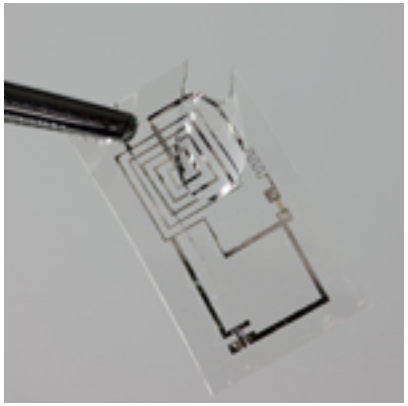
Researchers at the University of Illinois, in collaboration with Tufts University and Northwestern University, have demonstrated a new type of biodegradable electronics technology that could introduce new design paradigms for medical implants, environmental monitors and consumer devices.



Three application areas appear particularly promising. First are medical implants that perform important diagnostic or therapeutic functions for a useful amount of time and then simply dissolve and resorb in the body. Second are environmental monitors, such as wireless sensors that are dispersed after a chemical spill, that degrade over time to eliminate any ecological impact. Third are consumer electronic systems or sub-components that are compostable, to reduce electronic waste streams generated by devices that are frequently upgraded, such as cellphones or other portable devices.

http://news.illinois.edu/news/12/0927transient_electronics_JohnRogers.html

- Questions about lab?

```
#include <stdio.h>

int globalvar;

int  func1(int *x);
int  func2(int  x, int *y);
void func3_hint_seepage215(int x);

main() {
  int mainvar = 0;
  int ret =0;

  globalvar = 364;

  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  ret = func1(&mainvar);
  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  ret = func2(mainvar, &globalvar);
  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  func3_hint_seepage215(ret);
}
```

```
int func1(int *x) {

  *x = 1 + 364;

  return 0;
}

int func2(int x, int *y) {

  return x * *y;
}
```

```
void func3_hint_seepage215(int x) {
  int val = 2;

  switch(x) {
    case 1: {
      val = x + 1;
      break;
    }
    case 2: {
      val = x / 10;
      break;
    }
    case 3: {
      val = x * 10;
      break;
    }
    case 4: {
      val = x % 10;
      break;
    }
    case 128: {
      val = 3;
      break;
    }
    default: {val = 0; }
  }
  printf("Something %d\n", val);
}
```

"For" Loop → … → Goto

For Version

```
for (Init; Test; Update )
    Body
```

While Version

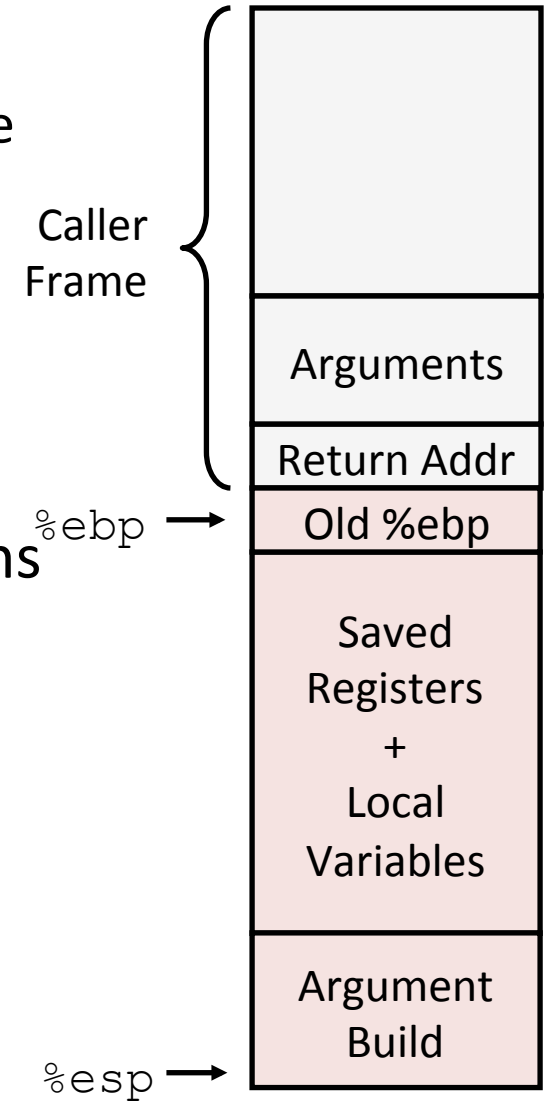
```
Init;
while (Test ) {
    Body
    Update;
}
```

