15-213

"The course that gives CMU its Zip!"

Structured Data I: Homogenous Data
Sept. 17, 1998

Topics

- **Arrays**
  - Single
  - Nested
- **Pointers**
  - Multilevel Arrays
- **Optimized Array Code**

Basic Data Types

Integral

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

Alpha	Bytes	C
byte	1	[unsigned] `char`
word	2	[unsigned] `short`
long word	4	[unsigned] `int`
quad word	8	[unsigned] `long int`, pointers

Floating Point

- **Stored & operated on in floating point registers**
- **Special instructions for four different formats (only 2 we care about)**

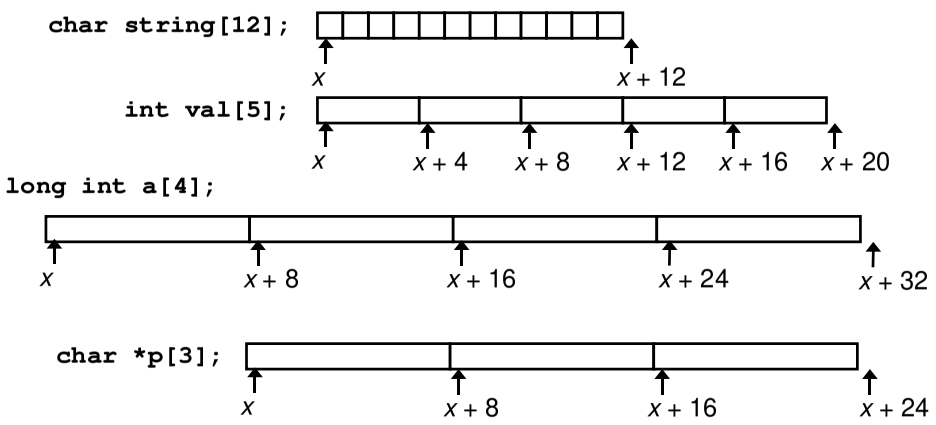
Alpha	Bytes	C
S_floating	4	`float`
T_floating	8	`double`

Array Allocation

Basic Principle

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

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

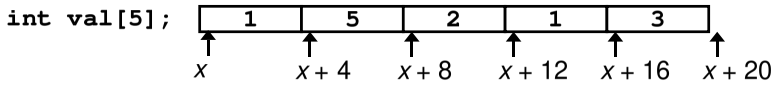


Array Access

Basic Principle

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

- **Array of data type *T* and length *L***
- **Identifier `A` can be used as a pointer to starting element of the array**

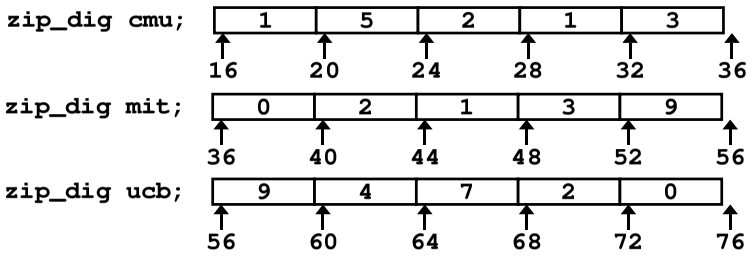


Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

Array Example

```
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 };
```



Notes

- **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

Computation

- **Register `$16` contains starting address of array**
- **Register `$17` contains array index**
- **Desired digit at `4*$17 + $16`**
- **Use instruction `s4addq`**

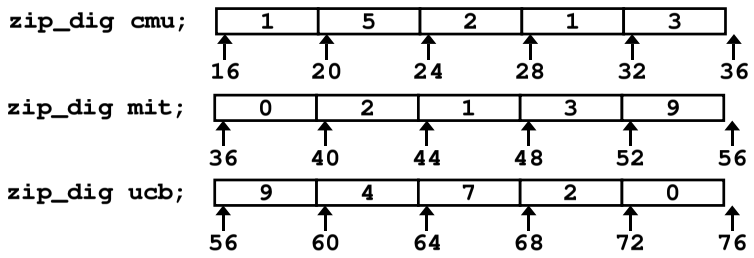
```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

Memory Access

- **Instruction `ldl` used since stored value is `int`, rather than `long int`**
  - Alpha "long word"

