Machine-Level programming:

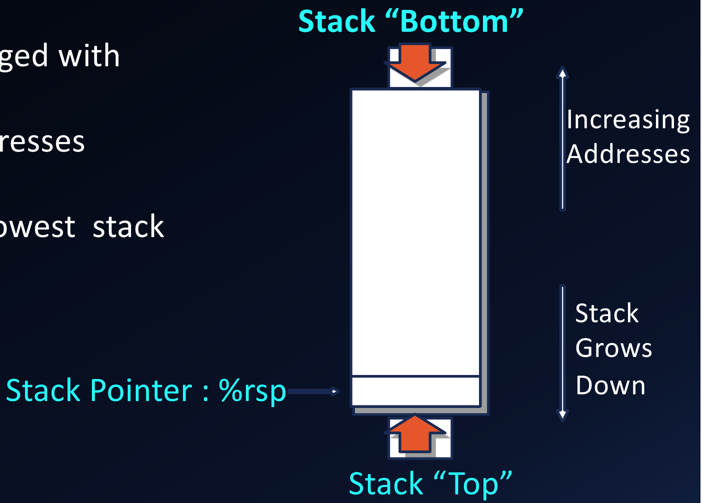
Procedures & Data

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

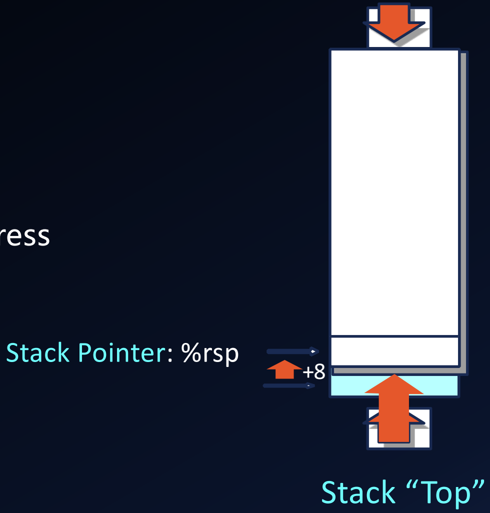
X86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %rsp contains lowest stack address
  - address of “top” element



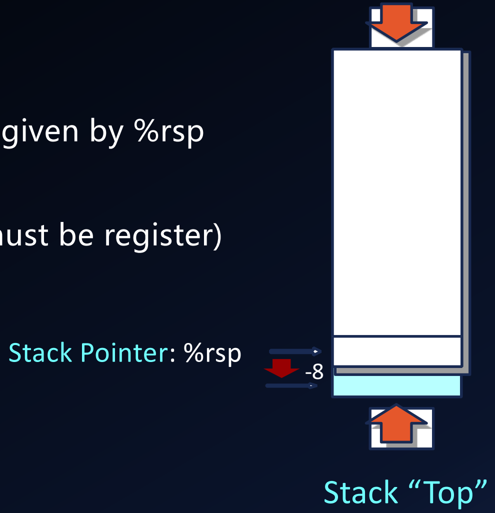
X86-64 Stack :Push

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



X86-64 Stack :Pop

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



Procedures

- Stack Structure
- Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
- Illustration of Recursion

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: call label
    - Push return address on stack
    - Jump to label
- Return address:
    - Address of the next instruction right after call
    - Example from disassembly
- Procedure Return: ret
    - Pop address from stack
    - Jump to address

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

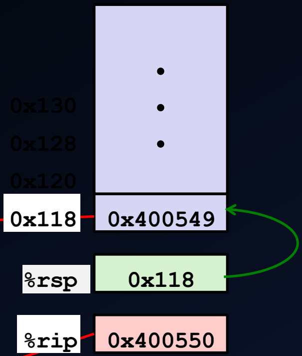
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Control Flow Example #2

Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

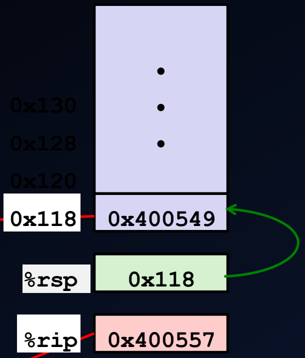


Control Flow Example #3

Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

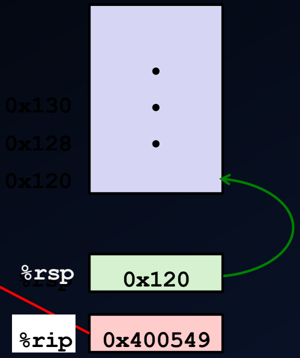


Control Flow Example #4

Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Procedure Data Flow

Registers

- First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

- Return Value

%rax

Stack

- First 6 arguments

• • •
Arg *n*
• • •
Arg 8
Arg 7

Only allocate stack space when needed

Data Flow Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  •••
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax        # a
  400553:  imul   %rsi,%rax        # a * b
  # s in %rax
  400557:  retq                    # Return
```

Managing Local Data

Stack-Based Languages

- Recursion
  - Code must be “Reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

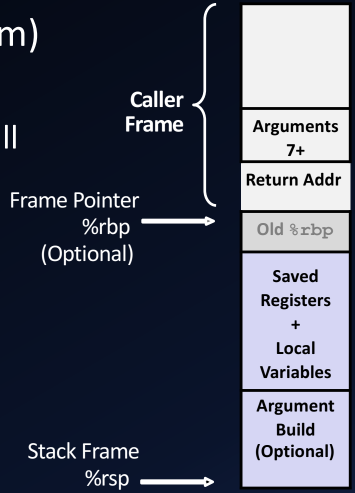
Managing Local Data

Stack-Based Languages

- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in Frames
  - state for single procedure instantiation

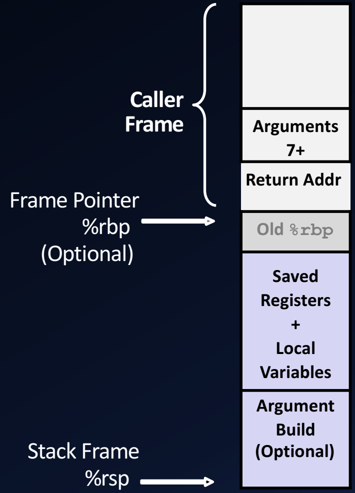
Stack Frames

- Current Stack Frame("Top" to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    if can't keep in registers
  - Saved register context
  - Old frame pointer (optional)



Stack Frames

- Caller Stack Frame
  - Return address
    - Pushed by call instruction
  - Argument for this call



Register Saving Conventions

- When procedure yoo calls who:
  - yoo is the *caller*
  - who is the *callee*
- Can register be used for temporary storage?

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

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

  - Contents of register %rdx 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 Saved”*
    - Caller saves temporary values in its frame before the call
  - *“callee Saved”*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

X86-64 Linux Register Usage #1

- %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

X86-64 Linux Register Usage #2

- %rbx,%r12,%r13,%r14
  - Callee-saved
  - Callee must save & restore
- %rdi,...,%r9
  - 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

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

Illustration of Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
  - 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

Data

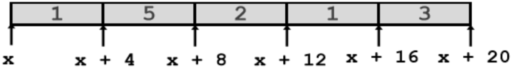
- Arrays
  - One-dimensional
  - Multi-dimensional(nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

- **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 a pointer to array element 0: Type *T**

`int val[5];`



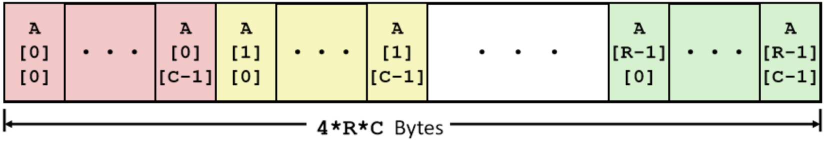
Reference	Type	Value
`val[4]`	`int`	`3`
`val`	`int *`	`x`
`val+1`	`int *`	`x + 4`
`&val[2]`	`int *`	`x + 8`
`val[5]`	`int`	`??`
`*(val+1)`	`int`	`5 //val[1]`
`val + i`	`int *`	`x + 4 * i //&val[i]`

Multi-dimensional(nested) Arrays

- **Declaration**
  - *T* `A[`*R*`][`*C*`];`
  - 2D 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

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

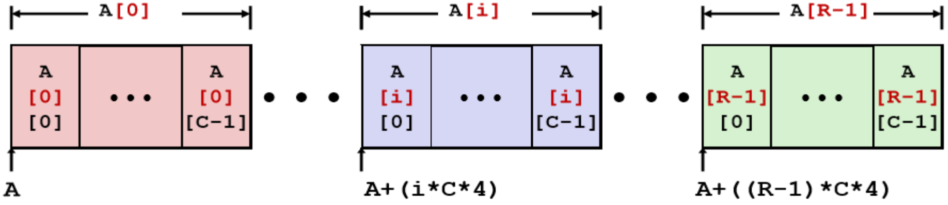
```
int A[R][C];
```



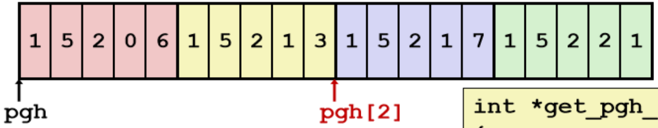
Nested Array Row Access

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

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



Nested Array Row Access Code




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

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

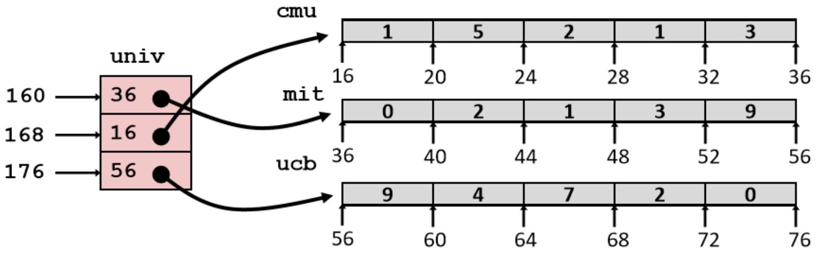
- **Row Vector**
  - `pgh[index]` is array of 5 `int`'s
  - Starting address `pgh+20*index`
- **Machine Code**
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`

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 };
```