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

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

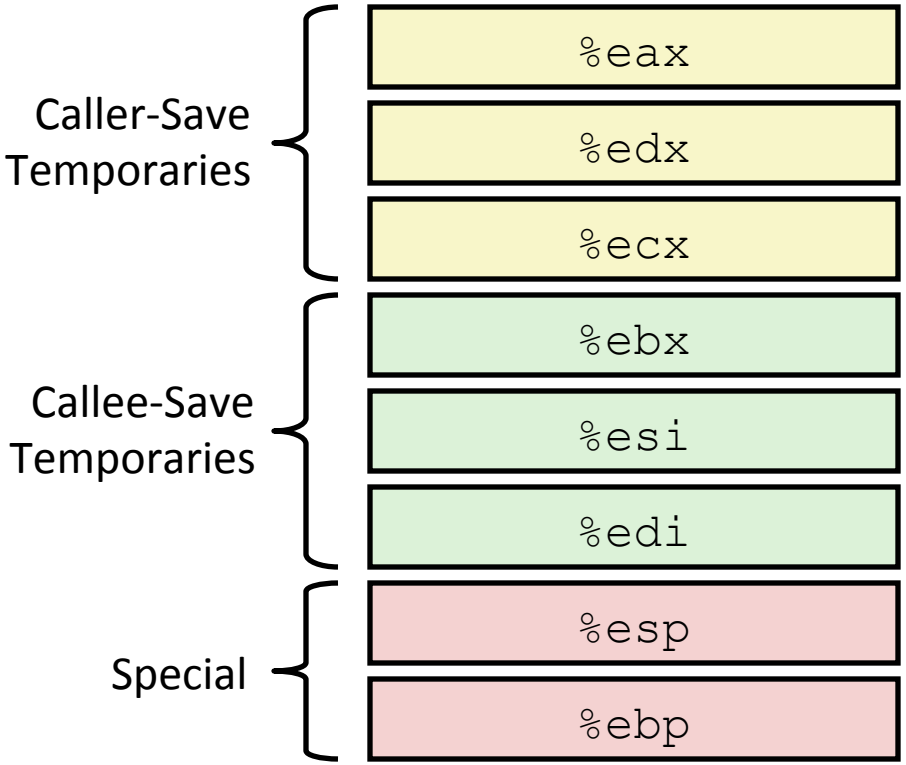
IA 32 Procedure Summary

- The Stack Makes Recursion Work
  - Private storage for each *instance* of procedure call
    - Instantiations don't clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Managed by stack discipline
    - Procedures return in inverse order of calls
- IA32 Procedures Combination of Instructions + Conventions
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - **`%ebp`** and **`%esp`**
  - Stack frame organization conventions



IA32/Linux Register Usage

- `%eax, %edx, %ecx`
  - Caller saves prior to call if values are used later
- `%eax`
  - also used to return integer value
- `%ebx, %esi, %edi`
  - Callee saves if wants to use them
- `%esp, %ebp`
  - special



A C function `fun` has the following code body:

```
*p = d;
return x-c;
```

The IA32 code implementing this body is as follows:

```
movsbl  12(%ebp),%edx
movl    16(%ebp), %eax
movl    %edx, (%eax)
movswl  8(%ebp),%eax
movl    20(%ebp), %edx
subl    %eax, %edx
movl    %edx, %eax
```

Write a prototype for function `fun`, showing the types and ordering of the arguments `p`, `d`, `x`, and `c`.

