Intel x86 Assembly Language Programming

CMST 385 – Systems and Database Administration

Tim Bower, Kansas State Universtity – Polytechnic Campus

The Intel x86 line of CPUs use the **accumulator machine** model.

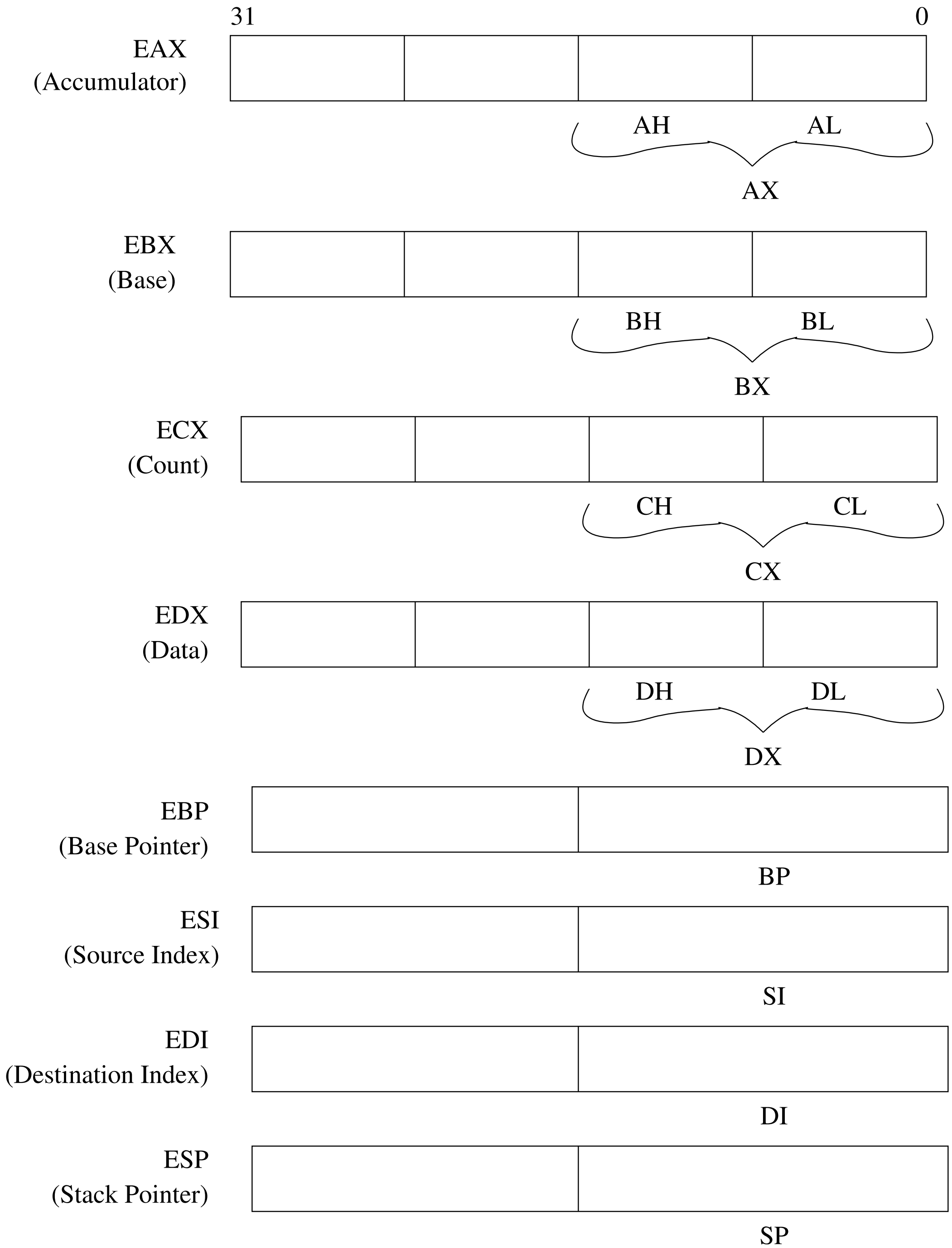
Registers

Note that each register has 32 bit, 16 bit and 8 bit names. We will usually use just the 32 bit names for the registers. See the diagrams of the registers on the following pages.

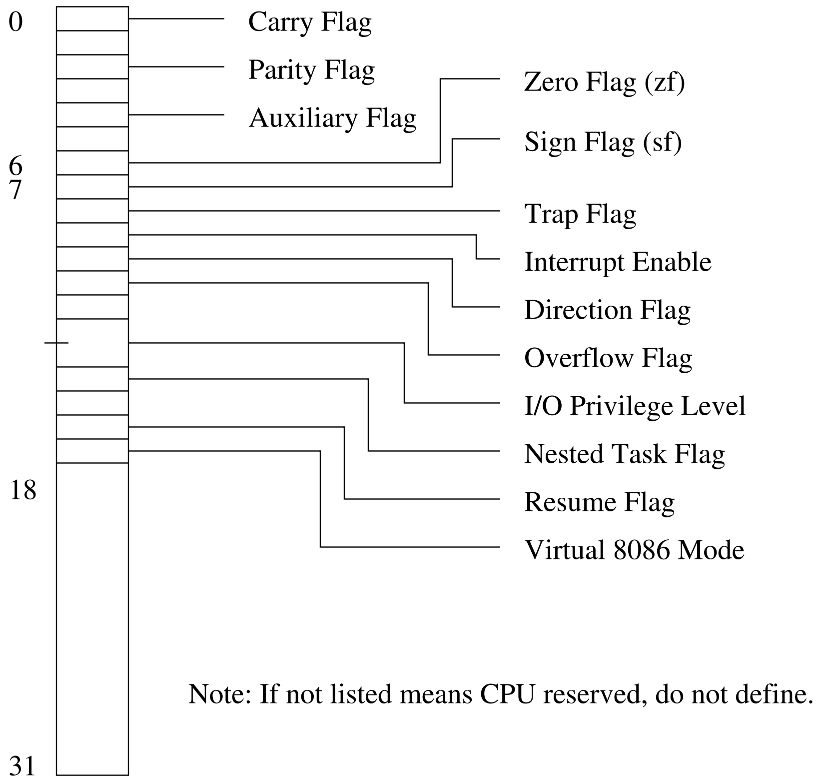
- The primary accumulator register is called EAX. The return value from a function call is saved in the EAX register. Secondary accumulator registers are: EBX, ECX, EDX.
- EBX is often used to hold the starting address of an array.
- ECX is often used as a counter or index register for an array or a loop.
- EDX is a general purpose register.
- The EBP register is the stack frame pointer. It is used to facilitate calling and returning from functions.
- ESI and EDI are general purpose registers. If a variable is to have register storage class, it is often stored in either ESI or EDI. A few instructions use ESI and EDI as pointers to source and destination addresses when copying a block of data. Most compilers preserve the value of ESI and EDI across function calls — not generally true of the accumulator registers.
- The ESP register is the stack pointer. It is a pointer to the "top" of the stack.
- The EFLAGS register is sometimes also called the status register. Several instructions either set or check individual bits in this register. For example, the sign flag (bit 7) and the zero flag (bit 6) are set by the compare (cmp) instruction and checked by all the conditional branching instructions.
- The EIP register holds the instruction pointer or program counter (pc), which points to the next instruction in the text section of the currently running program.

