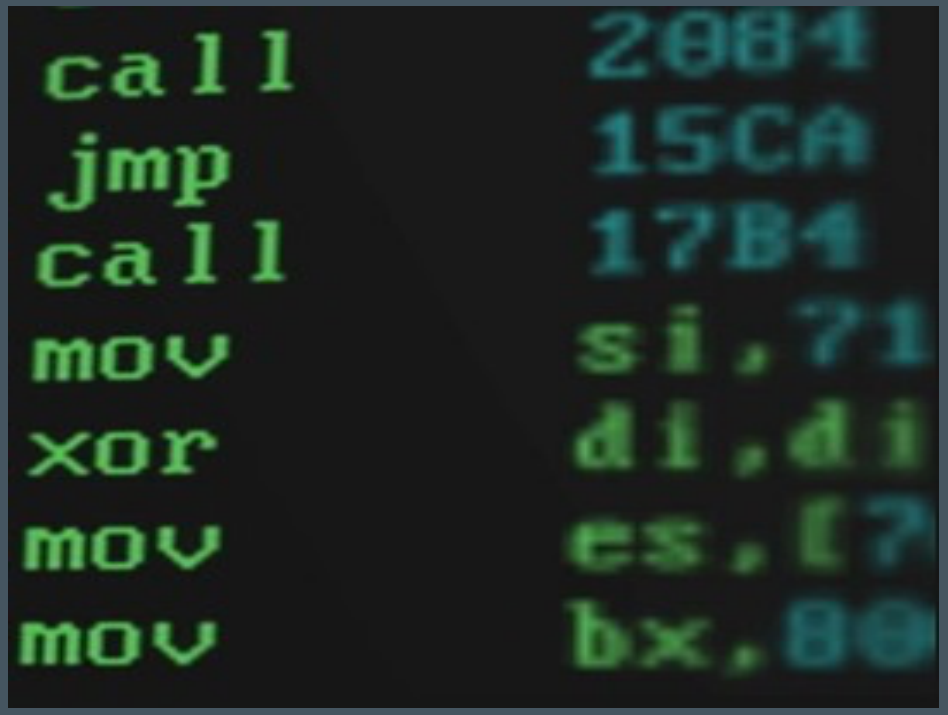
COMP6016 Assembly Language Programming II



Dr MUHAMMAD HILMI KAMARUDIN

Instruction Operands

OXFORD BROOKES UNIVERSITY

- Operands are used to specify the values that an instruction will act on.
- Most instructions require source operands plus a single destination operand. In most cases these must be specified, but in some cases the operands can be implicit
- In the case of two operand instructions, the first is normally the destination (where the results is going to be stored) and the second is the source (a value to be stored, or the address at which the value can be found)
- There are 3 classes of operands -
- Immediate operands – a constant value or expression that can be immediately evaluated
- Register operands – the value is contained in a GPR
- Memory operands – a memory address to be used to retrieve or store the information. NB either source or destination can be a memory operand, but not both

Instruction Operands

Type	Example	Equivalent C/C++ Statement
Immediate	mov eax,42	eax = 42
	imul ebx,11h	ebx *= 0x11
	xor dl,55h	dl ^= 0x55
	add esi,8	esi += 8
Register	mov eax,ebx	eax = ebx
	inc ecx	ecx += 1
	add ebx,esi	ebx += esi
	mul ebx	edx:eax = eax * ebx
Memory	mov eax,[ebx]	eax = *ebx
	add eax,[val1]	eax += *val1
	or ecx,[ebx+esi]	ecx |= *(ebx + esi)
	sub word ptr [edi],12	*(short*)edi -= 12