One step in learning to read IA32 code is to become very familiar with the way arguments are passed on the stack. The key to solving this problem is to note that the storage of `d` at `p` is implemented by the instruction at line 3 of the assembly code, from which you work backward to determine the types and positions of arguments `d` and `p`. Similarly, the subtraction is performed at line 6, and from this you can work backward to determine the types and positions of arguments `x` and `c`.

The following is the function prototype:

```
int fun(short c, char d, int *p, int x);
```

As this example shows, reverse engineering is like solving a puzzle. It's important to identify the points where there is a unique choice, and then work around these points to fill in the rest of the details.

```
1  int proc(void)
2  {
3      int x,y;
4      scanf("%x %x", &y, &x);
5      return x-y;
6  }
```

```
1   proc:
2     pushl   %ebp
3     movl    %esp, %ebp
4     subl    $40, %esp
5     leal    -4(%ebp), %eax
6     movl    %eax, 8(%esp)
7     leal    -8(%ebp), %eax
8     movl    %eax, 4(%esp)
9     movl    $.LC0, (%esp)     Pointer to string "%x %x"
10    call    scanf
      Diagram stack frame at this point
11    movl    -4(%ebp), %eax
12    subl    -8(%ebp), %eax
13    leave
14    ret
```

Register	Value
%esp	0x800040
%ebp	0x800060

Suppose `proc` calls `scanf` (line 10), and that `scanf` reads values 0x46 and 0x53 from the standard input. Assume that the string "%x %x" is stored at memory location 0x300070.

A. What value does `%ebp` get set to on line 3?

B. What value does `%esp` get set to on line 4?

C. At what addresses are local variables `x` and `y` stored?

D. Draw a diagram of the stack frame for `proc` right after `scanf` returns. Include as much information as you can about the addresses and the contents of the stack frame elements.

E. Indicate the regions of the stack frame that are not used by `proc`.

A. We started with `%esp` having value `0x800040`. The `pushl` instruction on line 2 decrements the stack pointer by 4, giving `0x80003C`, and this becomes the new value of `%ebp`.

B. Line 4 decrements the stack pointer by 40 (hex `0x28`), yielding `0x800014`.

C. We can see how the two `leal` instructions (lines 5 and 7) compute the arguments to pass to `scanf`, while the two `movl` instructions (lines 6 and 8) store them on the stack. Since the function arguments appear on the stack at increasingly positive offsets from `%esp`, we can conclude that line 5 computes `&x`, while line 7 computes line `&y`. These have values `0x800038` and `0x800034`, respectively.

D. The stack frame has the following structure and contents:

0x80003C	0x800060	← %ebp
0x800038	0x53	x
0x800034	0x46	y
0x800030		
0x80002C		
0x800028		
0x800024		
0x800020		
0x80001C	0x800038	
0x800018	0x800034	
0x800014	0x300070	← %esp

E. Byte addresses `0x800020` through `0x800033` are unused.

Basic Data Types

- Integral
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used

Intel	suffix	Bytes	C
byte	**b**	1	**[unsigned] char**
word	**w**	2	**[unsigned] short**
double word	**l**	4	**[unsigned] int**
quad word	**q**	8	**[unsigned] long int** (x86-64)

- Floating Point
  - Stored & operated on in floating point registers

Intel	suffix	Bytes	C
Single	**s**	4	**float**
Double	**l**	8	**double**
Extended	**t**	10/12/16	**long double**

