CS429: Computer Organization and Architecture
Instruction Set Architecture V

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 16, 2014 at 15:19

Basic Data Types

Integral

- Stored and operated on in general registers.
- Signed vs. unsigned depends on instructions used.

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int

Floating Point

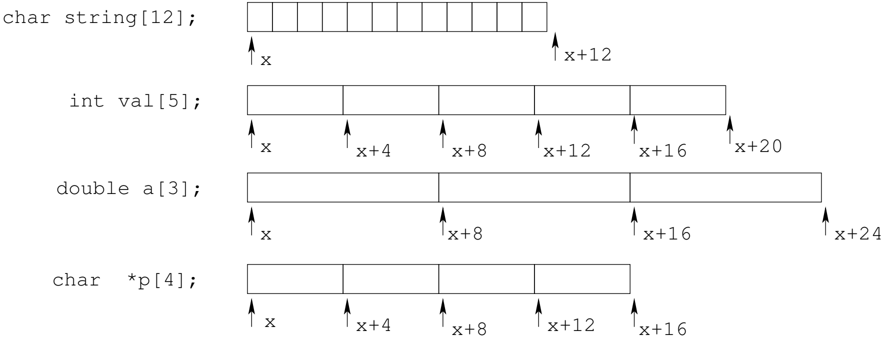
Stored and operated on in floating point registers.

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

Array Allocation

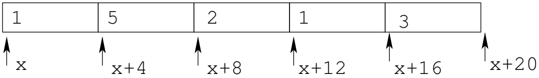
Basic Principle: `T A[L]`

- Array (named A) of data type `T` and length `L`.
- Contiguously allocated region of `L * sizeof(T)` bytes.



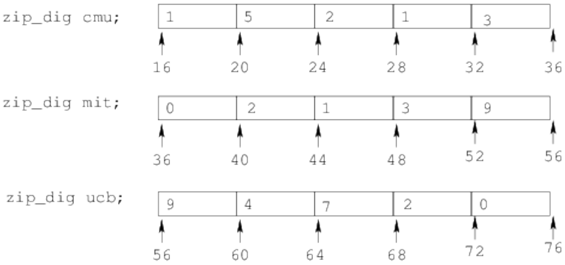
Array Access

`int val[5];`



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+j`	`int *`	$x+4j$

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` is equivalent to `int cmu[5]`.

Example arrays were allocated in successive 20 byte block.

That's not guaranteed to happen in general.

Array Accessing Example

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

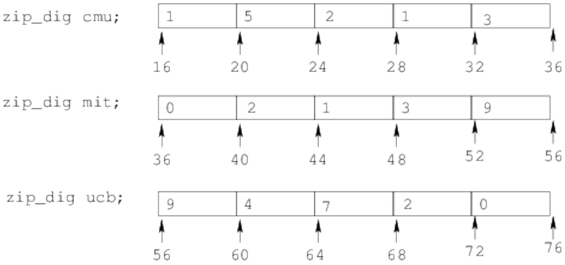
Memory Reference Code

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

Computation

- Register %edx contains the starting address of the array.
- Register %eax contains the array index.
- The desired digit is at `(4 * %eax) + %edx`.
- User memory reference `(%edx,%eax,4)`.

Referencing Examples



- Code does not do any bounds checking!
- Out of range behavior is implementation-dependent.
- There is no guaranteed relative allocation of different arrays.

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

Array Loop Example

Original Source

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

Transformed Version

- As generated by gcc.
- Eliminates loop variable `i`.
- Converts array code to pointer code.
- Expresses in do-while form.
- No need to 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

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

%ecx holds z
%eax holds zi
%ebx holds zend

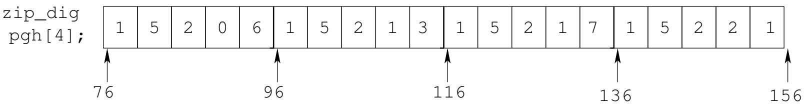
Computations

- "(10*zi + *z)" implemented as "2*(zi+4*zi) + *zi."
- z++ increments by 4.

```
  # %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            # compare z : zend
  .jle   .L59                 # if <= goto loop
```

