**Assembly Survival Guide**

**Registers**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 64-bit | 32-bit | 16-bit | High 8-bit | Low 8-bit | Possible Purpose | Volatile\* |
| rax | eax | ax | ah | al | Return | y |
| rbx | ebx | bx | bh | bl |  |  |
| rcx | ecx | cx | ch | cl | Counter | y |
| rdx | edx | dx | dh | dl |  | y |
| rsi | esi | si |  |  | Source |  |
| rdi | edi | di |  |  | Destination |  |
| rbp | ebp | bp |  |  | Base pointer |  |
| rsp | esp | sp |  |  | Stack pointer |  |
| rip | eip | ip |  |  | Instruction Pointer |  |
| r8 |  |  |  |  | Additional 64-bit general purpose registers | y |
| r9 |  |  |  |  | y |
| r10 |  |  |  |  | y |
| r11 |  |  |  |  | y |
| r12 |  |  |  |  |  |
| r13 |  |  |  |  |  |
| r14 |  |  |  |  |  |
| r15 |  |  |  |  |  |

\* Volatile Registers (don't need to be preserved by callee)

**Data Sizes**

|  |  |
| --- | --- |
| byte | Smallest addressable unit – 8 bits |
| word | 2 bytes – 16 bits |
| dword | Double word – 4 bytes – 32 bits |
| qword | Quad word – 8 bytes – 64 bits |

|  |  |
| --- | --- |
| mov al, byte [rsi] | Copy a single byte (1 byte) |
| mov ah, byte [rsi] | Copy a single byte on the high side of ax |
| mov ax, word [rsi] | Copy a word (2 bytes) |
| mov eax, dword [rcx] | Copy a dword (4 bytes) |
| mov rax, qword [rsi] | Copy a qword (8 bytes) |

**Debugging**

|  |  |
| --- | --- |
| Item | Description |
| int3 | Sets break point in assembly program. Runs only in GDB |
| asm(“int3”); |  |
| gdb lab1  (gdb) run  ...  (gdb) quit | Overall process to debug assembly |
| (gdb) s | Single step |
| (gdb) n | Step over |
| (gdb) c | Continue running |
| (gdb) break myfunc | Sets breakpoint at given function |
| (gdb) break myfunc.label | Break on a particular label |
| (gdb) delete 1 | Deletes 1st breakpoint |
| (gdb) info break | Show breakpoints |
| (gdb) r | Refresh screen (same as CTRL-L) |
| (gdb) x/i $rip | Show memory at $rip |
|  |  |

Debugging on Microsoft requires the “Solutions Platform” drop down list to be “x86”. To step through assembly choose Tools->Options->Debugging->General check box “Show disassembly if source not available”

Changing projects in VS “Set as Startup Project”