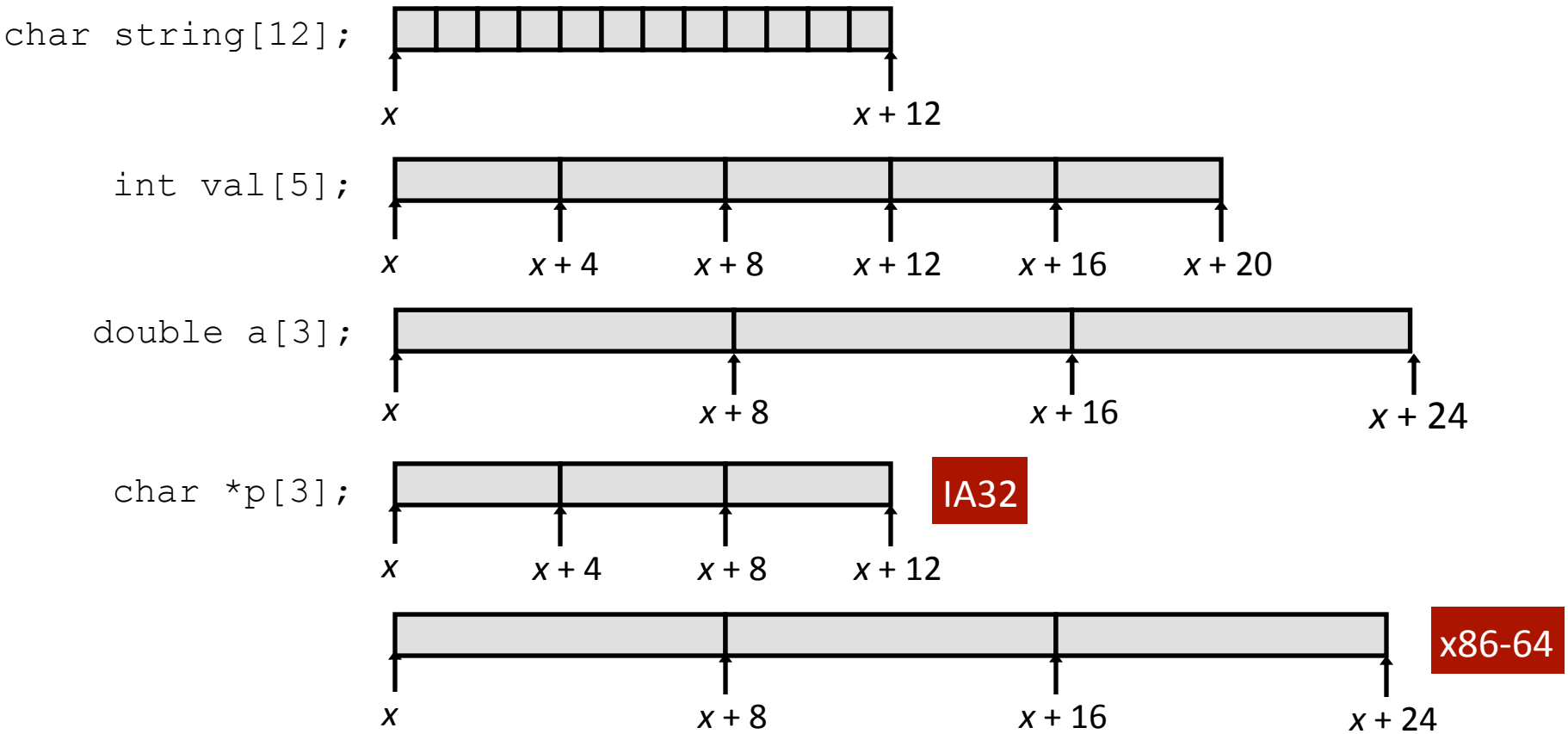
Array Allocation

- Basic Principle

  T **A[***L***];**

  – Array of data type *T* and length *L*

  – Contiguously allocated region of *L* * **sizeof** (*T*) bytes



Array Access

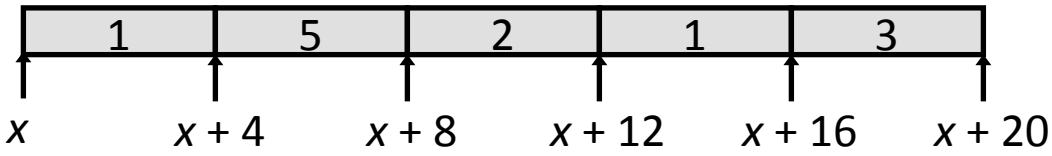
- Basic Principle

  T **A[***L***]** *;*

  – Array of data type *T* and length *L*

  – Identifier **A** can be used as a pointer to array element 0: Type *T**

```
int val[5];
```



- Reference Type Value

Reference	Type	Value
val[4]		
val		
val+1		
&val[2]		