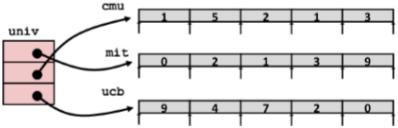
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

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



Element Access in Multi-Level 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**
  - Element access **Mem**[**Mem**[univ+8*index]+4*digit]
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

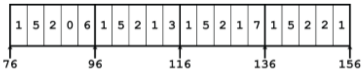
Array Element Accesses

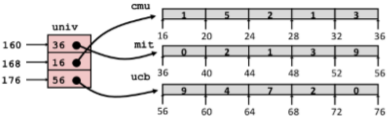
Nested array

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

Multi-level array

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





Accesses looks similar in C, but address computations very different:

`Mem[pgh+20*index+4*digit]` `Mem[Mem[univ+8*index]+4*digit]`

N X N Matrix Code

- **Fixed dimensions**
  - Know value of N at compile time
- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays
- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
            size_t i, size_t j)
{
  return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
            size_t i, size_t j)
{
  return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
            size_t i, size_t j) {
  return a[i][j];
}
```

n X n Matrix Access

- **Array Elements**
  - Address **A** **+** $i * (C * K) + j * K$
  - C = n, K = 4
  - Must perform integer multiplication

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
  return a[i][j];
}
```

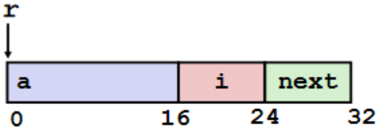
```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi              # n*i
  leaq    (%rsi,%rdi,4), %rax     # a + 4*n*i
  movl    (%rax,%rcx,4), %eax     # a + 4*n*i + 4*j
  ret
```

Data

- Arrays
    - One-dimensional
    - Multi-dimensional(nested)
    - Multi-level
- Structures
    - Allocation
    - Access
    - Alignment
- Floating Point

Structure Representation

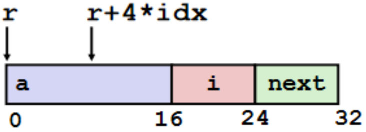
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Generating Pointer to Structure Member