**Instructions**

|  |  |  |
| --- | --- | --- |
| Op | Mnemonic | Purpose |
| mov | mov dest, src | Move/Copy data from src to dest |
| movzx | movzx dest, src | Copies data from src to dest then zero extend the remaining space |
| movsx | movsx dest, src | Copies data from src to dest then places 1’s in the remaining upper part of the register thus preserving the “signed” attributes |
| xchg | xchg a, b | Exchanges values in a and b |
| lea | lea reg1, [reg2] | Load effective address or computes address of reg2 and copies to reg1 |
| add | add dest, src | Add value in src to dest |
| sub | sub dest, src | Subtracts value in src from dest |
| mul | mul reg | Multiples reg \* register rax (eax, ax, etc) stores the result in reg |
| div | div reg | Divides reg/rax stores result in rax, and remainder in rdx |
| inc | inc reg | Add 1 to register reg |
| dec | dec reg | Decrement 1 from register reg |
| push | push reg | Push the value in register reg onto the stack |
| pop | pop reg | Pop a value off the stack into register reg |
| shl | shl reg, val | Shift left the register by val |
| shr | shr reg, val | Shift right the register by val |
| and | and reg1, reg2 | And to registers, store result in reg1 |
| or | or reg1, reg2 | Or to registers, store result in reg1 |
| not | not reg | Invert bits in reg |
| xor | xor reg, reg | Xor reg with reg |
| shl | shl reg, val | Left shift reg by val (in bits) |
| shr | shr reg, val | Right shift reg by val (in bits) |
| rol | rol reg, val | Rotate left reg by val (in bits), add bits that fall off to the right side |
| ror | ror reg, val | Rotate right reg by val (in bits), add bits that fall off to the left side |
| pushf | pushf | Push flags to the stack (in order to get them) |
| popf | popf | Restore flags back to the flags register |
| jmp | jmp label | Unconditional jump to label |
| call | call label | Equivalent to push rip then jump to label, indicates a function call |
| ret | ret | Pops return pointer (rip) off stack and jumps to it |
| cmp | cmp reg, val | Compare two values by subtraction then sets flags to indicate whether the values were equal, or if one was larger. Flags set by this instruction: CF, OF, SF, ZF, AF, and PF |
| test | test reg, reg | Compares two value by bitwise AND. The SF, PF, and ZF get set by this operation. Often used to test whether or not a register is 0 |
| jz | jz label | Jump if zero flag is set |
| jnz | jnz label | Jump if zero flag is not set |
| ja | ja label | Jump if above (if operand compared previously is greater) |
| jb | jb label | Jump if below (if operand compared previously is lesser) |
| jc | jc label | Jump if carry flag set |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| std | std | The direction flag has been set |
| cld | cld | Clear the direction flag, continue operations |
| cpuid | cpuid | The value in EAX at the time of the CPUID call determines what information comes back  ◦0 -> Vender ID String - stored in EBX/EDX/ECX  ◦1 -> Returns a bitfield containing supported features |
| rdtsc | rdtsc | Low bits of result are stored into EAX/high bits in EDX |

**Examples**

|  |  |
| --- | --- |
| Example | Description |
| mov rax, 4 | Move value 4 into rax |
| mov rax, 0x01 | Move hex value 1 into rax |
| mov rax, rbx | Move value from register rbx into rax |
| mov rax, [rbx] | Dereference pointer register rbx and move its value into rax |
| mov [rax], rbx | Move the value from register rbx into the dereference pointer rax |
| mov rax, qword [rbx + 8] | Copy a quad word (8 bytes) into rax |
| mov al, byte [rsi] | Copy a single byte into al |
| mov eax, dword [rcx] | Copy a double word (4 bytes) into eax |
| mov rax, qword [rsi] | Copy a quad word (8 bytes) into rax |
| movzx rax, cl | Copy cl to rax, everything above al is set to zero |
| movzx rax, byte [rsi + 5] |  |
| xchg al, ah | Exchange (switch) data in al and ah |
| lea rax, [rdx + 8]  mov rcx, [rax] | Adding 8 to the address of rdx points and store in rax  Dereference the address in rax and move contents to rcx |
| add rax, 1 | Adds 1 to the existing value in rax |
| sub rax, 1 | Subtracts 1 from the existing value in rax |
| mul ecx | Multiples eax \* ecx |
| mov rdx, 0  mov rax, 10  mov rcx, 2  div rcx | Division process, clear reminder register rdx, load numerator rax, load denominator rcx. Thus rax/rcx, result value of 5 is stored in rax |
| inc rax | Add 1 to rax |
| dec rax | Subtract 1 from rax |
| sub rsp, 8  mov [rsp], rcx  ...  mov [rsp], rax  add rsp, 8 | Making space on the stack pointer for a 8 byte register  Saving rcx onto the stack (these two command are push)  Returning the value on the stack to rax  Removing the space on the stack (these two commands are pop) |
| push rcx | Push rax on the stack (FILO) |
| pop rax | Pop a value off the stack into rax |
| mov cl, -1  movsx rax, cl | Movsx preserves the sign, rax is now -1 |
| shl rax, 1 | Shift bits in rax left by 1, effectively multiplying by 2 |
| shr rax, 2 | Shift bits in rax right by 2, effectively dividing by 4 |
| and rax, rcx | And rax and rcx, store result in rax |
| or rax, rcx | Or rax and rcx, store result in rax |
| not rax | Inverts bits in register |
| xor rax, rax | Zero out rax |
| rol rax, 1 | Rotate bits left by 1, left bit that fell off added to right side |
| ror rax, 1 | Rotate bits right by 1, right bit that fell off added to left side |
| rol rax, 8 | Rotating a string in rax, each character is 1 byte or 8 bits, to rotate four characterss it would be 32 bits |
| xor rax, rcx  ror rax, 8 | Deobfuscating an xor’d string in rax with a key in rcx, rotating rax so that each character can be ‘translated’ |
| cmp rax, 0 | If rax is zero then ZF set |
| pushf  pop ax  or ax 2048  push ax  popf | The process to set the Overflag which is 2^11 or 2048 |