val[5]		
***(val+1)**		
val + *i*		

Array Access

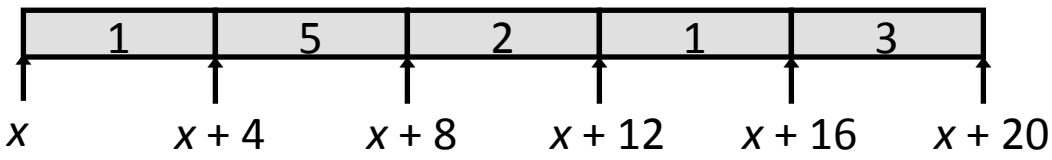
- Basic Principle

  T **A[***L***]** *;*

  – Array of data type *T* and length *L*

  – Identifier **A** can be used as a pointer to array element 0: Type *T**

```
int val[5];
```



- Reference Type Value

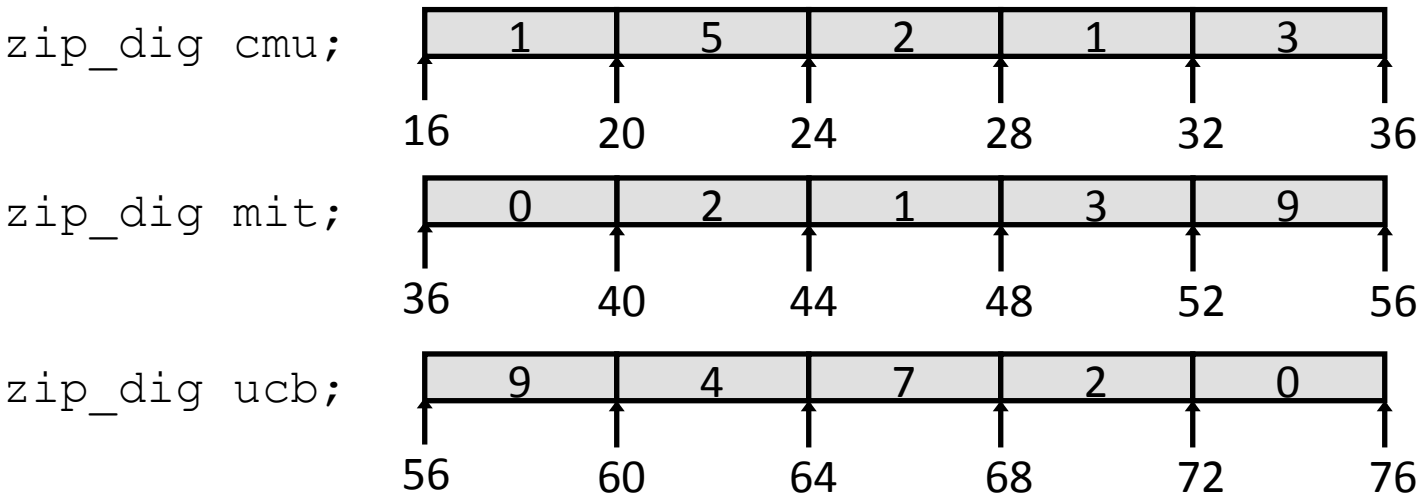
Reference	Type	Value
val[4]	**int**	3
val	**int ***	*x*
val+1	**int ***	*x* + 4
&val[2]	**int ***	*x* + 8
val[5]	**int**	??
***(val+1)**	**int**	5
val + *i*	**int ***	*x* + 4 *i*