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



- **Generating Pointer to 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];
}
```

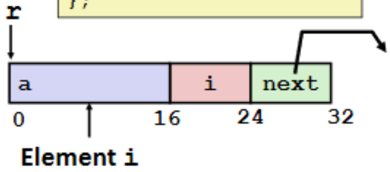
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List

- C Code

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

```
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
```

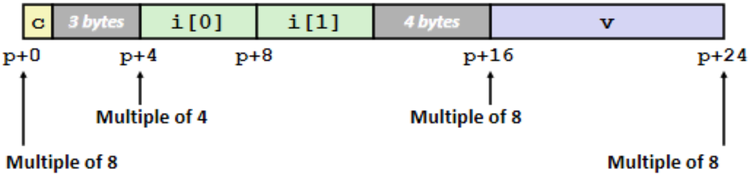
Alignment Principles

- **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 Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- **Example:**
  - K = 8, due to `double` element

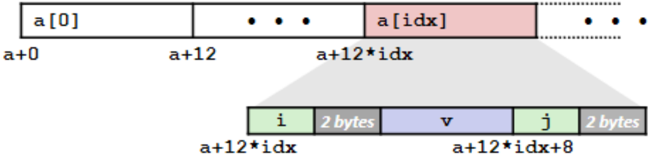
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

- **Compute array offset 12*idx**
  - `sizeof(S3)`, including alignment spacers
- **Element `j` is at offset 8 within structure**
- **Assembler gives offset a+8**
  - Resolved during linking




```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

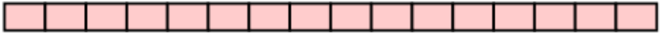
Data

- Arrays
    - One-dimensional
    - Multi-dimensional(nested)
    - Multi-level
- Structures
    - Allocation
    - Access
    - Alignment
- Floating Point

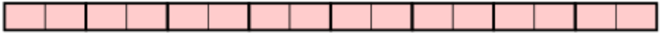
Programming with SSE3

XMM Registers

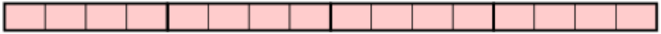
- 16 total, each 16 bytes
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers



- 4 single-precision floats

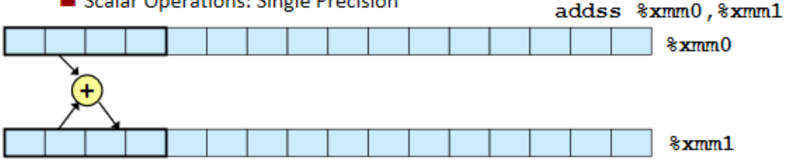
- 2 double-precision floats

- 1 single-precision float

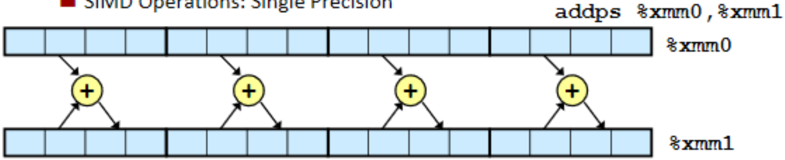
- 1 double-precision float

Scalar & SIMD Operations

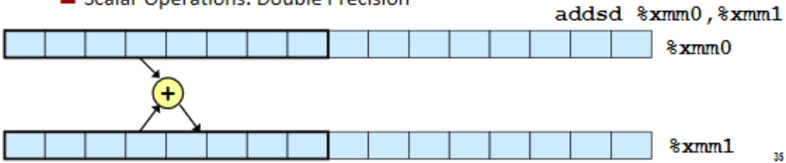
- Scalar Operations: Single Precision



- SIMD Operations: Single Precision



- Scalar Operations: Double Precision



FP Basics

- Arguments passed in `%xmm0`, `%xmm1`, ...
- Result returned in `%xmm0`
- All XMM registers caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret
```