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}};
```

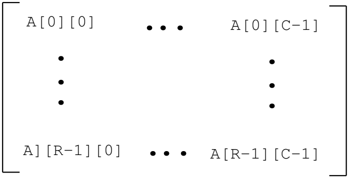


- Declaration "`zip_dig pgh[4]`" is equivalent to "`int pgh[4][5]`."
- Variable `pgh` denotes an array of 4 elements allocated contiguously.
- Each element is an array of 5 ints, which are allocated contiguously.
- This is "row-major" ordering of all elements, guaranteed.

Nested Array Allocation

Declaration: `T A[R][C]`

- Array of (element) data type T.
- R rows, C columns
- Assume type T element requires K bytes.



Array Size: `R * C * K`
Arrangement: row-major ordering

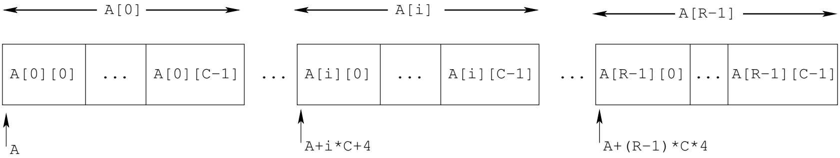
Row major order means the elements are stored in the following order:

$$[A_{0,0}, \ldots, A_{0,C-1}, A_{1,0}, \ldots, A_{1,C-1}, \ldots, A_{R-1,0}, \ldots, A_{R-1,C-1}] .$$

Nested Array Row Access

Given an nested array declaration `A[R][C]`, you can think of this as an array of arrays.

- `A[i]` is an array of C elements.
- Each element has type T.
- The starting address is `A + i * C * K`.



Nested Array Row Access Code

```
int *get_pgh_zip( int index )
{
   return pgh[index];
}
```

Row Vector

- `pgh[index]` is an array of 5 ints.
- The starting address is `pgh+20*index`.

Code

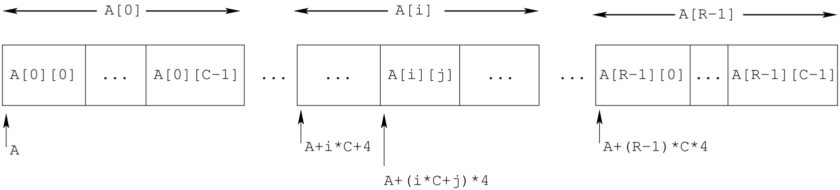
- Computes and returns the address.
- Compute this as `pgh + 4*(index + 4*index)`.

```
  # %eax holds the index
  leal   (%eax,%eax,4),%eax    # 5 * index
  leal   pgh(,%eax,4),%eax     # pgh + (20 * index)
```

Nested Array Element Access

Array Elements

- `A[i][j]` is an element of type T.
- The address is `A + (i * C + j)*K`.



Nested Array Element Access Code

```
int get_pgh_zip_dig( int index, int dig )
{
    return pgh[index][dig];
}
```

Array Elements

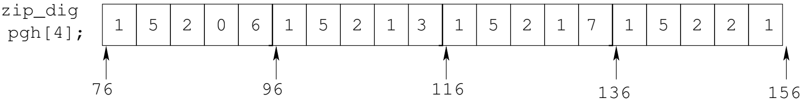
- pgh[index][dig] is an int.
- The address is pgh + 20*index + 4*dig.

Code

- Computes address pgh + 4*dig + 4*(index + 4*index).
- movl then performs the memory reference.

```
    # %ecx holds dig
    # %eax holds index
    leal   0(,%ecx,4),%edx         # 4 * dig
    leal   (%eax,%eax,4),%eax      # 5 * index
    movl   pgh(%edx,%eax,4),%eax   # *(pgh + 4*dig + 20*index)
```

Strange Referencing Examples



- Code does not do any bounds checking!
- Ordering of elements within array is guaranteed.

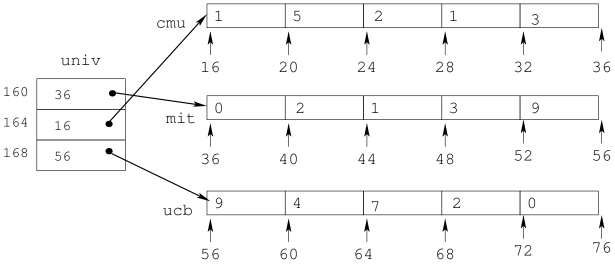
Reference	Address	Value	Guaranteed?
pgh[3][3]	$76 + 20 * 3 + 4 * 3 = 148$	2	Yes
pgh[2][5]	$76 + 20 * 2 + 4 * 5 = 136$	1	Yes
pgh[2][-1]	$76 + 20 * 2 + 4 * (-1) = 112$	3	Yes
pgh[4][-1]	$76 + 20 * 4 + 4 * (-1) = 152$	1	Yes
pgh[0][19]	$76 + 20 * 0 + 4 * 19 = 152$	1	Yes
pgh[0][-1]	$76 + 20 * 0 + 4 * (-1) = 72$	??	No

Multi-Level Array 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};
```

- Variable `univ` denotes an array of 3 elements.
- Each element is a pointer.
- Each pointer points to an array of ints.