Array Example

```
int a[5][5];

typedef int zip_dig[5];
zip_dig b[5];  //equivalent to a
```

Array Example

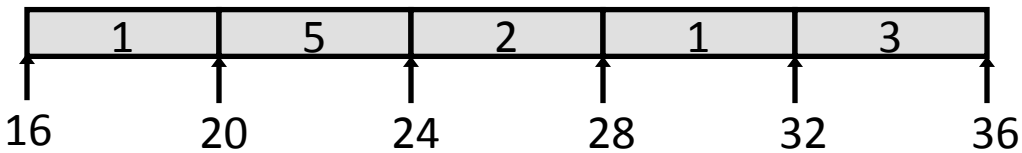
```
typedef int zip_dig[5];

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]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

`zip_dig cmu;`




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

IA32

```
  # %edx = z
  # %eax = dig
   movl (%edx,%eax,4),%eax  # z[dig]
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

```
int E[10]; //an array of integers
int i; //index into array
```

E is in %edx, i is in %ecx

Move *something* into %eax

E	int *	movl %edx, %eax
E[0]	?	movl ????, %eax
E[i]	?	movl ????, %eax
&E[2]	?	leal ????, %eax
&E[i]-E	?	movl ????, %eax

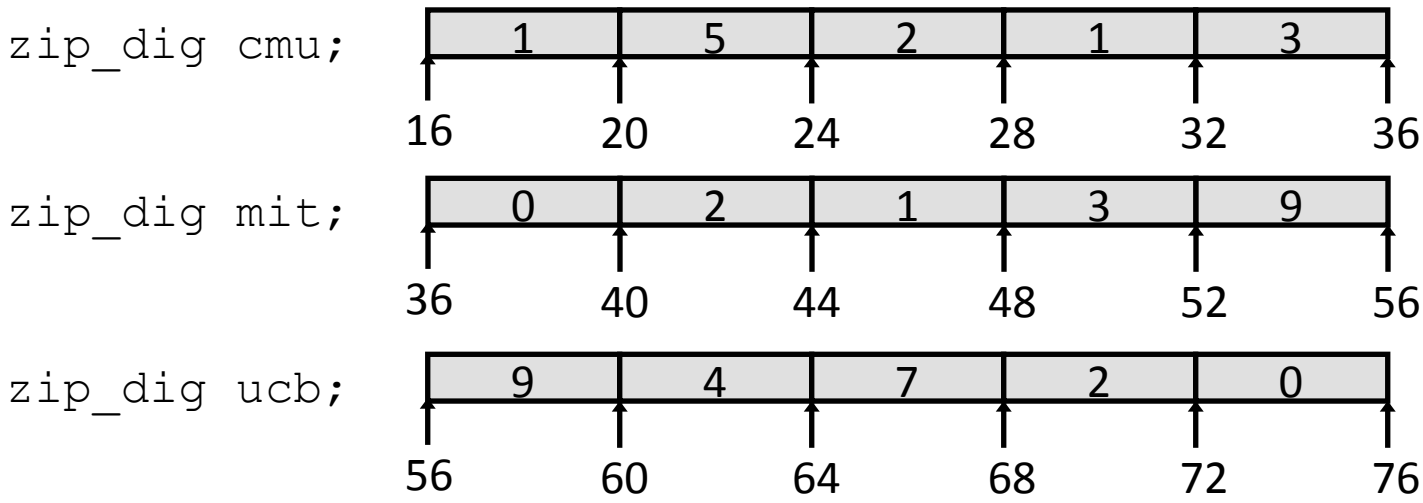
```
int E[10]; //an array of integers
int i; //index into array
```