```
s4addq $17,$16,$17 # z + 4*dig
ldl $0,0($17)      # z[dig]
ret $31,($26),1    # return
```


Referencing Examples



Code Does Not Do Any Bounds Checking!

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

- **Out of range behavior implementation-dependent**
  - No guranteed relative allocation of different arrays

Array Loop Example

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

Registers

`$16` `z`
`$0` `zi`
`$3` `i`

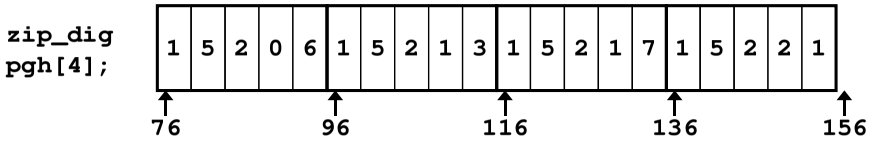
Computations

- 10 * *x* **implemented as** (4+1) * *x* + (4+1) * *x*
  - Avoids use of slow multiply instruction
- **Digit access as `4*$3 + $16`**

```
   bis $31,$31,$0    # zi = 0
   bis $31,$31,$3    # i = 0
$61:                 # loop
   s4addl $0,$0,$2   # 5*zi
   addq $2,$2,$2     # 10*zi
   s4addq $3,$16,$1  # &z[i]
   ldl $1,0($1)      # z[i]
   addl $2,$1,$0     # z[i]=10*zi+z[i]
   addl $3,1,$3      # i++
   cmple $3,4,$1     # if i<=4
   bne $1,$61        # goto loop
   ret $31,($26),1
```

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]`" equivalent to "`int pgh[4][5]`"**
  - Variable **`pgh`** denotes array of 4 elements
    - » Allocated contiguously
  - Each element is an array of 5 **`int`**'s
    - » Allocated contiguously
- **"Row-Major" ordering of all elements guaranteed**

Nested Array Allocation

Declaration

T **`A`**`[`*R*`][`*C*`];`

- **Array of data type *T***
- ***R* rows**
- ***C* columns**
- **Type *T* element requires *K* bytes**

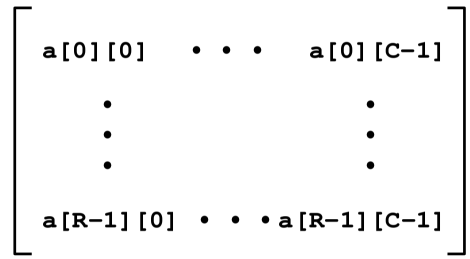
Array Size

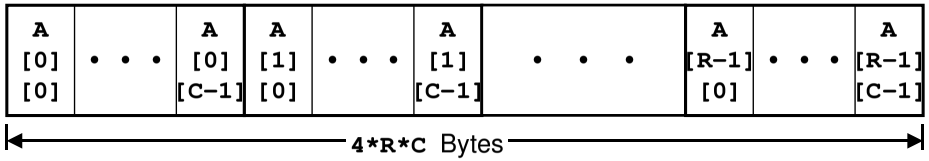
- ***R* * *C* * *K* bytes**

Arrangement

- Row-Major Ordering

`int A[R][C];`



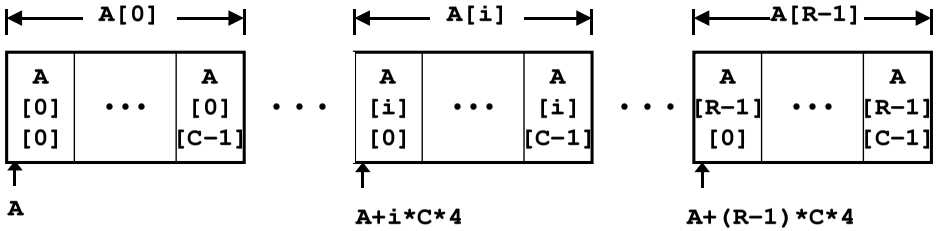


Nested Array Row Access

Row Vectors

- **`A[i]` is array of *C* elements**
- **Each element of type *T***
- **Starting address `A` + *i* * *C* * *K***

`int A[R][C];`



Nested Array Row Access Code

Row Vector

- **`pgh[index]` is array of 5 `int`'s**
- **Starting address `pgh+20*index`**

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

Code

- **Computes and returns address**

