Addressing Modes, Subprograms and Stack Frames

Including Recursion

16-bit Addressing Modes

- 16-bit x86 provides the following addressing modes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Segment</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>[Disp] or Variable</td>
<td>DS</td>
<td>MOV AX,[081H]</td>
</tr>
<tr>
<td>Indexed</td>
<td>[DI]</td>
<td>DS</td>
<td>MOV AX,[DI]</td>
</tr>
<tr>
<td>Based</td>
<td>[BX]</td>
<td>DS</td>
<td>MOV AX,[BX]</td>
</tr>
<tr>
<td>Indexed</td>
<td>[SI]</td>
<td>DS</td>
<td>MOV AX,[SI]</td>
</tr>
<tr>
<td>Displaced</td>
<td>[SI+disp]</td>
<td>DS</td>
<td>MOV AX,[SI+4]</td>
</tr>
<tr>
<td>Displaced</td>
<td>[DI+disp]</td>
<td>DS</td>
<td>MOV AX,[DI+4]</td>
</tr>
<tr>
<td>Displaced</td>
<td>[BX+disp]</td>
<td>DS</td>
<td>MOV AX,[BX+4]</td>
</tr>
<tr>
<td>Displaced</td>
<td>[BP+disp]</td>
<td>SS</td>
<td>MOV AX,[BP-4]</td>
</tr>
</tbody>
</table>

8086 Addressing Modes - 2

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Segment</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based</td>
<td>[BX+DI]</td>
<td>DS</td>
<td>MOV AX,[BX+DI]</td>
</tr>
<tr>
<td>Indexed</td>
<td>[BX+SI]</td>
<td>DS</td>
<td>MOV AX,[BX+SI]</td>
</tr>
<tr>
<td>Based</td>
<td>[BP+DI]</td>
<td>SS</td>
<td>MOV AX,[BP+DI]</td>
</tr>
<tr>
<td>Indexed</td>
<td>[BP+SI]</td>
<td>SS</td>
<td>MOV AX,[BP+SI]</td>
</tr>
<tr>
<td>Based</td>
<td>[BX+DI+disp]</td>
<td>DS</td>
<td>MOV AX,[BX+DI+4]</td>
</tr>
<tr>
<td>Indexed</td>
<td>[BX+SI+disp]</td>
<td>DS</td>
<td>MOV AX,[BX+SI+4]</td>
</tr>
<tr>
<td>Based</td>
<td>[BP+DI+disp]</td>
<td>SS</td>
<td>MOV AX,[BP+DI+4]</td>
</tr>
<tr>
<td>Indexed</td>
<td>[BP+SI+disp]</td>
<td>SS</td>
<td>MOV AX,[BP+SI+4]</td>
</tr>
</tbody>
</table>

String | Format | Example |
--------|--------|---------|
| [SI]   | SS     | LODSB   |

Operations | Format | Example |
------------|--------|---------|
| [DI]      | ES     | STOSB   |

• For displacement addressing, displacements can be either:
  - 8-bit signed -128 to +127
  - 16-bit unsigned 0 to 65,535 (or 32 bits for 386+)

• Names of addressing modes are not important and are not standardized
• The term "register-indirect" for any based, indexed or indirect modes
• You do NOT need to memorize or understand 16-bit addressing

32 bit Indirect and SIB Addressing

- Any 32-bit general register (including EAX, ECX, EDX, ESP) can be used as an index or base register
  - e.g. MOV EDX, [EAX]
  - "scaled-indexed-based addressing" (SIB) allows any 2 general purpose registers together with a scaling factor to be used to compute an address
    - mov edx,[ebx+4*ecx]
    - mov [edi+2*esp+100h],cx
    - mov al,[ebx+edx-2]

• Scaling factor must be 1, 2, 4, or 8
• Optional displacement can be used
• Allows easy access to arrays of bytes, words, doublewords, or quadwords with indices rather than offsets
• Determination of default segment can get very complex if ESP and / or EBP involved

Indirect Operands

- When the offset of variable is loaded into a base or index register, the register becomes a POINTER to the variable
  - string DB 'This is a string'
  - ... mov ebx,string
  - add ebx,4
  - mov al,[ebx] ; now what's in al?
- The same character can be loaded into al with:
  - mov ebx,string
  - mov al,[ebx+4]
- Or
  - mov ebx,4
  - mov al,[ebx+string]
- Or
  - mov al,[string+4]