E is in %edx, i is in %ecx

Move *something* into %eax

E	int *	movl %edx, %eax
E[0]	int	movl (%edx), %eax
E[i]	int	movl (%edx, %ecx, 4), %eax
&E[2]	int*	leal 8(%edx), %eax
&E[i]-E	int	movl %ecx, %eax

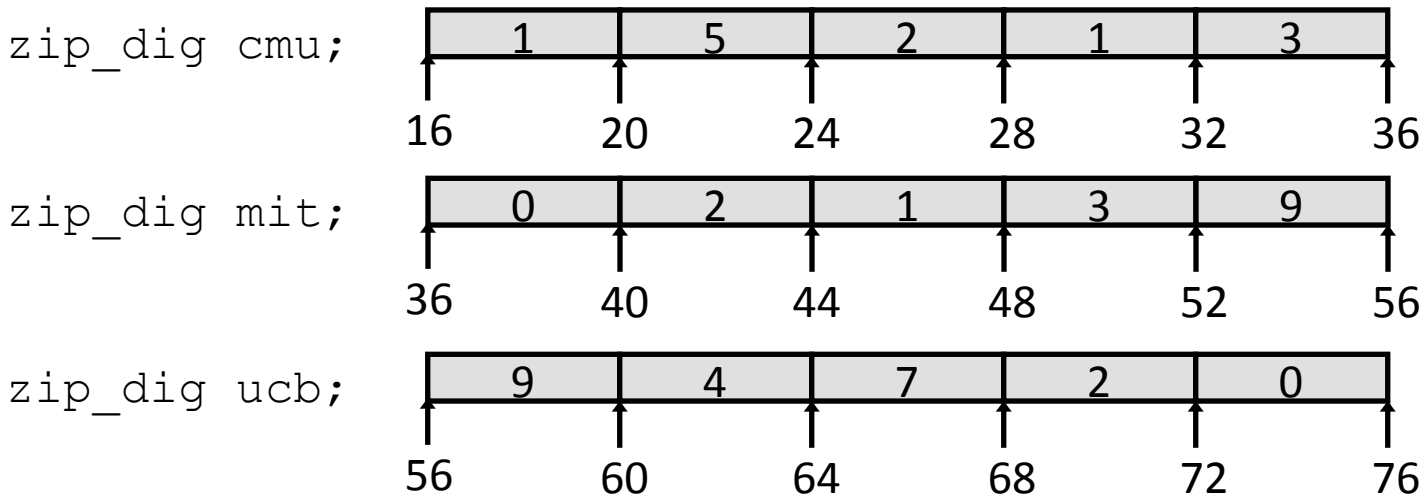
Referencing Examples



- Reference Address Value Guaranteed?

Reference	Address	Value	Guaranteed?
mit[3]	**36 + 4* 3 = 48**	**3**	**Yes**
mit[5]			
mit[-1]			
cmu[15]			

Referencing Examples



- Reference Address Value Guaranteed?

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	Yes
`mit[5]`	`36 + 4* 5 = 56`	`9`	No
`mit[-1]`	`36 + 4*-1 = 32`	`3`	No
`cmu[15]`	`16 + 4*15 = 76`	`??`	No

  - No bound checking
  - Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays

Intermission

http://cheezburger.com/42481409

Array Loop Example

- Original

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- Transformed
  - As generated by GCC
  - Eliminate loop variable `i`
  - Convert array code to pointer code
  - Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z <= zend);
  return zi;
}
```

Array Loop Implementation (IA32)

- Registers

  %ecx z

  %eax zi

  %ebx zend

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