Element Access in a Multi-Level Array

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

Computation

- Element access
  `Mem[Mem[univ+4*index]`
  `+ 4*dig]`
- Must do two memory reads:
  - First get pointer to row array.
  - Then access element within the row.

```
# %ecx = index
# %eax = dig
   leal  0(,%ecx,4),%edx          # 4 * index
   movl  univ(%edx),%edx          # Mem[univ+4*index]
   movl  (%edx,%eax,4),%eax       # Mem[...+4*dig]
```

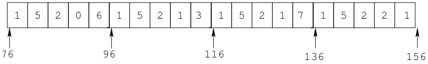
Array Element Accesses

Nested Array

```
int get_pgh_digit
   (int index, int dig)
{
   return pgh[index][dig];
}
```

Element at

`Mem[pgh+20*index+4*dig]`

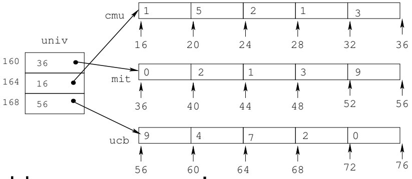


Multi-Level Array

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

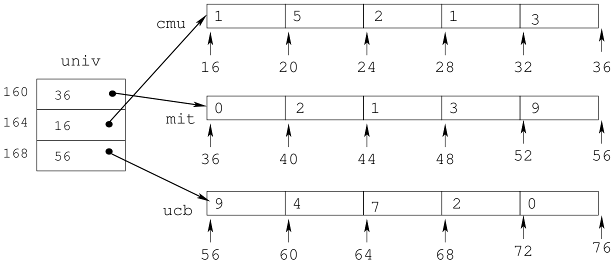
Element at

`Mem[Mem[univ+4*index]+4*dig]`



Similar C references, but different address computations.

Strange Referencing Examples



- Code does not do any bounds checking.
- Ordering of elements in different arrays is not guaranteed.

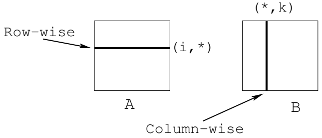
Reference	Address	Value	Guaranteed?
`univ[2][3]`	$56 + 4 * 3 = 68$	2	Yes
`univ[1][5]`	$16 + 4 * 5 = 36$	0	No
`univ[2][-1]`	$56 + 4 * (-1) = 52$	9	No
`univ[3][-1]`	??	??	No
`univ[1][12]`	$16 + 4 * 12 = 64$	7	No

Strengths

- C compiler handles doubly subscripted arrays.
- Generates very efficient code.

Limitation

- It only works with fixed array sizes.




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

/* Compute element i,i of
   fixed matrix product. */
int fix_prod_elem
   (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;
}
```

Concept

- Contiguously-allocated region of memory.
- Refer to members within the structure by name.
- Members may be of differenct types.

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

Memory Layout

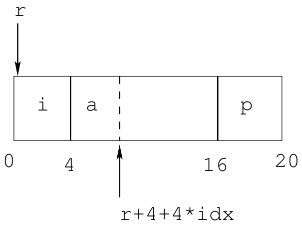
Accessing Structure Member

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

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

Generating Pointer to Struct Member

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



Generating Pointer to an Array Element

- Offset of each structure member is determined at compile time.

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

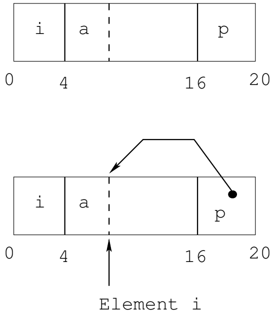
```
# %ecx = idx
# %edx = r
leal   0(,%ecx,4),%eax     # 4*idx
leal   4(%eax,%edx),%eax   # r+4*idx+4
```

Structure Referencing (Cont.)

C Code

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

void set_p(struct rec *r)
{
   r->p = &r->a[r->i];
}
```




```
# %edx = r
   movl (%edx),%ecx              # r->i
   leal 0(,%ecx,4),%eax          # 4*(r->i)
   leal 4(%edx,%eax),%eax        # r+4+4*(r->i)
   movl %eax,16(%edx)            # update r->p
```