From: Kusswurm, D (2014) *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*

Variables & constants

OXFORD BROOKES UNIVERSITY

- Assembler does not have conventional variables, instead we allocate memory and optionally initialise it
- Allocating memory is done via labels – the label becomes, in effect, the variable name
- Memory can then be initialised, or left unitialised

db	0x55	; a byte with value 0x55
db	0x55,0x56,0x57	; three bytes in succession
db	'a',0x55	; two byes, character constants are OK
db	'hello',13,10,'$'	; so are string constants
dw	0x1234	; create memory in word sized chunks and initialise ; it to 0x34 0x12
dw	'a'	; 0x61 0x00 (it's just a number)
dw	'ab'	; 0x61 0x62 (character constant)
dw	'abc'	; 0x61 0x62 0x63 0x00 (string)
dd	0x12345678	; allocate memory in double word chunks and set ; it to 0x78 0x56 0x34 0x12

Variables & constants

OXFORD BROOKES UNIVERSITY

- Uninitialised memory -

```
buffer:         resb    64                      ;
reserve 64 bytes
wordvar:        resw    1                       ;
reserve a word
Realarray:       resq    10                     ;
array of ten reals
```

- Constants can be defined using the equ directive

```
Maxstudents             equ         100
```

Key instructions – data transfer

OXFORD BROOKES UNIVERSITY

- mov – copy data to/from a GPR or memory location to/from a GPR or memory location. (Remember you cannot move from a memory location to a memory location). The mov instruction will not change any flags

```
mov     ecx,msg          ; copy the value of msg to ecx
mov     ebx,1              ; set ebx to the value 1
```

- push – push a GPR, memory location or immediate value onto the stack
- pop – take the top most item off the stack and store it in the specified GPR or memory location

```
push    eax    ; push the value stored in eax onto the stack
pop     ebp    ; take the top value from the stack and store
it in ebp
```

- xchg – swap data between two GPRs or a GPR and a memory location

```
xchg    ax, bx         ; Put AX in BX and BX in AX
xchg    memory, ax     ; Put "memory" in AX and AX in "memory"
```

Key instructions – binary arithmetic

OXFORD BROOKES UNIVERSITY

Instruction	Description
`sub eax, Ox10`	Subtracts Ox10 from EAX
`add eax, ebx`	Adds EBX to EAX and stores the result in EAX
`inc edx`	Increments EDX by 1
`dec ecx`	Decrements ECX by 1

Key instructions – binary arithmetic

OXFORD BROOKES UNIVERSITY

- Multiplication and division – predefined register. The command : instruction, plus the value that the register will be multiplied or divided by. The format: mul *value,* div *value*
- By default, mul *value* instruction always multiplies eax by *value.* Result stored as a 64-bit value across two registers: EDX and EAX. EDX stores the most significant 32 bits of the operation, while EAX stores the least significant 32 bits.

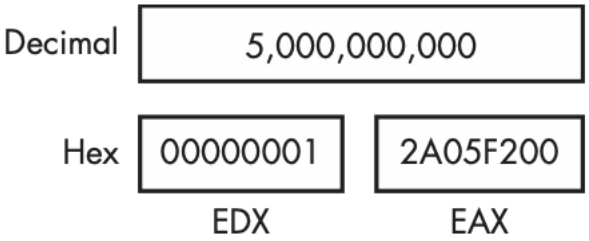


Figure: Multiplication result stored across EDX and EAX registers

- The div *value* instruction does the same thing as mul, except in the opposite direction: It divides the 64 bits across EDX and EAX by *value*. There- fore, the EDX and EAX registers must be set up appropriately before the division occurs. The result of the division operation is stored in EAX, and the remainder is stored in EDX.

OXFORD BROOKES UNIVERSITY

Instruction	Description
`mul 0x50`	Multiplies EAX by 0x50 and stores the result in EDX:EAX
`div 0x75`	Divides EDX:EAX by 0x75 and stores the result in EAX and the remainder in EDX

Table: Multiplication and Division Instruction Examples

OXFORD BROOKES UNIVERSITY

Key instructions – binary arithmetic

- NOP- does nothing

- Execution simply proceeds to the next instruction. It is actually a pseudonym for xchg eax, eax.

- Thus instruction is 0x90, commonly used in a NOP sled for buffer overflow attacks, when attackers don't have perfect control of their exploitation.