Memory Segmentation and Protection

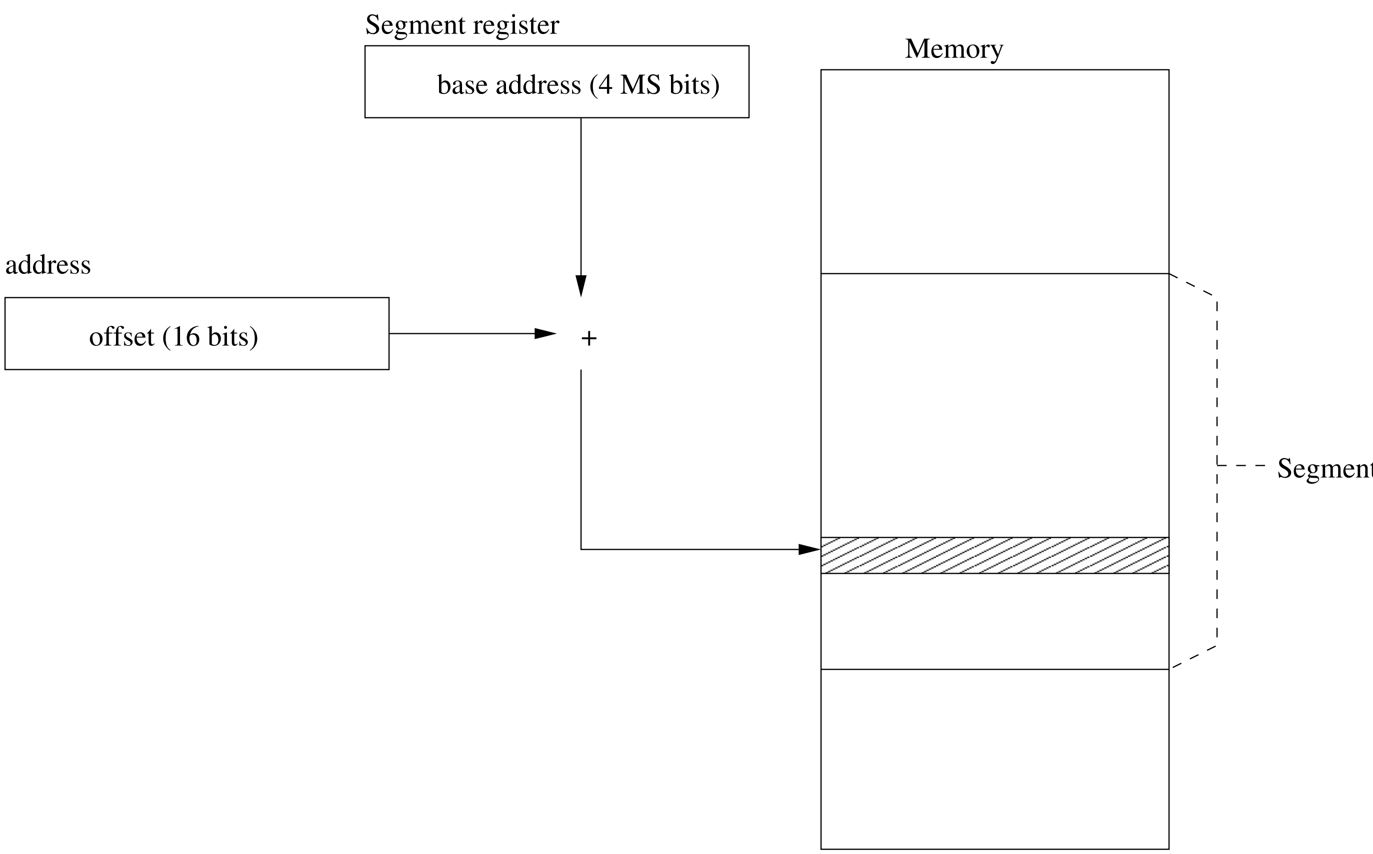
The earliest processors in the x86 family had 16 bit registers, thus memory addresses were limited to 16 bits (64 Kbytes). This amount of memory is not large enough for both the code and the data of many programs. The solution was to *segment* the memory into 64 K blocks. The code goes into one segment,



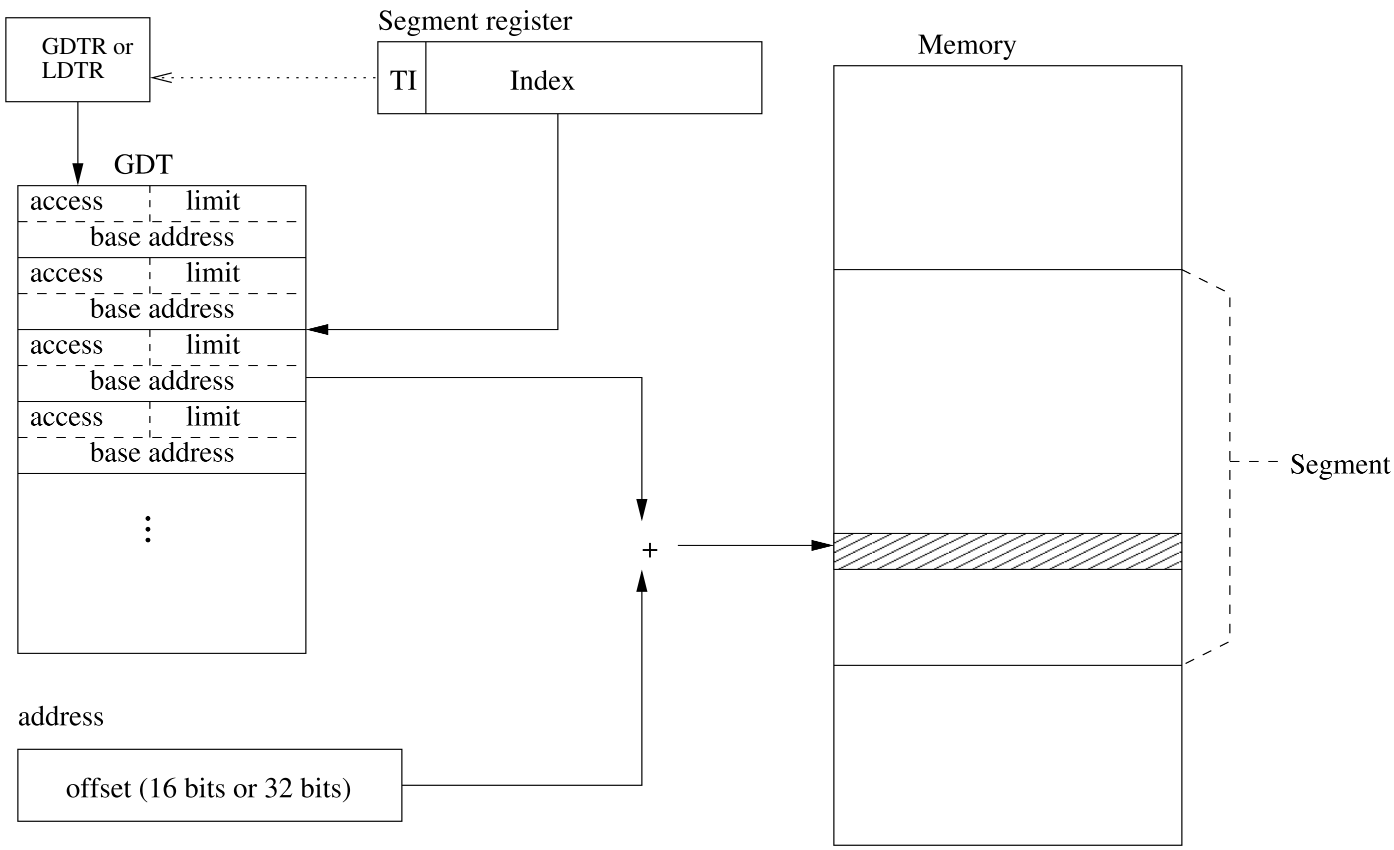
General Register Designations for x86 CPUs.



EFLAGS Register.



Real Mode, Segmented Memory Model.