**Examples Loops**

|  |  |
| --- | --- |
| xor rax, rax  test rax, rax  jz .myend | Loop by decrementing rax and testing. Check if result of operation is zero, then jump to .end  test will set the zero flag |
| mov rcx, 10  .myloop:  ...  dec rcx  test rcx, rcx  jnz .myloop | Loop by decrementing rcx to 0. Creates a loop that decrements rcx down to zero, tests rcx for zero, if so sets the zero flag, jump if rcx is not 0 yet |
| mov rcx, 10  xor rax, rax  .continue:  add rax, rcx  loop .continue  ret | Loop Macro: expects rcx to be loaded with the counter, then performs: dec rcx, test rcx, rcx, jnz <target> |
| xor rax, rax  . myloop:  add eax, [rdi]  lea rdi, [rdi + 4]  dec rsi  test rsi, rsi  jnz .myloop | Loop through an array of integers that rdi points to. Lea calculates the next 4 bytes (integers are 4 bytes) and puts it into rdi. Thus it increments the pointer by 4 bytes. Rsi is the number of elements in the array. |
| xor rax, rax  .myloop:  mov bl, [rdi]  cmp bl, 0  je .myend  inc rax  lea rdi, [rdi + 1]  jmp .myloop  .myend: | Finds the length of the provided NULL-terminated string that rdi points to.  Mov 1 byte into bl, compare bl to Null, if equal jump.  Increment rax. Lea calculate the next byte of the string and move it into rdi (increments the pointer by 1) |

**String Instructions**

|  |  |  |
| --- | --- | --- |
| Instr | Usage | Description |
| scas | scac(b/w/d/q) | Scan String: Scans a string located as RDI for the value found in RAX/EAX/AX/AL (depending on size used), and increments the pointer |
| stos | stos(b/w/d/q) | Store String: Initializes the string located at RDI to the value pointer at by RAX/EAX/AX/AL (depending on size used) and increments the pointer |
| lods | lods(b/w/d/q) | Load String: Copies the value from RSI into RAX/EAX/AX/AL, and increments the pointer |
| movs | movs(b/w/d/q) | Move String: Copies data from RSI into RDI, and increments both pointers |
| cmps | cmps(b/w/d/q) | Compare String: Compares the values stored at RSI and RDI, and increments the pointer, updating the RFLAGS (or EFLAGS) register with the result |

|  |  |
| --- | --- |
| Register | Information |
| RSI (or ESI, in x86) | Treated as a pointer to the beginning of the "source" |
| RDI (or EDI, in x86) | Treated as a pointer to the beginning of the "destination" |
| RCX (or ECX, in x86) | Assumed to hold the count, if needed |
| RAX (or EAX, in x86) | Assumed to hold the value to evaluate |