```
  # %ecx = z
  xorl %eax,%eax
  leal 16(%ecx),%ebx
.L59:
  leal (%eax,%eax,4),%edx
  movl (%ecx),%eax
  addl $4,%ecx
  leal (%eax,%edx,2),%eax
  cmpl %ebx,%ecx
  jle .L59
```

Array Loop Implementation (IA32)

- Registers
  %ecx z
  %eax zi
  %ebx zend
- Computations
  - **10*zi + *z** implemented as
    ***z + 2*(zi+4*zi)**
  - **z++** increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

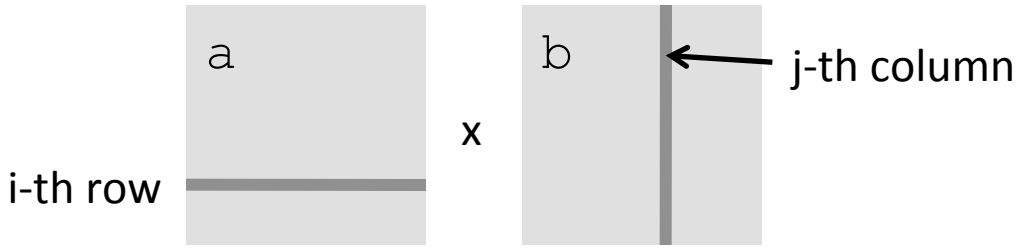
```
  # %ecx = z
  xorl %eax,%eax            # zi = 0
  leal 16(%ecx),%ebx        # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx # 5*zi
  movl (%ecx),%eax          # *z
  addl $4,%ecx              # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx            # z : zend
  jle .L59                  # if <= goto loop
```

Using Nested Arrays

- Strengths
  - C compiler handles doubly subscripted arrays
  - Generates very efficient code
- Limitation
  - Only works for fixed array size

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
  int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```



(a) Original C code

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele (fix_matrix A, fix_matrix B,  int i, int k) {
    int j;
    int result = 0;

    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

(b) Optimized C code

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k) {
    int *Arow = &A[i][0];
    int *Bptr = &B[0][k];
    int result = 0;
    int j;
    for (j = 0; j != N; j++) {
        result += Arow[j] * *Bptr;
        Bptr += N;
    }
    return result;
}
```

Dynamic Nested Arrays

- Strength
  - Can create matrix of any size
- Programming
  - Must do index computation explicitly
- Performance
  - Accessing single element costly
  - Must do multiplication

```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

```
int var_ele
  (int *a, int i, int j, int n)
{
  return a[i*n+j];
}
```

```
movl 12(%ebp),%eax        # i
movl 8(%ebp),%edx         # a
imull 20(%ebp),%eax       # n*i
addl 16(%ebp),%eax        # n*i+j
movl (%edx,%eax,4),%eax   # Mem[a+4*(i*n+j)]
```

Dynamic Array Multiplication

- Without Optimizations
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 for array indexing
    - 1 for loop index
    - 1 for data

```
/* Compute element i,k of
   variable matrix product */
int var_prod_ele
  (int *a, int *b,
   int i, int k, int n)
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}
```

Optimizing Dynamic Array Multiplication

- Optimizations
  - Performed when set optimization level to **-O2**
- Code Motion
  - Expression **i*n** can be computed outside loop
- Strength Reduction
  - Incrementing **j** has effect of incrementing **j*n+k** by **n**
- Operations count
  - 4 adds, 1 mult
- Compiler can optimize regular access patterns

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

```
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

Structures

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

Memory Layout

- Concept
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types
- Accessing Structure Member

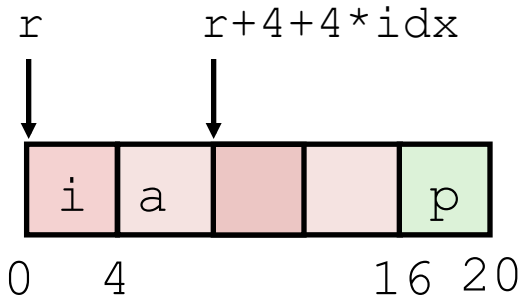
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Structure Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

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

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax   # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```