Protected Mode, Segmented Memory Model.

the data into another, and the stack is placed into a third segment. Each segment is given its own address space of up to 64 Kbytes in length. The 16–bit addresses used by the program are actually an offset from a segment base address. This is called *real mode, segmented memory model* and instructions and data are referenced relative to a base address held in the segment register (see diagram). The segment registers are CS (code segment), SS (stack segment), DS, ES, FS, GS (all data segments). The segmented model increases the addressable memory size to $2^{20} = 1Mbyte$. The segment and offset registers are combined in an unusual manner. The two registers are offset by four bits and added together to come up with a 20-bit address. This is the memory model used by DOS.

The only advantage to this mode was that it was very easy for developers to write their own device drivers. Once DOS loaded a program, it stayed out of the way and the program had full control of the CPU. The program can either let the BIOS handle the interrupts or handle them itself. This worked great for small programs which could fit into the available memory and did not require multi-tasking.

BIOS: Software in read–only–memory of the computer with basic device drivers and interrupt handlers for I/O devices (keyboard, drives, monitor, printer, mouse). BIOS is used when the computer is turned on to load the operating system. Modern operating systems (Unix, Linux, Windows) do not use the BIOS drivers once the operating system is running (booted).

For more demanding applications, the limitations of the real mode scheme were prohibitive. So beginning with the Intel 80286 processor, a *protected mode* was also available. In protected mode, these processors provide the following features:

Protection: Each program can be allocated a certain section of memory. Other programs cannot use this memory, so each program is protected from interference from other programs.

Extended memory: Enables a single program to access more than 640K of memory.

Virtual memory: Expands the address space to 16 MB for 16–bit processors and 4 GB for 32–bit processors (80386 and later).

Multitasking: Enables the microprocessor to switch from one program to another so the computer can execute several programs at once.

In the *protected mode, segmented memory model*, the code segment contains an offset into the *global descriptor table*, where more details about the base address and memory protection / limits are stored. A special register called the *GDTR* points to the location of the GDT and the segment registers hold offsets pointing to the desired entry called a segment descriptor in the GDT (see diagram). The Minix OS uses a protected mode, segmented memory model. Minix boots into this mode and stays in protected mode. Very complicated articles can be found in literature and on the Internet describing how a DOS program can switch the processor to protected mode and then return to real mode when the program exits.

Modern x86 based operating systems (Windows and Linux) use a *protected mode, flat memory model* where the base memory addresses in the segment descriptors in the GDT are all set to the same value. This mode greatly simplifies things, making segmentation and memory protection a non-issue for programmers.

Summary

4004 First Intel CPU - 4 bit.

8088 16 bit CPU with 8 bit external data bus. DOS ran in real mode with segments.

8086 16 bit CPU.

80186 Used mainly with embedded systems. Added some new instructions.

80286 Added protected mode. Some versions of Unix (SC0 Xenix, minix) used protected mode with segments.

80386 32 bit CPU. Windows 3.0, Linux used protected mode flat memory model.

80486 Math co-processor now included on CPU.

Pentium Faster; later Pentiums have a RISC core processor.

IA-64 aka Itanium - 64 bit processor.

Addressing Modes

The **addressing mode** refers to how operands are referenced in an assembly language instruction. We will use the `mov` instruction here to describe the available addressing modes of the x86 family of processors. The `mov` instruction copies data between two locations. It's syntax is shown below — `dest` and `source` represent the operands. Data is copied from the `source` to the `dest`ination.

```
mov dest, source
```

Register Mode A register mode operand simply names a register. Both operands use register mode below. Here we copy the contents of register ECX to register EAX. Note that register names are not case sensitive in the assembly code.

```
mov EAX, ECX
```

Immediate Mode An immediate mode operand is a constant listed directly in the code. Below, we use immediate mode with the second operand to store the value 10 in the EAX register. The immediate mode operand must be the source operand.

```
mov EAX, 10
```

Register Indirect (On SPARC, this same mode is called *Register direct.*) Here we use a register to hold a pointer (address in main memory) of where data can be moved to or from. Both operands of an instruction can not be register indirect — one of the operands must be either register mode or immediate mode. Brackets are placed around the operand to indicate register indirect. In C language terminology, brackets may be viewed as the dereference operator. Some compilers use square brackets, others use parentheses.

```
mov [EAX], EDX   ; contents of edx goes to address pointed to by eax.
mov ebx, [edx]   ; data at address pointed to by edx goes to ebx.

; the semicolon designates the beginning of a comment for some
assemblers.
! other assemblers use the exclamation mark for comments.
```

Base Displacement Constants or offsets of 8–, 16– or 32–bits may also be added to the contents of a register to come up with an effective address. As shown below, there are several forms of base displacement. The other operand combined with a base displacement operand must be either register mode or immediate mode.

```
mov EBX, 16[EBP]        ; data at 16+EBP goes to EBX
mov ebx, [ebp+16]       ; same as above
mov ebx, [ebp]16        ; same as above
mov [EDI][EBP], 10      ; 10 goes to EDI+EBP
mov [EDI][EBP+16], 18   ; 18 goes to EDI+EBP+16
```

The default operation with the `mov` instruction is to move 32– bits (double word) of data. Some compilers (MS Visual C++), specify the type of operation even if it is the default.

```
mov EAX, DWORD PTR [EBX]
```

There are actually several ways of specifying a smaller quantity of data to be copied. The following are all examples of instructions which copy 16–bits (word) of data.

```
mov EAX, WORD PTR [EBX]
mov AX, [EBX]
o16 mov -6(ebp), 3
```

The keyword `byte` or the 8–bit designation of a register may be used to copy 8 bits of data.

Basic Instructions

In the descriptions of the instructions, the following symbols are used to indicate the accepted addressing modes.

Operator Type	Definition
reg	register mode operand
immed	immediate mode operand (a constant)
mem	operand is a memory address, either register indirect or base displacement operand.

Listed here are only the most commonly used instructions. Information on additional instructions can be found from the Intel manual (/pub/cis450/Pentium.pdf or /pub/cis450/x86Instructions.ps)

Data Movement Instructions

Instruction	Operands	Notes
mov movb	reg, immed reg, reg reg, mem mem, immed mem, reg	Copy data movb copies one byte destination, source destination is overwritten
movsx	reg, immed reg, reg reg, mem	Copy data with sign extend
movzx	reg, immed reg, reg reg, mem	Copy data with zero extend
push	reg immed	Copy data to the top of the stack (esp) The stack pointer (ESP) is decremented by 4 bytes.
pop	reg	Copy data from the top of the stack to a register The stack pointer (ESP) is incremented by 4 bytes.
lea	reg, mem	Load a pointer (memory address) in a register

Integer Arithmetic Instructions

The destination register for all of these instructions must be one of the accumulator registers (EAX, EBX, ECX, EDX).

Instruction	Operands	Notes
add	reg, reg reg, immed reg, mem	two's complement addition first operand is used as source and overwritten as destination
sub	reg, reg reg, immed reg, mem	two's complement subtraction first operand is used as source and overwritten as destination
inc	reg	increment the value in register
dec	reg	decrement the value in register
neg	reg	additive inverse
mul	EAX, reg EAX, immed EAX, mem	Unsigned multiply Some compilers tend to use imul instead
imul	reg reg, reg reg, immed reg, mem	Signed multiply, EAX*reg → EAX
div	reg mem	Unsigned divide EAX / reg,mem; EAX = quotient, EDX = remainder,
idiv	reg mem	Signed divide EAX / reg,mem; EAX = quotient, EDX = remainder,

Structure of an assembly language file

In addition to the assembly instructions, there are a few other declarations in an assembly language program produced by a compiler.