```
get_pgh_zip..ng:
  s4addq $16,$16,$16  # 5*index
  lda $0,pgh          # pgh
  s4addq $16,$0,$0    # 20*index + pgh
  ret $31,($26),1     # return computed address
```

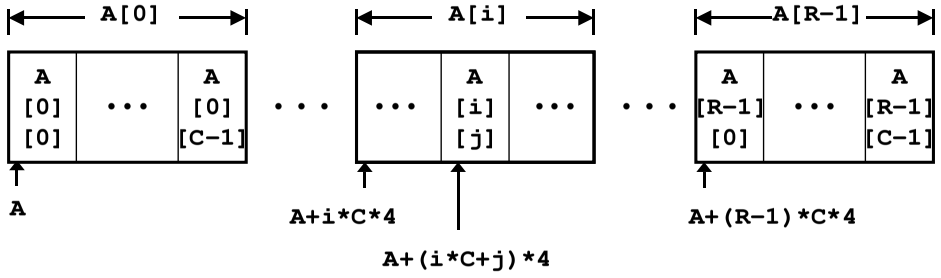
Nested Array Element Access

Array Elements

- `A[i][j]` is element of type *T*
- Address `A` + (*i* * *C* + *j*) * *K*

```
A
[i]
[j]
```

```
int A[R][C];
```



Nested Array Element Access Code

Array Elements

- `pgh[index][dig]` is `int`
- Address:

  `pgh + 20*index + 4*dig`

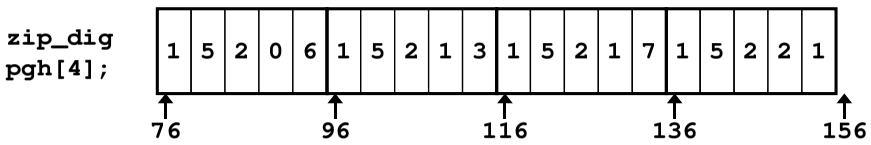
```
int get_pgh_digit
   (int index, int dig)
{
  return pgh[index][dig];
}
```

Code

- Computes address
- Loads and returns value

```
get_pgh_digit..ng:
   s4addq $16,$16,$16   # 5 *index
   lda $1,pgh           # pgh
   s4addq $16,$1,$16    # pgh + 20*index
   s4addq $17,$16,$17   # pgh+20*index+4*dig
   ldl $0,0($17)        # pgh[index][dig]
   ret $31,($26),1
```


Strange Referencing Examples



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

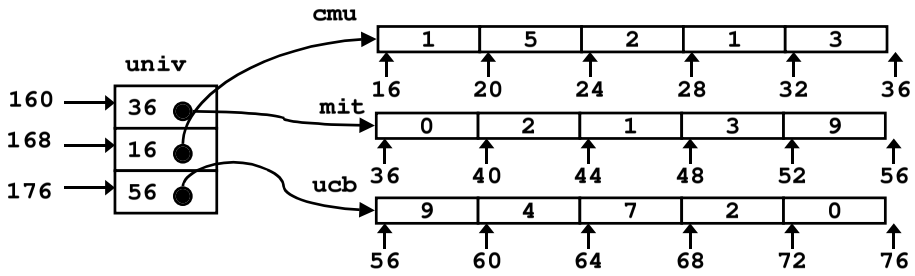
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s

```
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};
```



Referencing "Row" in Multi-Level Array

Row Vector

- `univ[index]` is pointer to array of `int`'s
- Starting address `Mem[univ+8*index]`