```
 # x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
  # p in %rdi, v in %xmm0
  movapd  %xmm0, %xmm1    # Copy v
  movsd   (%rdi), %xmm0   # x = *p
  addsd   %xmm0, %xmm1    # t = x + v
  movsd   %xmm1, (%rdi)   # *p = t
  ret
```

Other Aspects of FP Code

- **Floating-point comparisons**
  - Instructions `ucomiss` and `ucomisd`
  - Set condition codes CF, ZF, and PF
- **Using constant values**
  - Set XMM0 register to 0 with instruction `xorpd %xmm0, %xmm0`
  - Others loaded from memory

例题3.61

**3.61 C编译器为 var_prod_ele 产生的代码（图 3-29）不能将它在循环中使用的所有值都放进寄存器中，因此它必须在每次循环时都从存储器中读出 *n* 的值。写出这个函数的 C 代码，使用类似于 GCC 执行的那些优化，但是它的编译代码不会让循环值溢出到存储器中。

回忆一下，处理器只有 6 个寄存器可用来保存临时数据，因为寄存器 %ebp 和 %esp 不能用于此目的。其中一个寄存器还必须用来保存乘法指令的结果。因此，你必须把循环中的值的数量从 6 个（result、Arow、Bcol、j、n 和 4*n）减少到 5 个。

需要找到一个对你那种编译器行之有效的策略。不断尝试各种不同的策略，直到有一种能工作。

```
/* Compute i,k of variable matrix product */
int var_prod_ele(int n, int A[n][n], int B[n][n], int i, int k) {
    int j;
    int result = 0;

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

    return result;
}
```

图 3-29 计算变长数组的矩阵乘积的元素 *i*, *k*。编译器执行的优化类似于对定长数组的优化

例题3.61

下面是 var_prod_ele 循环的汇编代码：

```
n stored at %ebp+8
Registers: Arow in %esi, Bptr in %ecx, j in %edx,
  result in %ebx, %edi holds 4*n
.L30:                              loop:
  movl   (%ecx), %eax                Get *Bptr          (eax)
  imull  (%esi,%edx,4), %eax         Multiply by Arow[j]  Arow
  addl   %eax, %ebx                  Add to result      result +=
  addl   $1, %edx                    Increment j        ++j
  addl   %edi, %ecx                  Add 4*n to Bptr    Bptr
  cmpl   %edx, 8(%ebp)               Compare n:j        cmp n:j
  jg     .L30                        If >, goto loop
```

例题3.61

我们看到程序既使用了伸缩过的值 4*n*（寄存器 %edi）来增加 Bptr，也使用了存储在相对于 %ebp 偏移量为 8 处的 *n* 的实际值来检查循环的边界。C 代码中并没有体现出需要这两个值，但是由于指针运算的伸缩，才使用了这两个值。每次循环中，代码从存储器中取出 *n* 的值，检查循环是否终止（第 7 行）。这是一个寄存器溢出（register spilling）的例子：没有足够多的寄存器来保存需要的临时数据，因此编译器必须把一些局部变量放在存储器中。在这个情况下，编译器选择把 *n* 溢出，因为它是一个“只读”的值——在循环中不会改变它的值。因为 IA32 处理器的寄存器数量太少，必须常常将循环值溢出到存储器中。通常，读存储器完成起来比写存储器要容易得多，因此将只读变量溢出是比较合适的。关于如何改进这段代码以避免寄存器溢出，请参见家庭作业 3.61。

3.61

```c
int var_prod_ele(int n, int A[n][n], int B[n][n
], int i, int k)
{
    int j = n-1;
    int result = 0;
    for(; j!=-1; --j)
        result += A[i][j] * B[j][k];
    return result;
}
```

但是这样得到的结果仍然会使用到存储器。

按下面的代码，循环里面貌似就没有用到存储器。

但是用到了一个常量 4，就是增加 a 的时候，会 add 4。

只需要 result，a，e，b，4n 这五个变量。

```c
int var_prod_ele(int n, int A[n][n], int B[n][n
], int i, int k)
{
    int result = 0;
    int *a = &A[i][0];
    int *b = &B[0][k];
    int *e = &A[i][n];
    for(;a!=e;)
    {
        result += *a * *b;
        b+=n;
        a++;
    }
```

```
    return result;
}
```

下面是其汇编代码的循环部分：

edi 是 4*n，ebx 和 edx 分别是 b 和 a，esi 是 e，eax 是 re-sult。

ecx 是用于存储乘法的寄存器。

```
L4:
movl (%ebx), %ecx
imull (%edx), %ecx
addl %ecx, %eax
addl %edi, %ebx
addl $4, %edx
cmpl %edx, %esi
jneL4
```