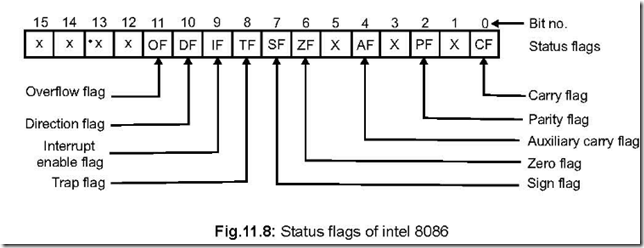
|  |  |  |  |
| --- | --- | --- | --- |
| Prefixes | Information | Exit Condition 1 | Exit Condition 2  When found |
| REP | Continue the action rcx times | ECX = 0 | (none) |
| REPNE | Continue the action rcx times, or until FLAGS register indicates operands are equal | ECX = 0 | ZF = 0 |
| REPE | Continue the action rcx times, or until FLAGS register indicates the operands were not equal | ECX = 0 | ZF = 1 |

|  |  |
| --- | --- |
| Direction Flag | Information |
| std | Changes the direction flag |
| cld | Clear the direction flag, continues operations |

**Examples String Instructions**

|  |  |
| --- | --- |
| .continue:  lodsb  xor al, 0x5f  stosb  loop .continue | XOR Byte by byte:  Register esi (or rsi) points to a string to load and loads 1 byte into al. Al is xor’d with a value (in this case 0x5f). Then al is loaded into edi (or rdi). Both loadsb and stosb auto increment their pointers |
| rep movsb | Copy string in rsi to rdi for rcx bytes (rcx represents the buffer size) |
| xor rax, rax  mov rcx, 20  mov rdi, \_my\_string\_buf  rep stosb | Zero out rdi (pointer to a string) with rax (which is zero) until rcx is zero |
|  |  |

**Flags**



|  |  |
| --- | --- |
| flags | 16 bits |
| eflags | 32 bits |
| rflags | 64 bits |

|  |  |
| --- | --- |
| Flag | Description |
| Zero Flag (ZF) | Set when an operation that sets the zero flag produces a zero - includes arithmetic and bitshift operations |
| Carry Flag (CF) | set when an arithmetic borrow or carry occurs during add/sub - e.g., the result of an add would have set bit 33 (in x86), or bit 65 (in x86\_64) also set with some bitshift operations (such as when a bit falls off the end in a shl/shr) |
| Overflow Flag (OF) | Indicates that sign bit of the result of an operation is different than the sign bits of the operands. Aka the carry flag for signed operations. |
| Sign Flag (SF) | Set to indicate the result of an operation is negative |

**Microsoft Conventions**

|  |  |
| --- | --- |
| myfunc(int a, int b);  \_myfunc@8:  ; do stuff  ret 8 | stdcall. The function called @8 so the return is ‘ret 8’ |
| \_myfunc@8:  mov eax, [esp + 4]  mov ecx, [esp + 8]  ; do stuff  ret 8 | stdcall pulling passed in variables off the stack. exp + 4 is parameter 1-above the return pointer, and exp + 8 is parameter 2 |
| \_caller:  push 2 ; arg 2  push 1 ; arg 1  ...  add esp, 8 ; clean up | cdecl (Microsoft and SysV calling convention), caller is responsible for cleanup, should have a “\_” underscore at the beginning of the name |
| \_third\_func:  xor eax, eax  xor ebx, ebx  mov edi, [esp + 8]  .mystart:  mov bl, [edi]  cmp bl, 0  je .myend  inc eax  lea edi, [edi + 1]  jmp .mystart  .myend:  ret | Using the following function definition, given a NULL-terminated string, and a max value for length, find and return the string's length. |
| push ebp ; save base pointer  mov ebp, esp ; set up stack  ...  pop ebp ; restore base pointer | Setting up a stack |

**Labs**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

**Other**

Params into a function (for 64 bits sysV)

1. RDI
2. RSI
3. RDX
4. RCX

Params for x86

* Parameters start at
* [esp + 4] ; last pushed on the sack
* [esp + 8] ; second to last pushed on the sack

struc MyStruct

.field1 resd 1 ; field1's size is 1 dword

.field2 resd 1 ; field2's size is 1 dword

.field3 resq 1 ; field3's size is 1 qword

endstruc

; ...

; This will be equivalent to: mov rax, [rdi+8]

mov rax, [rdi + MyStruct.field3]