```
int* get_univ_zip(int index)
{
  return univ[index];
}
```

Code

- Computes address within `univ`
- Loads and returns pointer

```
get_univ_zip..ng:
   lda $1,univ        # univ
   s8addq $16,$1,$16  # univ+8*index
   ldq $0,0($16)      # univ[index]
   ret $31,($26),1
```


Accessing Element in Multi-Level Array

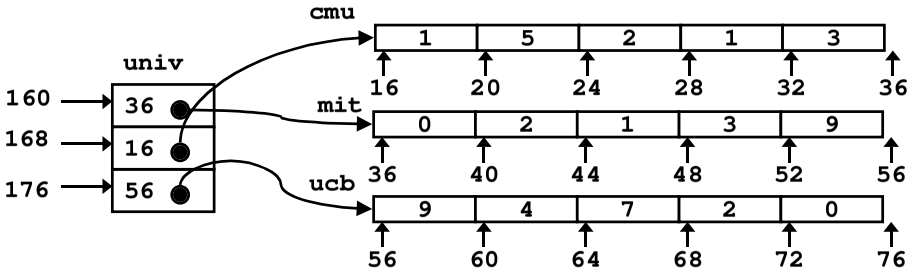
Computation

- Element access `Mem[Mem[univ+8*index]+4*dig]`
- Must do two loads
  - First get pointer to row array
  - Then access element within array

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

```
get_univ_digit..ng:
   lda $1,univ        # univ
   s8addq $16,$1,$16  # univ+8*index
   ldq $1,0($16)      # univ[index]
   s4addq $17,$1,$17  # univ[index]+4*dig
   ldl $0,0($17)      # univ[index][dig]
   ret $31,($26),1
```


Strange Referencing Examples



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

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

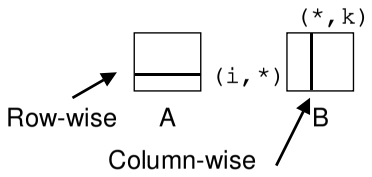
Using Nested Arrays

Strengths

- C compiler handles doubly subscripted arrays
- Generates very efficient code
  - Avoids multiply in index computation

Limitation

- Only works if have 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,
 long int i, long int k)
{
  long int j;
  int result = 0;
  for (j = 0; j < N; j++)
    result += a[i][j]*b[j][k];
  return result;
}
```

Dynamic Nested Arrays

Strength

- Can create matrix of arbitrary size

Programming

- Must do index computation explicitly

Performance

- Accessing single element costly
- Must do multiplication

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

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

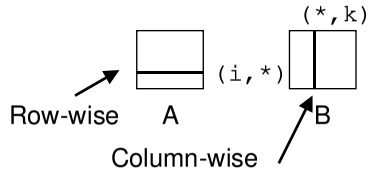
```
var_ele..ng:
 mulq $17,$19,$19    # i*n
 addq $19,$18,$19    # i*n+j
 s4addq $19,$16,$19  # a+i*n+j
 ldl $0,0($19)       # Get element
 ret $31,($26),1
```


Dynamic Array Multiplication

Without Optimizations

- Multiplies
  - 2 for subscripts
  - 1 for data
- Adds
  - 4 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,
 long int i, long int k,
 long int n)
{
  long 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`

Performance

- Compiler can optimize regular access patterns

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

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


Dynamic Array Multiplication

Optimized Code

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

Inner Loop

```
  # $0:  result
  # $4:  j
  # $18: i*n
  # $19: j*n+k (Initially =k)
$46:                  # Loop
  addq $18,$4,$1      # i*n+j
  s4addq $1,$16,$1    # &a[i][j]
  s4addq $19,$17,$2   # &b[j][k]
  ldl $3,0($1)        # a[i][j]
  ldl $1,0($2)        # b[j][k]
  mull $3,$1,$3       # product
  addq $4,1,$4        # j++
  addq $19,$20,$19    # $19+= n
  cmplt $4,$20,$1     # j < n?
  addl $0,$3,$0       # result+= ..
  bne $1,$46          # goto Loop
```