Here we review the elements of an assembly language program. These notes are for the Minix assembler. There may be some variance with other assemblers.

Segment declaration

There are four different assembly segments: text, rom, data and bss. Segments are declared and selected by the *sect* pseudo-op. It is customary to declare all segments at the top of an assembly file like this:

```
.sect .text; .sect .rom; .sect .data; .sect .bss
```

Then within the body of the code, segment declarations are used to begin the declarations for each segment. Note that the '.' symbol refers to the location in the current segment.

Labels

There are two types: name and numeric. Name labels consist of a name followed by a colon (:).

The numeric labels are single digits. The nearest 0: label may be referenced as 0f in the forward direction, or 0b backwards.

Statement Syntax

Each line consists of a single statement. Blank or comment lines are allowed.

The most general form of an instruction is

```
label: opcode operand1, operand2    ! comment
```

Local Variables and the Stack

The stack is used to store local variables. They may be put on the stack with either the `push` instruction or by first allocating space on the stack (subtract from `esp`) and then using the `mov` instruction to store data in the allocated space. Here we will show an example of how local variables are used from the stack.

Recall that the stack is upside down from how stacks are normally viewed in that the "*top*" of the stack has the lowest memory address of the stack data. The processor maintains a special register (ESP) which is a pointer to the memory address of the 'top' of the stack. Another important register associated with the stack is the frame pointer (EBP). The frame pointer is sort of a book-mark or reference point in the stack. Nearly all memory references are relative to the frame pointer. Management of the frame pointer is critical to how functions are called and more importantly, how the program returns to the calling function. Function calls will be covered in more detail later.

C compilers implement a restriction that each function may only access (i.e. scope) those elements on the stack which are within the function's **Activation Record**. The Activation Record for each function includes the following:

function parameters	
return address	
old frame pointer	⟵ frame pointer (ebp)
local variables	⟵ stack pointer (esp)

To set up the frame pointer at the beginning of each function (including main), the following two lines of assembly code are used.

```
push ebp
mov ebp,esp
```

So first, the old frame pointer is pushed onto the stack for use when the function returns to the calling (parent) function. Then, since the old frame pointer is now at the top of the stack, we can use the pointer value in the `esp` register to copy a pointer to where the old frame pointer was stored to the `ebp` register, making this the new frame pointer.

Here is a simple example of how local variables in the stack are managed. Try to draw a memory map of the stack.

Function Calls and the Stack

The stack is also used to store data that is used for making calls to functions. Data is pushed onto the stack when a function is called and is removed from the stack when the function returns.

```
                              .sect .text; .sect .rom; .sect .data; .sect .bss
                              .extern _main
#include <stdio.h>            .sect .text
                              _main:
int main(void)                push ebp
{                             mov ebp,esp
    char c = 'a';             sub esp,12
    int i;                    push esi
    short j;                  movb -1(ebp),97
                              mov esi,10
    i = 10;                   o16 mov -10(ebp),5
    j = 5;                    movsx eax,-10(ebp)
    i += j;                   add esi,eax
}                             pop esi
                              leave
                              ret
```

Recall that C compilers implement a restriction that each function may only access (i.e. scope) those elements on the stack which are within the function's **Activation Record**. The Activation Record for each function includes the following:

function parameters	
return address	
old frame pointer	⟵ frame pointer (ebp)
local variables	⟵ stack pointer (esp)

The steps for a function are the same for every C function. It should be pointed out that this is the scheme used by compilers. Some assembly programmers follow this scheme for hand written assembly code. But many assembly programmers never worry about setting the frame pointer.

1. The calling function pushes the function parameters onto the stack prior to the function call.
2. The call instruction pushes the return address (EIP register) onto the stack which is used on function exit by the ret (return) instruction which loads the EIP register with this address.
3. The function (assembly code) pushes the old frame pointer onto the stack and sets the EBP register to point to this location on the stack.

   ```
   push ebp
   mov ebp,esp
   ```

4. During the execution of the function, the frame pointer is used as a reference point to the rest of the memory in the activation record. On function exit, the leave instruction loads the EBP register from this saved value so that when control returns to the calling function, the frame pointer is still correct.

5. Local variables are stored on the stack and are removed from the stack when the function exits.

6. If the function returns data to the calling function, the return value is placed in the EAX register.

7. The calling function removes and discards the function parameters when control is returned from the function.

8. The calling function looks to the EAX register for a return value.

```
int main(void)
{
   ...                                  k |   c       |
   f(a, b, c);                          j |   b       |
   ...                                  i |   a       |
}                                         | ret addr |
                                          | old fp   | <--- fp (ebp)
void f(int i, int j, int k)               |   x       |
{                                         |   y       |
   int x, y, z;                           |   z       | <--- sp (esp)
   ...
}
```

Some instructions related to function calls are:

call
1. push eip
2. Jump to the new location (set eip to the location of the instructions for the called function).

leave
1. mov esp,ebp — throw away local variables
2. pop ebp — set frame pointer back to old value

ret n
1. pop eip — set pc to return to calling function
2. pop n words and discard — n is almost always 0.

Here is a more extensive example, again try to draw a memory map. Check your memory map with the memory map posted on the class web page for ar.c. This example includes examples of global and static data which are saved in the bss and data section of memory.

```
#include <stdio.h>

int gbss;
int gdata = 5;

int f( int, int, int );

int main(void)
{
   int lauto1, lauto2, lauto3;
   static int lbss;

   gbss = 10;
   lbss = 20;
   lauto1 = f( gdata, gbss, lbss );
   lauto2 = 5;
   lauto3 = 15;
   printf( "%d %d %d\n", lauto1, lauto2, lauto3 );
   printf( "%d\n", f( lauto3, lauto2, 5 ));

   return 0;
}

int f( int a, int b, int c )
{
   static int d;
   int e;

   d += a + b + c;
   e = d*a;
   return e;
}
```

```
.sect .text; .sect .rom; .sect .data; .sect .bss
.extern _gdata
.sect .data
_gdata:
.extern _main
.data4 5            ! gdata = 5 in data section
.sect .text
_main:
push ebp              ! save old frame pointer
mov ebp,esp           ! new frame pointer goes to ebp
sub esp,4           ! lauto1 = -4(ebp)
```

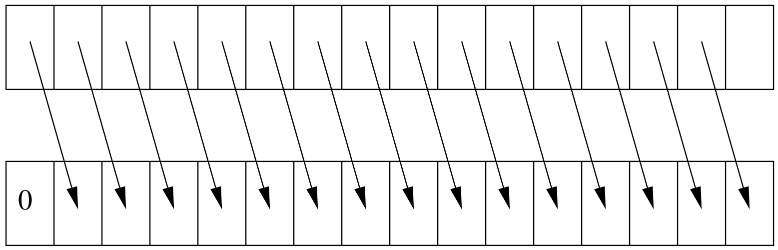
```
push esi              ! lauto3 = esi -- note: register without asking
push edi              ! lauto2 = edi
.sect .bss
.comm I_1,4           ! 4 bytes in bss (I_1) for static int lbss
.sect .text
mov (_gbss),10        ! gbss = 10
mov edx,20
mov (I_1),edx         ! lbss (I_1) = edx = 20
push edx
push (_gbss)          ! push params in reverse order
push (_gdata)
call _f
add esp,12            ! remove params from stack
mov -4(ebp),eax       ! lauto1 = f(...)
mov edi,5             ! lauto2 = 5
mov esi,15            ! lauto3 = 15
push esi
push edi              ! push params in reverse order
push -4(ebp)
push I_2              ! format ... "%d %d %d\n"
call _printf
add esp,16            ! remove params
push 5
push edi
push esi
call _f
add esp,12            ! remove params
push eax              ! push return value to stack
push I_3              ! format ... "%d\n"
call _printf
pop ecx
pop ecx               ! remove params, alternate to 'add esp,8'
xor eax,eax           ! return 0
pop edi
pop esi               ! restore registers
leave                 ! restore old frame pointer from stack
ret                   ! return address comes from stack
.sect .rom            ! rom is part of text
I_3:
.data4 680997         ! format ... "%d\n"
I_2:                  ! format ... "%d %d %d\n"
.data4 622879781
.data4 1680154724
.extern _f
.data4 10
```

```
.sect .text
_f:
push ebp                ! save old frame pointer
mov ebp,esp             ! new frame pointer goes to ebp
sub esp,4             ! e = -4(ebp)
.sect .bss
.comm I_4,4             ! 4 bytes in bss (I_4) for static int d
.sect .text
mov edx,12(ebp)
add edx,8(ebp)          ! add parameters (a, b, c)
add edx,16(ebp)
add edx,(I_4)           ! d += a + b + c
mov (I_4),edx
imul edx,8(ebp)         ! edx = d*a
mov eax,edx             ! return e; note -- no need to save edx to -4(ebp)
leave                   ! restore old frame pointer from stack
ret                     ! return address comes from stack
.extern _gbss
.sect .bss
.comm _gbss,4           ! 4 bytes in bss for global int lbss
.sect .text
```

