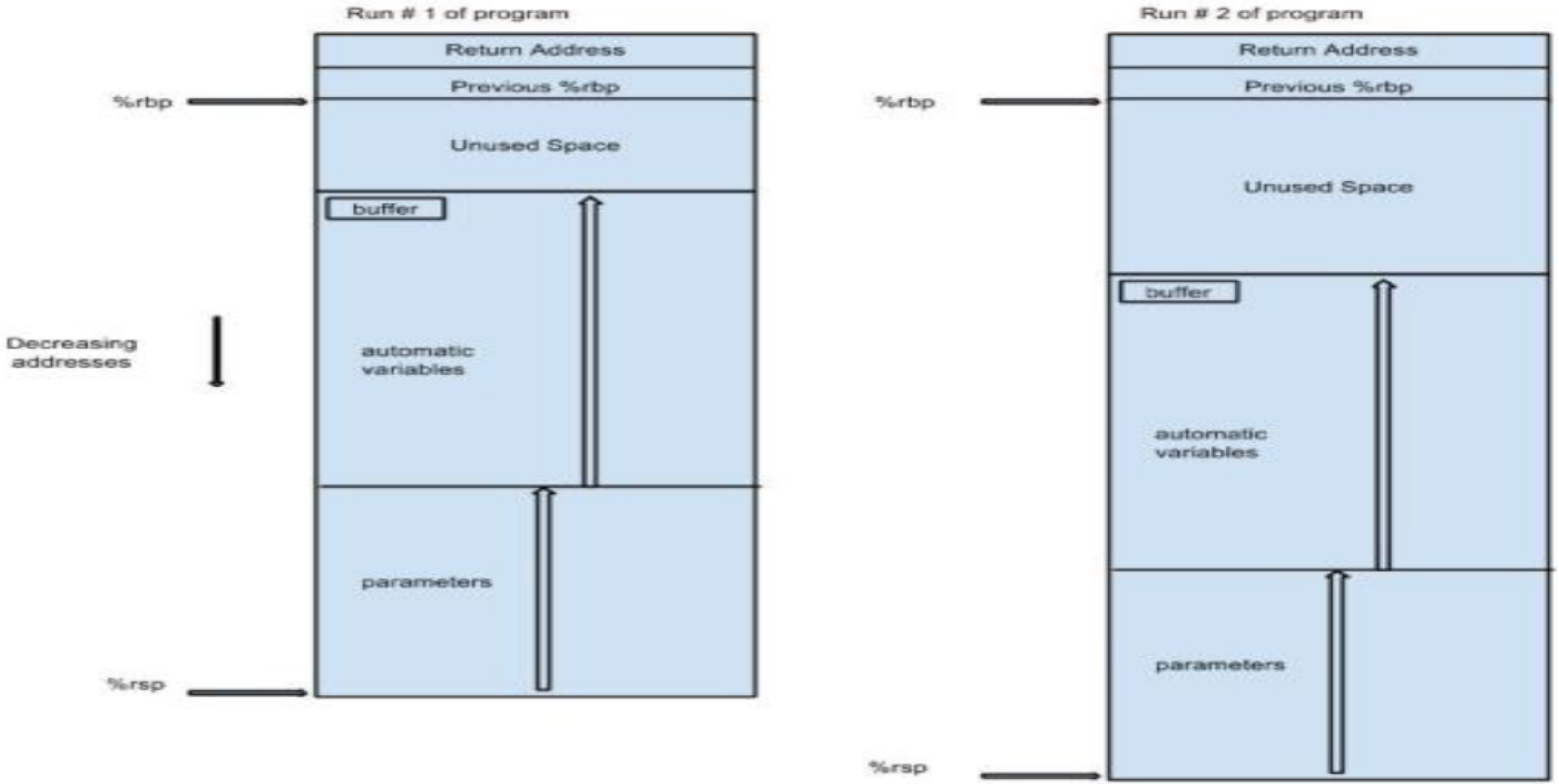
Avoid Overflow Vulnerabilities in Code!

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

- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don’t use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string

Memory Corruption

Sample Stack Randomization



Other System-level Protection

- Corruption detection: Store a random value somewhere in stack at the beginning of the program. Store that value in a protected area of memory. At the end of the program compare the values. If changed, raise the red flag.

- Hardware which prevents pages from executing code. Memory is divided into 2K or 4K byte “pages”. Each page can be set with read/write/execute bits when in supervisory mode.

Thanks!