Indirect Operands with Displacement

- A register is added to a displacement to obtain the "effective address"
- Displacement is either a number or label whose offset is known at assembly time
  - There are several syntactic notations
    - hits read 100
      - ... mov ebx,20 ; address 8th element at 4*(n-1)
      - mov edx, [ebx+hits]; these all do the same thing
      - mov edx, [hits+ebx]
      - mov edx, [hits+string]
    - Note that concatenation of symbols in assembler implies addition
    - We can also use the base of the array in the index reg:
      - mov ebx,hits
      - mov edx, [ebx + 28]
Indirect Operands with Displacement

- The above example constructs a string in memory. It gets a bit tricky doing it a word at a time because of back-words storage:

```assembly
mov byte [edi], 'H'
mov byte [edi+1], 'o'
mov byte [edi+2], 'w'
mov byte [edi+3], 'd'
mov byte [edi+4], 'y'
```

16 and 32 bit Indirect Operands

- We will discuss indirect addressing in more detail with array operations.
- The 16-bit indirect addressing rules are very complex compared to 32 bit rules.
- For 32-bit code just consider:
  - Any general purpose register including esp can be used as an index register.
  - With SIB addressing any 2 registers can be combined to form an effective address because the scaling factor is 1, 2, 4 or 8 (e.g., `mov al, [eax+esi*4]` is the same as `mov al, [eax+esi*8]`).
  - A displacement can be used with a single register or with a SIB expression:
    - `mov al, [ebx+4]`
    - `mov al, [ebx+4*ecx+12]`.

Why SIB Addressing?

- SIB addressing means that you can use registers to index into arrays with a logical index value.

```assembly
array resd 1024
```

```assembly
mov ecx, 1024 ; number of elements
mov edx, anarray ; base address
sub eax, eax ; zero accumulator
```

```assembly
L1:
add eax, [edx + ecx * 4]
loop L1
```

Subroutines: CALL and RET

- HLLs have many words for the same concept:
  - Subroutine
  - Procedure
  - Function
  - Subprogram
- All involve a transfer of control, normally followed by a return to the point of departure.
- Often a function is considered to be a procedure that returns a value. Some languages make a syntactic distinction, others do not.
- In C everything is a function but some functions return values.
- At the assembler level there is no syntactic difference.
- We will treat procedures and functions as the same thing in assembly language; both involve a transfer of program control to a set of instructions with CALL or INT that ends with a RET or IRET.
- When a RET (IRET) is executed control returns to the instruction following the CALL (INT) instruction.
- A value-returning function typically returns a value in a register while a procedure or non-value returning function does not.

CALL

- Syntax:
  - CALL dest
  - CALL FAR dest
  - CALL NEAR dest

- Semantics:
  - (NEAR)
    1. ESP ← ESP − 4
    2. [SS:ESP] ← EIP
    3. EIP ← dest
      - CS ← HIGH 16 dest;
      - EIP ← LOW 32 dest
  - (FAR)
    1. SP ← SP − 2
    2. [SS:SP] ← IP
    3. IP ← dest
      - CS ← HIGH dest;
      - IP ← LOW dest

- Flags:
  - ODITSZAPC unchanged

- Operands:
  - label reg mem

- Notes:
  1. Precise syntax of far call varies with assembler.
  2. Destination address can be indirect:
    - CALL [ebx]
    - CALL FAR [ebx]
    - CALL [foo]
    - CALL FAR [foo]
  3. Because flags are unaffected flags can be used as parameters.
  4. Indirect calls allow runtime computation of function addresses.

Hardware basis for object-oriented programming (OOP) and polymorphic namespaces.
RET (RETurn) and RETF (RETurn Far)

- **Purpose:**
  Return from a subroutine, popping IP or CS:IP from the stack. RET forces a far return. An immediate value as an operand adjusts the stack pointer by that amount after popping the return address.

- **Syntax:**
  1. RET
  2. RETF
  3. RET imm
  4. RETF imm

- **Semantics:**
  1. IP < - [SS:SP] eip < [ss:esp]
     SP < SP + 2 esp < esp + 4
  2. IP < - [SS:SP] eip < [ss:esp]
     SP < SP + 4 esp < esp + 6
  3. IP < - [SS:SP] eip < [ss:esp]
     SP < SP + 2 + imm esp < esp + 4 + imm
  4. IP < - [SS:SP] eip < [ss:esp]
     SP < SP + 4 + imm esp < esp + 6 + imm

RET Notes

- Even though data can only be stacked as words or dwords, the stack adjustment operand is specified in bytes. Therefore it must always be EVEN.
- Some assemblers do not allow the RETF mnemonic. Instead, the assembler uses
  PROC foo NEAR
  Or PROC foo FAR
  to determine what kind of RET to assemble