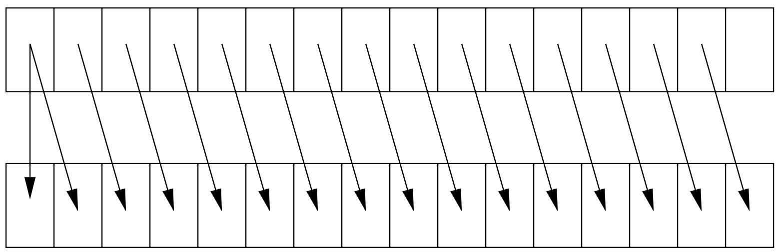
Additional Instructions

Logical Instructions

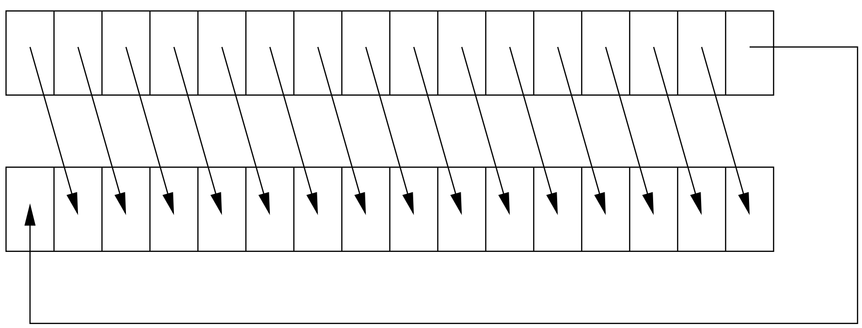
Instruction	Operands	Notes
not	reg	logical not (one's complement operation)
and	reg, reg reg, mem reg, immed	logical and
or	reg, reg reg, mem reg, immed	logical or
xor	reg, reg reg, mem reg, immed	logical xor
cmp	reg, reg reg, mem reg, immed mem, immed	Compare (dest - source) result in EFLAGS sf and zf see control instructions
test	reg, reg reg, mem reg, immed	logical and, EFLAGS set based on result see control instructions



Logical Shift Right



Arithmetic Shift Right



Rotate Shift Right

A logical shift moves the bits a set number of positions to the right or left. Positions which are not filled by the shift operation are filled with a zero bit. An arithmetic shift does the same, except the sign bit is always retained. This variation allows a shift operation to provide a quick mechanism to either multiply or divide 2's–complement numbers by 2.

Instruction	**Operands**	**Notes**
sal	reg, immed	arithmetic shift left
shl	reg, immed	logical shift left
sar	reg, immed	arithmetic shift right
shr	reg, immed	logical shift right
rol	reg, immed	rotate shift left
ror	reg, immed	rotate shift right

Example: Multiply and Divide by multiple of 2

Control Instructions

The following instructions are used to implement various control constructs (if, while, do while, for). Conditional branch instructions follow a cmp or test instruction and evaluate the sign and zero flag (SF, ZF) bit in the EFLAGS register. For each of these instructions, the operand is the name of a label found in the assembly code.

See the notes below on control flow for examples of how they are used.

Instruction	Operands	Notes
jmp	label	unconditional jump
jg jnle	label	jump if greater than zero
jge jnl	label	jump if greater than or equal to zero
jl jnge	label	jump if less than zero
jle jng	label	jump if less than or equal to zero
je jz	label	jump if zero
jne jnz	label	jump if not zero

Iterative Instructions

The above control instructions can be used to implement looping constructs, but there are also some special instructions just for the purpose of looping.

Instruction	Operands	Notes
loop	label	decrement ecx and if ecx is not equal to zero, jump
loope	label	jump if ZF in EFLAGS is set and ecx is not equal to zero ecx is decremented
loopne	label	jump if ZF in EFLAGS is not set and ecx is not equal to zero ecx is decremented
rep	instruction	execute the instruction and decrement ecx until ecx is zero.

String Handling Instructions

These instructions are all used to copy data from one string to another. In each case the source location is the address in esi while destination is the address in edi. After the move, the esi and edi registers are either incremented and decremented by the appropriate amount depending on the direction flag (DF) in the EFLAGS register. If DF is 0 (CLD instruction was executed), the registers are incremented. If DF is 1 (STD instruction was executed), the registers are decremented.

Instruction	Notes
movs movsb	move one byte from [esi] to [edi]
movsw	move one word (2 bytes) from [esi] to [edi]
movsd	move one double word (4 bytes) from [esi] to [edi]

Here is a quick example:

```
lea  edi, -20(ebp)  ! destination
lea  esi, -40(ebp)  ! source
mov  ecx,10         ! copy 10 bytes
cld                 ! increment esi and edi
rep  movsb          ! move 10 bytes, one at a time
```

Miscellaneous Instructions

Instruction	Notes
cld	Clear the direction flag; used with string movement instructions
std	Set the direction flag; used with string movement instructions
cli	Clear or disable interrupts; Reserved for the OS
sti	Set or enable interrupts; Reserved for the OS
nop	no operation, used to make a memory location addressable

Input/Output Instructions

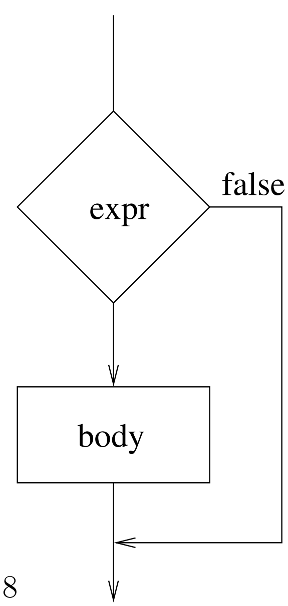
Instruction	Operands	Notes
in	acc,port	Read data in and save to eax, ax or al. The port is the base memory address for the hardware being read from (eg., a sound card).
out	acc,port	Write data in eax, ax or al to an I/O port.
insb insw		Read string data in and save to memory. The I/O port is taken from the edx register (eg., a keyboard or serial port). The destination is taken from the edi register. If used in a loop or with rep, the destination address is incremented or decremented depending on the direction flag.
outsb outsw		Write string data from memory to I/O port. The I/O port is taken from the edx register (eg., a keyboard or serial port). The source is taken from the esi register. If used in a loop or with rep, the source address is incremented or decremented depending on the direction flag.

Control Flow

In assembly language, the instructions used to implement control constructs is the various forms of the jump instructions. This is usually accomplished with a comparison (`cmp`) instruction to evaluate a logical expression following a conditional jump instruction.

if block

```
if( expr ) {
   body
}
```



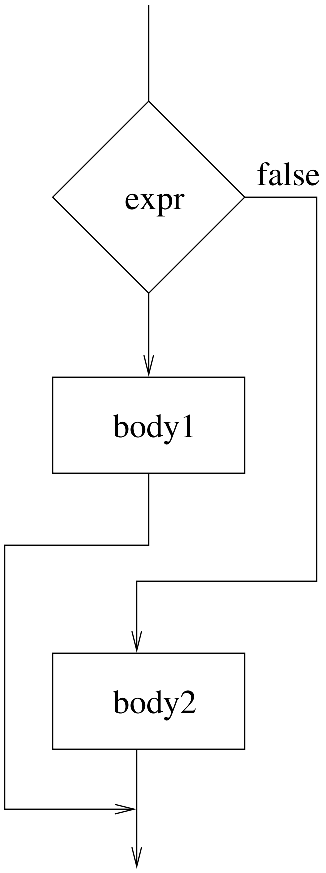
Note that in the assembly language code, the jump is made if we will *not* execute the body; therefore, the jump statement chosen tests if the expr evaluates to false.

```
main()
{
   int  a, b, c;

   if (a <= 17) {
     a = a - b;
     c++;
   }
}
```

```
     PUBLIC _main
     _TEXT  SEGMENT
     _a$ = -4
     _b$ = -8
     _c$ = -12
     _main  PROC NEAR

     ; 3    : {

        push   ebp
        mov ebp, esp
        sub esp, 12
        push   ebx
        push   esi
        push   edi

     ; 4    :  int  a, b, c;
     ; 5    :
     ; 6    :  if (a <= 17) {

        cmp DWORD PTR _a$[ebp], 17
        jg  $L28

     ; 7    :    a = a - b;

        xor eax, eax       ; 0 -> eax
        sub eax, DWORD PTR _b$[ebp]
        neg eax            ; b -> eax
        sub DWORD PTR _a$[ebp], eax

     ; 8    :    c++;

        inc DWORD PTR _c$[ebp]
     $L28:
     $L24:

     ; 9    :  }
     ; 10   : }

        pop edi
        pop esi
        pop ebx
        leave
        ret 0
     _main  ENDP
     _TEXT  ENDS
     END
```

if else

```
if( expr ) {
   body1
} else {
   body2
}
```



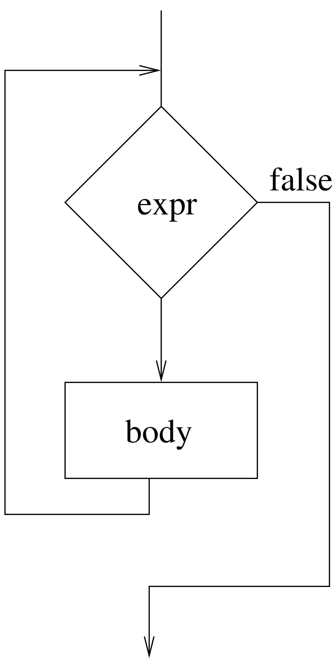

```
main()
{
   int  a, b, c;

   if (a <= 17) {
     a = a - b;
     c++;
   } else {
     b = a;
     c = b;
   }
}
```

```
    PUBLIC _main
    _TEXT  SEGMENT
    _a$ = -4
    _b$ = -8
    _c$ = -12
    _main  PROC NEAR

; 2    : void main(){

   push   ebp
   mov ebp, esp
   sub esp, 12
   push   ebx
   push   esi
   push   edi

; 3    :  int  a, b, c;
; 4    :
; 5    :  if (a <= 17) {

   cmp DWORD PTR _a$[ebp], 17
   jg  $L28

; 6    :    a = a - b;

   xor eax, eax
   sub eax, DWORD PTR _b$[ebp]
   neg eax
   sub DWORD PTR _a$[ebp], eax

; 7    :    c++;

   inc DWORD PTR _c$[ebp]

; 8    :  } else {

   jmp $L29
$L28:

; 9    :    b = a;

   mov eax, DWORD PTR _a$[ebp]
   mov DWORD PTR _b$[ebp], eax

; 10   :    c = b;

   mov eax, DWORD PTR _b$[ebp]
   mov DWORD PTR _c$[ebp], eax
$L29:
$L24:
```

```
; 11   :  }
; 12   : }

    pop edi
    pop esi
    pop ebx
    leave
    ret 0
_main  ENDP
_TEXT  ENDS
END
```

while loop

```
while( expr ) {
   body
}
```



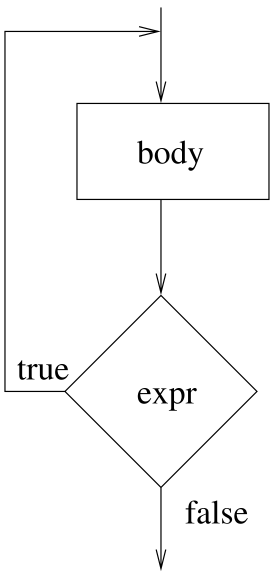

```
main()
{
   int  a, b, c;

   while (a <= 17) {
     a = a - b;
     c++;
   }
}
```

```
PUBLIC _main
_TEXT  SEGMENT
_a$ = -4
_b$ = -8
_c$ = -12
_main  PROC NEAR

; 2    : {

   push   ebp
   mov ebp, esp
   sub esp, 12
   push   ebx
   push   esi
   push   edi
$L29:

; 3    :  int  a, b, c;
; 4    :
; 5    :  while (a <= 17) {

   cmp DWORD PTR _a$[ebp], 17
   jg  $L30

; 6    :    a = a - b;

   xor eax, eax
   sub eax, DWORD PTR _b$[ebp]
   neg eax
   sub DWORD PTR _a$[ebp], eax

; 7    :    c++;

   inc DWORD PTR _c$[ebp]

; 8    :  }

   jmp $L29
$L30:
$L24:

; 9    : }

   pop edi
```

```
    pop esi
    pop ebx
    leave
    ret 0
_main  ENDP
_TEXT  ENDS
END
```

do loop

```
do {
   body
} while( expr );
```



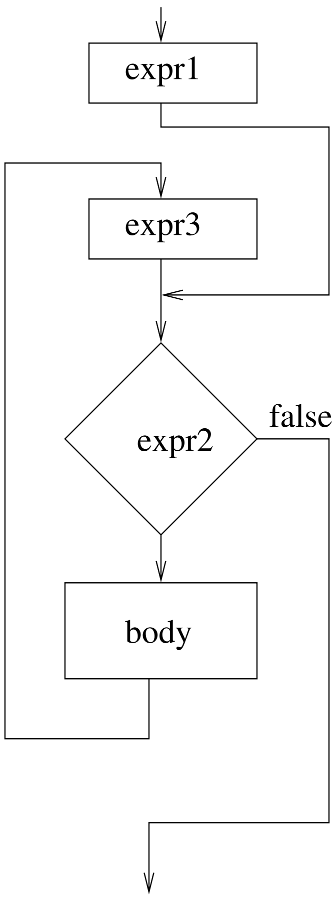

```
main()
{
   int  a, b, c;

   do  {
     a = a - b;
     c++;
   } while (a <= 17);
}
```

```
PUBLIC _main
_TEXT  SEGMENT
_a$ = -4
_b$ = -8
_c$ = -12
_main  PROC NEAR

; 2    : {

   push   ebp
   mov ebp, esp
   sub esp, 12
   push   ebx
   push   esi
   push   edi
$L28:

; 3    :  int  a, b, c;
; 4    :
; 5    :  do  {
; 6    :  a = a - b;

   xor eax, eax
   sub eax, DWORD PTR _b$[ebp]
   neg eax
   sub DWORD PTR _a$[ebp], eax

; 7    :  c++;

   inc DWORD PTR _c$[ebp]
$L29:

; 8    :  } while (a <= 17);

   cmp DWORD PTR _a$[ebp], 17
   jle $L28
$L30:
$L24:

; 9    : }

   pop edi
   pop esi
   pop ebx
   leave
   ret 0
_main  ENDP
_TEXT  ENDS
```

```
END
```