These NOPs let us locate our malicious shellcode away from the program's "core" so that we have a smaller risk of it crashing. These NOPs let us pad our shellcode basically.

Key instructions – data comparison

OXFORD BROOKES UNIVERSITY

- cmp – compare the two operands by subtracting the source operand from the destination and then set the status flags. The results of the subtraction are discarded.

```
cmp ecx, edx          ; compare the values in ecx and edx
```

- cmpxchg – compares the contents of register AL, AX, or EAX with the destination operand and performs an exchange based on the results. (i.e. compare AL/AX/EAX with destination operand, if they are equal, copy the source operand into the destination operand

```
cmpxchg [foo], edx
; compare the value at the address foo with EAX and if they are
equal, put the value of edx into the memory address foo
```

OXFORD BROOKES UNIVERSITY

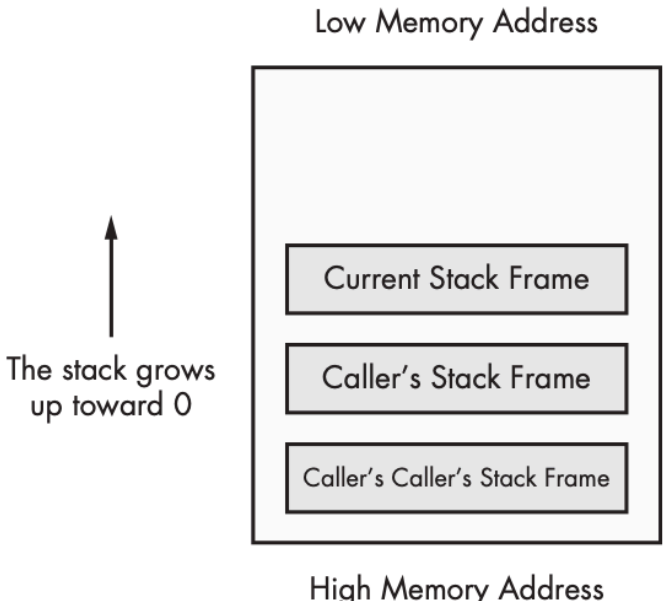
Key instructions – logical operators

- and, or, xor – calculate the bitwise AND, OR, or XOR respectively of the source and destination operands.
- not – calculate the 1's compliment of the specified operand, does not affect status flags
- test – calculate the bitwise AND of the source and destination operands and discard the results (essentially set the status flags without changing any values)