RET imm

- The immediate value is used to "clean parameters off the stack" by adjusting the stack pointer

```assembly
push offset hits ;put two params on the stack
pushd 128 ;stack
call processData

... processData:
  push ebp ; set up stack frame
  mov ebp, esp ; first param is at ebp+8
  mov ecx, [ebp+8]
  mov esi, [ebp+12]
  ...
  pop ebp ; restore caller's ebp
ret 8 ; return
```

Stack Frames and Parameters

- EBP (the base pointer) might be better called the Frame Pointer because that is what is used for addressing a stack frame
- The key to re-entrant and recursive programming is passing of parameters on the stack AND the use of the stack to store local variables
- A stack frame is a structure used to store and access parameters, return addresses, saved registers and local variables

Stack Frame or Activation Record

<table>
<thead>
<tr>
<th>Parameters passed by Caller</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Pointer -&gt; Caller's Frame Pointer</td>
<td></td>
</tr>
<tr>
<td>- Local (Automatic) Variables</td>
<td></td>
</tr>
<tr>
<td>Stack Pointer -&gt;</td>
<td>Saved Registers and Temporary Storage</td>
</tr>
</tbody>
</table>

Passing Parameters

- Three techniques for passing parameters in assembly language:
  1. use registers;
  2. use global variables;
  3. use the stack
- The Stack method is the most flexible technique
  - Stack technique is used by high-level languages.
  - Register parameters are fastest technique
    - Many C/C++ compilers allow _fastcall to specify register parameters
  - But you can get into trouble with recursive or reentrant routines if you use register parameters
Re-entrant Code

- Code that cannot call itself, or cannot be interrupted and called by another process before it is finished running is called non-reentrant code.
- Code that can call itself, or can be interrupted and called by another process before it is done is called reentrant code.
- Re-entrant code is essential to operating systems programming.

Passing Parameters on the Stack

- Using the stack for parameter passing generally means that the parameters are stacked with PUSHes and then the procedure is called.
- To call a function with two parameters:
  - push dwordvar1
  - push dwordvar2
  - call func
- So at entry to func the stack looks like this:

  | ESP+8 | dwordvar1 |
  | ESP+4 | dwordvar2 |
  | ESP   | return address |

Accessing Parameters

- We could do this:
  - pop edi; save return address in a register
  - pop eax; save dwordvar2
  - pop ebx; save dwordvar1
- But this method has a host of problems. Alternate Plan B:
  - mov eax,[esp+4]; load dwordvar2
  - mov ebx,[esp+8]; load dwordvar1
- But what happens if at some point we push eax?
  - All of the stack offsets would change.
- This is where the base pointer comes into play.
  - EBP by default is stack (SS) relative just as ESP is.
  - The convention is that the callee loads ebp to point at something on the stack.
  - The “something” is the caller’s saved base pointer.

Using the Base Pointer

- Here’s our call with two parameters:
  - push dwordvar1
  - push dwordvar2
  - call func
- Inside func we have:
  - push ebp; save caller’s base pointer
  - mov ebp, esp; load our frame pointer

  | EBP+12 | dwordvar1 |
  | EBP+8  | dwordvar2 |
  | EBP+4  | return address |
  | EBP    | saved EBP |

Cleaning up the Stack

- Who cleans up the stack after the call and removes the parameters?
- It seems a bit difficult for the callee to clean things up - after all we have to pop eip (return address) before we can adjust the stack pointer.
- But in most high-level languages and in most asm programs the stack is cleaned by the callee.
- The exception are C and C++ - the CALLER cleans up the stack.
- The instruction RET immed does a return and removes immed additional bytes after poppin the return offset into eip.
Note that in the Intel architecture it is more efficient to allow the callee to do the cleanup. ret imm is just as fast as ret. But if functions are allowed to have variable numbers of parameters (e.g., C printf or scanf) then the callee cannot clean up the stack but the value for imm is unknown.

The code executed at the beginning of the function is called the prologue and the code executed at end before returning is called the epilogue.

Cleanup by the CALLEE

push dvar1
foo:
push dvar2
    push ebp
    mov ebp, esp
    ...; continue
    ...; do some processing
pop ebp
ret 8

Cleanup by the CALLER

push dvar1
foo:
push dvar2
    push ebp
    mov ebp, esp
    add esp, 8
    ...; do some processing
    ...; continue
    pop ebp
    ret

Simple Subroutine Example: sub3

Here’s what the program does:

F:\\em\c335\paulcarter\\ws-asm\sub3
1) Enter an integer number (0 to quit): 101
2) Enter an integer number (0 to quit): 102
3) Enter an integer number (0 to quit): 103
4) Enter an integer number (0 to quit): 0
The sum is 306

sub3:1

#include “asm_io.inc”
segment .data
sum dd 0
segment .bss
input resd 1

; algorithm in C code
; i = 1;
; sum = 0;
; while( get_int(i, &input), input != 0 ) {
;    sum += input;
;    i++;
; }
; print_sum(num);

segment .text
    global _asm_main
_asm_main:
    enter 0, 0 ; setup routine
    pusha
sub3:2

while_loop:
push edx
    ; save i on stack
push word input
    ; push address on input on stack
stack
    call get_int
add esp, 8
    ; remove i and input from stack
mov eax, [input]
cmp eax, 0
je end_while
add [sum], eax
    ; sum += input
inc edx
jmp short while_loop
end_while:
push dword [sum]
call print_sum
pop eax
    ; remove [sum] from stack
popa
leave
ret

sub3:3

get_int – prompt and read integer

Parameters (in order pushed on stack)

; number of input (at [ebp + 12])
; address of word to store input into (at [ebp + 8])

segment .data
prompt db ”Enter an integer (0 to quit): ”, 0

segment .text
    global _asm_main
_asm_main:
    enter 0, 0 ; setup routine
    pusha
prompt db ”) Enter an integer (0 to quit): ”, 0
get_int:
push ebp
    mov esp, [ebp + 12]
call print_int
    mov esp, [ebp + 12]
call print_string
    call read_int
    mov eax, [ebp + 8]
    ; store input into memory
    pop ebp
    ret
```plaintext
sub3:4

; Parameter:
; sum to print out at [ebp+8]
; Note: destroys value of eax

segment .data
result db "The sum is ", 0

segment .text
print_sum:
push ebp
mov ebp, esp

mov eax, result
call print_string

mov eax, [ebp+8]
call print_int

call print_nl

pop ebp
ret

Local Variables on the Stack

- After a call, and the usual push ebp and mov ebp, esp the stack looks like this (with three parameters):

  | ESP+16 | EBP+16 |
  | ESP+12 | EBP+12 |
  | ESP+8  | EBP+8  |
  | ESP+4  | EBP+4  |
  | ESP    |  EBP   |

- Space for local variables can reserved on the stack by SUB ESP, n where n is the number of bytes needed

%defines for Stack Parameters

- It is a good practice to use %defines to address local variables and stack parameters

  %define PARAM1    dword [ebp+16]
  %define PARAM2    dword [ebp+12]
  %define PARAM3    dword [ebp+8]
  %define LOCAL1    dword [ebp-4]
  %define LOCAL2    dword [ebp-8]
  %define LOCAL3    dword [ebp-12]

- This makes code easier to write and read (especially when meaningful variable names are used, unlike the example above)

  MOV edx, LOCAL1
  MOV edx, [ebp-4]

Example

- After sub esp, 12 the stack looks like this:

  | ESP+28 | Param 1      | EBP+16 |
  | ESP+24 | Param 2      | EBP+12 |
  | ESP+20 | Param 3      | EBP+8  |
  | ESP+16 | Return ebp   | EBP+4  |
  | ESP+12 | Caller's ebp | EBP    |
  | ESP+8  | Local 1      | EBP+4  |
  | ESP+4  | Local 2      | EBP+8  |
  | ESP    | Local 3      | EBP+12 |

- Using this scheme, local variables on the stack are addressed using negative offsets from ebp
- Parameters are addressed using positive offsets from ebp
- Note that parameter addresses do NOT change relative to EBP but they do change relative to ESP

Reference and Value Parameters

- The previous example had two value parameters a and b and one reference parameter (a pointer)
- In order to access the data pointed by a reference parameters you must first load the pointer into an index register
- Then access memory via the index register

  Some other architectures allow memory-indirect addressing
```
Returning Values from Functions

- Function return values for simple types are almost universal:
  - bytes: AL
  - words: AX
  - dwords: EAX (or DX:AX in 16 bit code)
  - qwords: EDX:EAX
  - floats: ST(0) [top of x87 register stack]

- Note that the issue is not so much type as size
  - Both ints and pointers are returned in EAX

- For sizes other than those listed above, functions either:
  - (A) return a pointer to a data structure
  - (B) return a data structure on the stack.

MyProc as a Value-Returning Function

```assembly
MyProc:    ; return a^2 - b^2 in eax
    ;define a  dword [ebp+12]
    ;define b  dword [ebp+8]
    ;define tmp dword [ebp-4]
    push eax
    push ebx
    mov eax, b
    imul eax
    mov tmp, eax
    mov eax, a
    imul eax
    sub eax, tmp
    mov eax, esp
    mov esp, ebp
    pop ebp
    ret 8
```

The PUSHA Bug

- Watch out for the PUSHA bug in value returning functions:
  ```assembly
  myfunc:
    pusha ; save registers
    ... mov eax, retval
    popa
    ret
  ```

ENTER and LEAVE

- ENTER and LEAVE are CISC instructions designed to simplify prologue and epilogue code
- ENTER Syntax
  ```assembly
  ENTER imm16, imm8
  ```
  - Where imm16 is the number of bytes to reserve on the stack for locals and imm8 is the lexical nesting level
  - The imm8 parameter is 0 for languages such as C, C++ or Java that do not have nested function scope and could be non-zero for languages such as Javascript, Ada and Pascal
  ```assembly
  ;ENTER 0, 0
  push esp
  mov esp, esp
  ;ENTER 12, 0
  push esp
  mov esp, esp
  sub esp, 12
  ```

ENTER and LEAVE Synttacts

- LEAVE Syntax
  ```assembly
  LEAVE
  ```
  ```assembly
  ;LEAVE
  mov esp, ebp ; deallocate any locals
  pop ebp
  ```
  - Although designed for HLL compiler writers, ENTER and LEAVE are rarely used because the ENTER instruction is inefficiently implemented
  - ENTER is 11, 15, or 15 + 2 * imm8 clocks

Recursion

- A recursive function is a function that calls itself
- Why is recursion interesting?
  - Recursive function theory is another model of computation
  - Anything that can be computed by a Turing machine can be computed by a recursive function

- There are some languages whose primary control structure is recursion
  - In general any iterative solution to a problem can also be expressed as a recursive solution
  - Recursive programs provide elegant and simple solutions to some apparently complex problems
Example

• The factorial function is easily expressed in both iterative and recursive forms

```c
int factorial(int N) {
    int product = 1;
    for (int j=1; j<=N; j++)
        product *= j;
    return product;
}
```

```c
int factorial(int N) {
    if (N<=1)
        return 1
    else
        return N * factorial(N-1);
}
```

Optimized to use ESP

• If you can be sure that esp will not change during function execution you can bypass stack frame setup and cleanup

```asm
segment .text
_factorial:
    push ebp
    mov ebp,esp
    mov eax,[ebp+8]    ; eax = n
    cmp eax,10      ; if N > 10
    ja recursion   ; recurse
    sub eax,8      ; is N = 8?
    je case8 : yes, return 1 and 1
    dec eax         ; if N = 9 then eax is now 1
    je case9 : so return 3 and 0
    mov eax,0      ; N was 10
    mov ebx,2      ; so return 0 and 2
    jmp short exit
    case8:
        mov eax,1      ; return 1 and 1
        mov ebx,1
        jmp short exit
    case9:
        mov eax,3      ; return 3 and 0
        mov ebx,0
        jmp short exit
    recursion:
        sub eax,3      ; n <= n-3
        push eax        ; push arg to mult35
        call mult35
        add esp, 4      ; clean parameter
        inc eax      ; k was returned in eax, add 1 to it
        jmp short exit
    exit:
        pop ebp        ; and return to caller
        ret
```

Recursive Factorial in Asm

```asm
global _factorial
_factorial:
    push ebp
    mov ebp,esp
    mov eax,[ebp+8]  ; eax = n
    cmp eax,1        ; if n <= 1 then
    jbe short L1      ; return 1
    dec eax         ; push n-1 on the
    push eax        ; stack and
    call _factorial ; so eax = fact(n-1)
    add esp, 4      ; remove parameter n-1
    mov edx:eax = eax * [ebp+8] ; edx:eax = eax * [ebp+8]
    jmp short L2
L1:
    mov eax, 1
L2:
    pop ebp
    ret
```

The 3-5 Function

• Any number N >= 8 can be expressed in the form N = (3*K) + (5*M)

- Algorithm to compute M and K
  - if n = 8 then K <= -1 and M <= -1
  - elseif n = 9 then K <= -3 and M <= -0
  - elseif n = 10 then K <= 0 and M <= -2
  - else compute K and M for N-3
    - add 1 to K
    - return K and M
````

```
mult35:1
```

```
mult35:2
```