for loop

```
for( expr1; expr2; expr3 {
   body
}
```




```
main()
{
   int  a, b, c;
   int i;

   for (i = 1; i <= 17; i++) {
      a = a - b;
      c++;
   }
}
```

```
PUBLIC _main
_TEXT  SEGMENT
_a$ = -4
_b$ = -8
_c$ = -12
_i$ = -16
_main  PROC NEAR

; 3    : {

   push   ebp
   mov ebp, esp
   sub esp, 16
   push   ebx
   push   esi
   push   edi

; 4    :  int  a, b, c;
; 5    :  int i;
; 6    :
; 7    :  for (i = 1; i <= 17; ++i) {

   mov DWORD PTR _i$[ebp], 1
   jmp $L29
$L30:
   inc DWORD PTR _i$[ebp]
$L29:
   cmp DWORD PTR _i$[ebp], 17
   jg  $L31

; 8    :  a = a - b;

   xor eax, eax
   sub eax, DWORD PTR _b$[ebp]
   neg eax
   sub DWORD PTR _a$[ebp], eax

; 9    :  c++;

   inc DWORD PTR _c$[ebp]

; 10   :  }

   jmp $L30
$L31:
```

```
$L24:

; 11   : }

    pop edi
    pop esi
    pop ebx
    leave
    ret 0
_main  ENDP
_TEXT  ENDS
END
```

switch

Switch statements are implemented differently depending on the number of branches (case statements) in the switch structure.

In the following example, the number of branches is small and the compiler puts the test variable on the stack at -12[ebp] and uses a sequence of cmp and jump statements.

```
main()
{
   int i;
   int j;

   switch(i) {
   case 1: j = 1; break;
   case 2: j = 2; break;
   case 3: j = 3; break;
   default: j = 4;
   }
}
```

```
PUBLIC   _main
_TEXT SEGMENT
_i$ = -4
_j$ = -8
_main PROC NEAR

; 2    : {

    push  ebp
    mov   ebp, esp
    sub   esp, 12
    push  ebx
    push  esi
    push  edi

; 3    :    int i;
; 4    :    int j;
; 5    :
; 6    :    switch(i) {

    mov   eax, DWORD PTR _i$[ebp]
    mov   DWORD PTR -12+[ebp], eax
    jmp   $L27
$L31:

; 7    :    case 1: j = 1; break;

    mov   DWORD PTR _j$[ebp], 1
    jmp   $L28
$L32:

; 8    :    case 2: j = 2; break;

    mov   DWORD PTR _j$[ebp], 2
    jmp   $L28
$L33:

; 9    :    case 3: j = 3; break;

    mov   DWORD PTR _j$[ebp], 3
    jmp   $L28
$L34:

; 10   :    default: j = 4;

    mov   DWORD PTR _j$[ebp], 4

; 11   :    }

    jmp   $L28
$L27:
    cmp   DWORD PTR -12+[ebp], 1
```

```
     je $L31
     cmp   DWORD PTR -12+[ebp], 2
     je $L32
     cmp   DWORD PTR -12+[ebp], 3
     je $L33
     jmp   $L34
   $L28:
   $L24:

   ; 12   : }

     pop   edi
     pop   esi
     pop   ebx
     leave
     ret   0
   _main ENDP
   _TEXT ENDS
   END
```

The following example, which has a few more branches, uses a simple jump table to determine which branch to take. This code also fills an area of the stack from -76[ebp] to -13[ebp] with alternating ones and zeros (0xcccccccc). I do not know why this is done. It does not appear to accomplish anything.

```
int main()
{
   int i;
   int j;

   switch(i) {
   case 1: j = 1; break;
   case 3: j = 3; break;
   case 8: j = 8; break;
   case 6: j = 6; break;
   case 2: j = 2; break;
   case 7: j = 7; break;
   case 4: j = 4; break;
   default: j = 9; break;
   }
}
```

```
   PUBLIC   _main
   ;  COMDAT _main
   _TEXT SEGMENT
   _i$ = -4
   _j$ = -8
   _main PROC NEAR

   ; 2    : {

     push  ebp
     mov   ebp, esp
     sub   esp, 76
     push  ebx
     push  esi
     push  edi
     lea   edi, DWORD PTR [ebp-76]
     mov   ecx, 19
     mov   eax, -858993460   ; ccccccccH
     rep stosd

   ; 3    :    int i;
   ; 4    :    int j;
   ; 5    :
   ; 6    :    switch(i) {

     mov   eax, DWORD PTR _i$[ebp]
     mov   DWORD PTR -12+[ebp], eax
     mov   ecx, DWORD PTR -12+[ebp]
     sub   ecx, 1
     mov   DWORD PTR -12+[ebp], ecx
     cmp   DWORD PTR -12+[ebp], 7
     ja SHORT $L44
     mov   edx, DWORD PTR -12+[ebp]
     jmp   DWORD PTR $L49[edx*4]
   $L37:

   ; 7    :    case 1: j = 1; break;

     mov   DWORD PTR _j$[ebp], 1
     jmp   SHORT $L34
   $L38:

   ; 8    :    case 3: j = 3; break;

```

```
    mov   DWORD PTR _j$[ebp], 3
    jmp   SHORT $L34
$L39:

; 9    :    case 8: j = 8; break;

    mov   DWORD PTR _j$[ebp], 8
    jmp   SHORT $L34
$L40:

; 10   :    case 6: j = 6; break;

    mov   DWORD PTR _j$[ebp], 6
    jmp   SHORT $L34
$L41:

; 11   :    case 2: j = 2; break;

    mov   DWORD PTR _j$[ebp], 2
    jmp   SHORT $L34
$L42:

; 12   :    case 7: j = 7; break;

    mov   DWORD PTR _j$[ebp], 7
    jmp   SHORT $L34
$L43:

; 13   :    case 4: j = 4; break;

    mov   DWORD PTR _j$[ebp], 4
    jmp   SHORT $L34
$L44:

; 14   :    default: j = 9; break;

    mov   DWORD PTR _j$[ebp], 9
$L34:

; 16   : }

    pop   edi
    pop   esi
    pop   ebx
    mov   esp, ebp
    pop   ebp
    ret   0
$L49:
    DD $L37  ; case 1
    DD $L41  ; case 2
    DD $L38  ; case 3
    DD $L43  ; case 4
    DD $L44  ; case 5 - default
    DD $L40  ; case 6
    DD $L42  ; case 7
    DD $L39  ; case 8
_main ENDP
_TEXT ENDS
END
```