OXFORD BROOKES UNIVERSITY

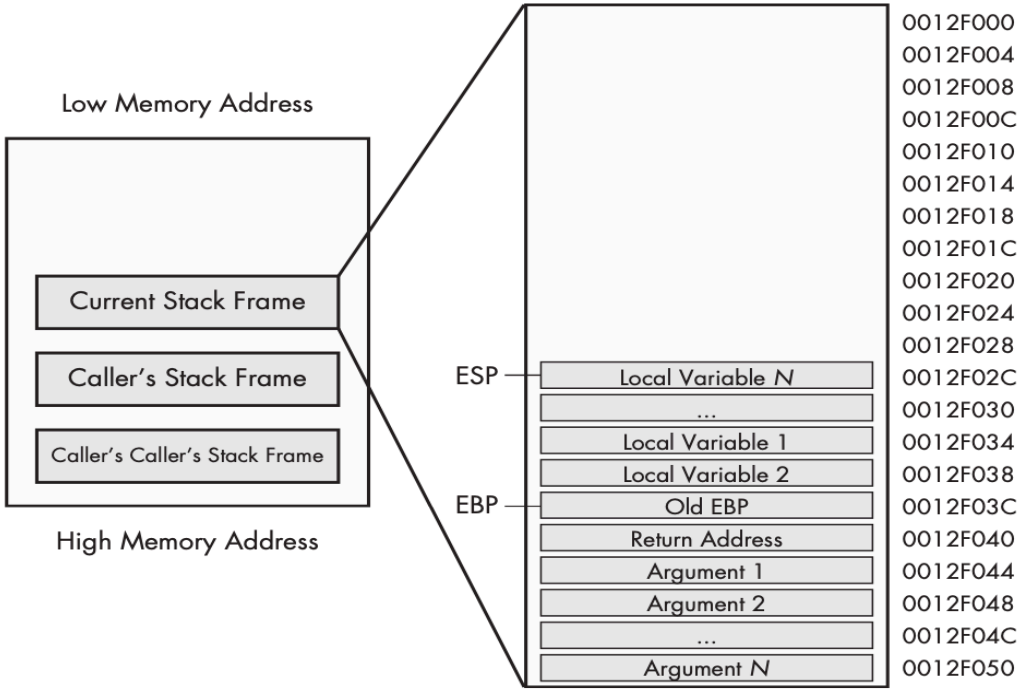
The Stack

- Memory for functions, local variables, and flow control is stored in a *stack*- data structure characterized by pushing and popping.
- A stack is a last in, first out (LIFO) structure
- The x86 architecture has built-in support for a stack mechanism. The register support includes the ESP and EBP registers
- ESP is the stack pointer and typically contains a memory address that points to the top of stack.
- EBP is the base pointer that stays consistent within a given function



OXFORD BROOKES UNIVERSITY

The Stack Layout



- The stack is allocated in a top-down fashion. Figure above shows a dissection of one of the individual stack frames from previous figure. The memory locations of individual items are also displayed
- ESP would point to the top of the stack, which is the memory address 0x12F02C. EBP would be set to 0x12F03C throughout the duration of the function, so that the local variables and arguments can be referenced using EBP.

OXFORD BROOKES UNIVERSITY

- Group- discuss stackexercise.asm

OXFORD BROOKES UNIVERSITY

Function Calls

- Are portions of code within a program that perform a specific task and that are relatively independent of the remaining code.
- The main code calls and temporarily transfers execution to functions before returning to the main code.
- Many functions contain a *prologue*—a few lines of code at the start of the function. The prologue prepares the stack and registers for use within the function.
- In the same vein, an *epilogue* at the end of a function restores the stack and registers to their state before the function was called.

OXFORD BROOKES UNIVERSITY

Function Calls continue

- Following list summarizes the flow of the most common implementation for function calls:

1) Arguments are placed on the stack using push instructions.

2) A function is called using call memory_location. This causes the current instruction address (that is, the contents of the EIP register) to be pushed onto the stack. This address will be used to return to the main code when the function is finished. When the function begins, EIP is set to memory_location (the start of the function).

3) Through the use of a function prologue, space is allocated on the stack for local variables and EBP (the base pointer) is pushed onto the stack. This is done to save EBP for the calling function.

4) The function performs its work.

5) Through the use of a function epilogue, the stack is restored. ESP is adjusted to free the local variables, and EBP is restored so that the call- ing function can address its variables properly. The leave instruction can be used as an epilogue because it sets ESP to equal EBP and pops EBP off the stack.

6) The function returns by calling the ret instruction. This pops the return address off the stack and into EIP, so that the program will continue exe- cuting from where the original call was made.

7) The stack is adjusted to remove the arguments that were sent, unless they'll be used again later.

OXFORD BROOKES UNIVERSITY

Conditionals

- *Conditionals* are instructions that perform the comparison.

- The two most popular conditional instructions are test and cmp. The test instruction is identical to the and instruction while The cmp instruction is identical to the sub instruction

- The test instruction only sets the flags. The zero flag (ZF) is typically the flag of interest after the test instruction. A test of something against itself is often used to check for NULL values. An example of this is test eax, eax.

- The cmp instruction is used only to set the flags. The zero flag and carry flag (CF) may be changed as a result of the cmp instruc- tion. Table below shows how the cmp instruction impacts the flags.