In the next example, the values in the the case statements are not are not close together, so the compiler uses a two stage jump table. One table hold an index into the second table which lists the location to jump to.

```
int main()
{
   int i;
   int j;

   switch(i) {
   case 10: j = 1; break;
   case 33: j = 3; break;
   case 85: j = 8; break;
   case 66: j = 6; break;
   case 20: j = 2; break;
   case 79: j = 7; break;
   case 41: j = 4; break;
   default: j = 9; break;
   }
}
```

```
PUBLIC   _main
;  COMDAT _main
_TEXT SEGMENT
_i$ = -4
```

```
_j$ = -8
_main PROC NEAR
; 2    : {
    push  ebp
    mov   ebp, esp
    sub   esp, 76
    push  ebx
    push  esi
    push  edi
    lea   edi, DWORD PTR [ebp-76]
    mov   ecx, 19
    mov   eax, -858993460 ;ccccccccH
    rep stosd
; 3    :    int i;
; 4    :    int j;
; 5    :
; 6    :    switch(i) {
    mov   eax, DWORD PTR _i$[ebp]
    mov   DWORD PTR -12+[ebp], eax
    mov   ecx, DWORD PTR -12+[ebp]
    sub   ecx, 10
    mov   DWORD PTR -12+[ebp], ecx
    cmp   DWORD PTR -12+[ebp], 75
    ja SHORT $L44
    mov   eax, DWORD PTR -12+[ebp]
    xor   edx, edx
    mov   dl, BYTE PTR $L49[eax]
    jmp   DWORD PTR $L50[edx*4]
$L37:
; 7    :    case 10: j = 1; break;
    mov   DWORD PTR _j$[ebp], 1
    jmp   SHORT $L34
$L38:
; 8    :    case 33: j = 3; break;
    mov   DWORD PTR _j$[ebp], 3
    jmp   SHORT $L34
$L39:
; 9    :    case 85: j = 8; break;
    mov   DWORD PTR _j$[ebp], 8
    jmp   SHORT $L34
$L40:
; 10   :    case 66: j = 6; break;
    mov   DWORD PTR _j$[ebp], 6
    jmp   SHORT $L34
$L41:
; 11   :    case 20: j = 2; break;
    mov   DWORD PTR _j$[ebp], 2
    jmp   SHORT $L34
$L42:
; 12   :    case 79: j = 7; break;
    mov   DWORD PTR _j$[ebp], 7
    jmp   SHORT $L34
$L43:
; 13   :    case 41: j = 4; break;
    mov   DWORD PTR _j$[ebp], 4
    jmp   SHORT $L34
$L44:
; 14   :    default: j = 9; break;
    mov   DWORD PTR _j$[ebp], 9
$L34:
; 16   : }
    pop   edi
    pop   esi
    pop   ebx
    mov   esp, ebp
    pop   ebp
    ret   0
$L50:
```

```
    DD $L37  ; entry 0 - case 10
    DD $L41  ; case  20
    DD $L38  ; case  33
    DD $L43  ; case  41
    DD $L40  ; case  66
    DD $L42  ; case  79
    DD $L39  ; case  85
    DD $L44  ; entry 7, default
$L49:
    DB 0 ; 10
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 1  ; 20
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 2 ; 33
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 3  ; 41
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 4  ; 66
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 5 ; 79
    DB 7
    DB 7
    DB 7
    DB 7
    DB 7
    DB 6 ; 85
_main ENDP
_TEXT ENDS
END
```

break, continue

```
void main()
{
   int  a, b;
   int i;

   for (i = 1; i <= 17; i++) {
     if (a == 0) continue;
     if (b == 0) break;
   }

   while (i <= 17) {
     if (a == 0) continue;
     if (b == 0) break;
   }

   do {
     if (a == 0) continue;
     if (b == 0) break;
   } while (i <= 17);
}
```

```
PUBLIC   _main
_TEXT SEGMENT
_a$ = -4
_b$ = -8
_i$ = -12
_main PROC NEAR

; 3    : {

   push  ebp
   mov   ebp, esp
   sub   esp, 12
   push  ebx
   push  esi
   push  edi

; 4    :    int  a, b;
; 5    :    int i;
; 6    :
; 7    :    for (i = 1; i <= 17; i++) {

   mov   DWORD PTR _i$[ebp], 1
   jmp   $L28
$L29:
   inc   DWORD PTR _i$[ebp]
$L28:
   cmp   DWORD PTR _i$[ebp], 17
   jg $L30

; 8    :      if (a == 0) continue;

   cmp   DWORD PTR _a$[ebp], 0
   jne   $L31
   jmp   $L29
$L31:

; 9    :      if (b == 0) break;

   cmp   DWORD PTR _b$[ebp], 0
   jne   $L32
   jmp   $L30
$L32:

; 10   :    }

   jmp   $L29
$L30:
$L34:

; 11   :
; 12   :         while (i <= 17) {

   cmp   DWORD PTR _i$[ebp], 17
   jg $L35

; 13   :      if (a == 0) continue;

   cmp   DWORD PTR _a$[ebp], 0
   jne   $L36
   jmp   $L34
$L36:

; 14   :      if (b == 0) break;

```

```
    cmp   DWORD PTR _b$[ebp], 0
    jne   $L37
    jmp   $L35
$L37:
; 15   :         }
    jmp   $L34
$L35:
$L38:
; 16   :
; 17   :    do {
; 18   :      if (a == 0) continue;
    cmp   DWORD PTR _a$[ebp], 0
    jne   $L41
    jmp   $L39
$L41:
; 19   :      if (b == 0) break;
    cmp   DWORD PTR _b$[ebp], 0
    jne   $L42
    jmp   $L40
$L42:
$L39:
; 20   :         } while (i <= 17);
    cmp   DWORD PTR _i$[ebp], 17
    jle   $L38
$L40:
$L24:
; 21   : }
    pop   edi
    pop   esi
    pop   ebx
    leave
    ret   0
_main ENDP
_TEXT ENDS
END
```

Floating Point Arithmetic

The floating point arithmetic unit, called the floating point unit (FPU), contains eight registers which function as a stack machine. The register which is currently at the top of the stack is referred to as ST. All floating point instructions specify operands relative to ST.

Floating Point Arithmetic Instructions

Instruction	Operands	Notes
finit		initialize the FPU
fld	mem	Push data onto the FPU stack
fldz		Push 0.0 onto the FPU stack
fst	mem	Store ST (top of stack) to memory
fstp	mem	Store ST to memory and pop ST
fadd	mem	Add data to ST and store result in ST
fsub	mem	Subtract data from ST and store result in ST
fsubr	mem	Subtract ST from data and store result in ST
fmul	mem	Multiply data with ST and store result in ST
fdiv	mem	Divide ST by data and store result in ST
fdivr	mem	Divide data by ST and store result in ST
frndint		Round ST to an integer and store result in ST
fchs		Change the sign of ST (ST = -ST)
fcom	mem	Compare floating point values, setting FPU flags C0–C3
ftst		Compare ST to 0.0, setting FPU flags C0–C3
ftsw	AX	Copy FPU status word to AX

The following example was generated using the Linux gcc compiler[1]; however, to avoid confusion, I changed the instruction names and the operand order to be consistent with Intel's Manual and other x86 C compilers.

```
# include <stdio.h>

int main(void)
{
   float pi=3.14159;
   float r = 0.25;

   printf("%f\n", pi*r*r);
   return 0;
}
```

```
        .file "area.c"
        .version "01.01"
   gcc2_compiled.:
   .section .rodata
   .LC0:
        .string  "%f\n"
   .text
        .align 4
   .globl main
        .type main,@function
   main:
        push %ebp
        mov  %ebp,%esp
        sub  %esp,8
        mov  -4(%ebp),1078530000 ! 0x40490fd0
        mov  -8(%ebp),1048576000 ! 0x3e800000
        fld  -4(%ebp)
        fmul -8(%ebp)
        fmul -8(%ebp)
        sub  %esp,8
        fstp (%esp)
        push $.LC0
        call printf
        add %esp,12
        xor %eax,%eax
```

[1] "gcc -S foo.c" will generate assembly code in foo.s

```
	jmp .L1
	.p2align 4,,7
.L1:
	leave
	ret
.Lfe1:
	.size main,.Lfe1-main
	.ident
	"GCC: (GNU) egcs-2.91.66 19990314/Linux"
```