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0

Table: cmp Instruction and Flags

Branching

- *branch-* sequence of code that is conditionally executed depending on the flow of the program.
- The most popular way branching- *jump instructions- the simplest*
- The format jmp *location* causes the next instruction executed to be the one specified by the jmp. This is known as an *unconditional* jump, because execution will always transfer to the target location.
- This simple jump will not satisfy all of your branching needs. For example, the logical equivalent to an if statement isn't possible with a jmp. There is no if statement in assembly code

Conditional Jumps

OXFORD BROOKES UNIVERSITY

Conditional jumps use the flags to determine whether to jump or to proceed to the next instruction.

Instruction	Description
jz loc	Jump to specified location if ZF = 1.
jnz loc	Jump to specified location if ZF = 0
je loc	Same as jz, but commonly used after a cmp instruction. Jump will occur if the destination operand equals the source operand.
jne loc	Same as jnz, but commonly used after a cmp. Jump will occur if the destination operand is not equal to the source operand.
jg loc	Performs signed comparison jump after a cmp if the destination operand is greater than the source operand.
jge loc	Performs signed comparison jump after a cmp if the destination operand is greater than or equal to the source operand.
jl loc	Performs signed comparison jump after a cmp if the destination operand is less than the source operand.
jle loc	Performs signed comparison jump after a cmp if the destination operand is less than or equal to the source operand.
jecxz loc	Jump to location If ECX = 0

```
Malware authors will usually
make jumps to random locations
to waste the analyst's time…
```

OXFORD BROOKES UNIVERSITY

Procedures

- Procedures are implemented using labels
- The `call` function is used to call a procedure. This pushes the RIP onto the stack.
- The `ret` function pops the RIP off the stack and goes back to the line that called the procedure
- Parameter passing is done via memory or registers

```
myadd:      mov rax, rcx
            add rax, rdx
            ret

main:       mov rcx, 7
            mov rdx, 8
            call myadd
```

OXFORD BROOKES UNIVERSITY

Procedures – push and pop

- We only have a limited amount of registers that we can use
- If we are using registers in our procedures, we need to save them before we use them and restore them afterwards. This is done using `push` and `pop`

```
myproc:       push rax
              mov rax, rdx
              add rax,rdx
              mov rcx, rax
              pop rax
              ret

main:  mov rax, 7
              call myadd
```

OXFORD BROOKES UNIVERSITY

libasm_io

- IO is problematic in assembler
- It is best to use a set of libraries – in our case we are using Joey libasm_io which is a 64bit version that work in SASM

- This is designed to work on a variety of POSIX platforms and handles basic terminal and file IO
- Functions are not guaranteed register safe, so remember to push and pop your registers before and after use

libasm_io – simple IO

```
; Test program for io
; nasm -g -I -f elf64 test-io.asm
; gcc -g test-io.o –no-pie -o test-io

; You'll need to point this to your version of the library
%include %include
"/home/malware/asm/joey_lib_io_v6_release.asm"


global main

section .data
        echo_request:   db      "Please enter a number: ",0
        echo_number:    db      "The number you entered was:",0
        echo_bye:       db      "Goodbye!",0
```

libasm_io – simple IO

```
section .text

main:
      mov rbp, rsp ; We have these three lines
                   for compatibility only

      push rbp
      mov rbp, rsp
      sub rsp,32

      ; We load up the message to ask for a
number
      mov rdi, QWORD echo_request
      call print_string_new
```

libasm_io – simple IO

```
; we read in an int
call read_int_new
push rax                ; we save the int for
                        safeties sake

; we print out our message
mov rdi, QWORD echo_number
call print_string_new
call print_nl_new

; we copy back the int we saved and print it
pop rdi
call print_int_new
call print_nl_new
```

libasm_io – simple IO

```
    ; we print our goodbye message
    mov rdi, QWORD echo_bye
    call print_string_new
    call print_nl_new

    ; and these lines are for
compatability
    add rsp, 32
    pop rbp

    ret
```

OXFORD BROOKES UNIVERSITY

Practical exercises in group: