Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

x86 Processor Architecture

Verified:
Olga Grosu asist. univ.

Belih Dmitrii

Moldova, May 2025

Contents

1 Questions **2**

1.1 Questions 2

1.1.1 Assembly Program Analysis 2

1.1.2 Floating-Point Assembly Program Analysis 7

1.2 Analysis of Assembly Code and Debugging Output 14

1.2.1 Source Code with Comments 14

1.2.2 Debugging Analysis Using `gdb` 17

1.2.3 Conclusion 19

1.3 Assembly Program Analysis and Debugging Part 2 19

1.3.1 Task Overview 19

1.3.2 Full Source Code with Comments 19

1.3.3 Debugging Analysis 24

1.3.4 Example Output 24

1.4 Assembly Program: Swapping Pairs in an Array 24

1.4.1 Task 24

1.4.2 Source Code with Comments 25

1.4.3 Explanation 27

1.4.4 Sample Output 27

1.4.5 Debugging Notes 27

1.5 Assembly Program: Copying a 16-bit Array into a 32-bit Array 28

1.5.1 Task 28

1.5.2 Source Code with Comments 28

1.5.3 Explanation 31

1.5.4 Sample Output 31

1.5.5 Debugging Tips 31

1.6 Procedure random string 31

2 Conclusion **38**

2.1 Conclusion 38

1 Questions

1.1 Questions

1.1.1 Assembly Program Analysis

Program Overview

This is an x86 assembly language program that demonstrates various addressing modes and data manipulation techniques. The program consists of a data section containing a variable named "alfa" and a text section containing the executable code.

Complete Assembly Code

Listing 1.1: *Complete Assembly Code*

```
section .data
 alfa dw 0, 0, 0 ; 3 words initialized to 0 (equivalent to WORD 3 DUP(?))

section .text
global _start

_start:
 mov ax, 17          ; Decimal representation
 mov ax, 0b10101     ; Binary representation (decimal 21)
 mov ax, 0b11        ; Binary representation (decimal 3)
 mov ax, 021o        ; Octal representation (decimal 17)
 mov ebx, alfa       ; Loads address of alfa into EBX
 lea ebx, [alfa]     ; Load Effective Address - same result as above
 mov [alfa], ax      ; Store value of AX at memory location alfa
 mov cx, ax          ; Copy value from AX to CX
 xchg ax, bx         ; Exchange values between AX and BX
 mov esi, 2          ; Set ESI to 2 for indexed addressing
 mov [alfa + esi], ax    ; Indexed addressing - stores AX at address (alfa + 2)

 ; Exit the program
```

```
21 mov eax, 1          ; sys_exit system call number
22 mov ebx, 0          ; exit code 0
23 int 0x80            ; system call interrupt
```

GDB Debugging Analysis

Starting the Program

```
Starting program: /home/dumas/Documents/Lab7/program
Breakpoint 1, _start () at program.asm:8
8           mov ax, 17
```

The program starts execution at the `_start` label. A breakpoint has been set at line 8, which contains the instruction `mov ax, 17`. This is the entry point of our program where execution begins.

Executing Instructions Step by Step

```
(gdb) stepi
9           mov ax, 0b10101      ; binary 10101
```

The instruction `mov ax, 17` is executed. This moves the decimal value 17 into the 16-bit AX register. The program counter advances to line 9. The `stepi` command in GDB executes a single assembly instruction and then stops again, allowing us to observe the effect of each instruction individually.

```
(gdb) stepi
10          mov ax, 0b11         ; binary 11
```

The instruction `mov ax, 0b10101` is executed. This moves the binary value 10101 (decimal 21) into the AX register, overwriting the previous value of 17. The program counter advances to line 10. Note that the binary representation is prefixed with `0b` to indicate it's a binary number.

```
(gdb) stepi
11          mov ax, 021o         ; octal 21
```

The instruction `mov ax, 0b11` is executed. This moves the binary value 11 (decimal 3) into the AX register, overwriting the previous value of 21. The program counter advances to line 11. This demonstrates another way to represent binary numbers in x86 assembly.

```
(gdb) stepi
12          mov ebx, alfa        ; offset of alfa (same as offset in original)
```

The instruction `mov ax, 021o` is executed. This moves the octal value 21 (denoted by the `o` suffix, decimal 17) into the AX register, overwriting the previous value of 3. The program counter advances to line 12. Assembly language allows us to express the same number in different bases for flexibility.

Register State

```
(gdb) info registers
eax            0x11                17
ecx            0x0                 0
edx            0x0                 0
ebx            0x0                 0
esp            0xffffcae0          0xffffcae0
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0x8048090           0x8048090 <_start+16>
eflags         0x202               [ IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x0                 0
```

The `info registers` command displays the current state of the CPU registers:

- The EAX register (which contains AX as its lower 16 bits) now holds the value 0x11 (decimal 17). This confirms that the octal value 021 (which is 17 in decimal) was successfully loaded into AX.
- Other general-purpose registers (ECX, EDX, EBX) are still at their initial values of 0.
- ESP (stack pointer) points to 0xffffcae0, which is the current top of the stack.
- EIP (instruction pointer) is at 0x8048090, which corresponds to the next instruction to be executed.
- The EFLAGS register has the IF (Interrupt Flag) set, indicating that interrupts are enabled.
- The segment registers (CS, SS, DS, ES, FS, GS) are set to their default values in a Linux process.

Memory Inspection

```
(gdb) x/3hw &alfa
0x8049000 <alfa>:     0x00000000     0x001c0000     0x00020000
```

The `x/3hw &alfa` command examines 3 half-words (16-bit values) starting at the address of the variable `alfa`:

- `alfa` is located at memory address 0x8049000 in the program's data section.
- The memory at this location contains three zero-initialized words as declared in the data section with `alfa dw 0, 0, 0`.
- Each "hw" represents a half-word (16 bits), so we see 3 half-words as defined in our data section.

Register Display

```
(gdb) display/x $ax
1: /x $ax = 0x11
```

The `display/x $ax` command sets up a continuous display of the AX register in hexadecimal format:

- The AX register currently contains 0x11 (decimal 17), which matches our expected value after executing the octal load instruction.
- This display will be updated automatically after each GDB command that could change the register value.

Program Structure and Comments

Data Section

```
section .data
 alfa dw 0, 0, 0 ; 3 words initialized to 0 (equivalent to WORD 3 DUP(?))
```

The data section declares a variable named "alfa" containing three 16-bit words (dw = define word), all initialized to 0. This creates a 6-byte storage area in memory that we'll reference later in our code.

Number Representation Instructions

```
mov ax, 17         ; Decimal representation
mov ax, 0b10101    ; Binary representation (decimal 21)
mov ax, 0b11       ; Binary representation (decimal 3)
mov ax, 021o       ; Octal representation (decimal 17)
```

The program demonstrates different ways to represent numbers in assembly:

- Decimal: Standard base-10 notation
- Binary: Prefixed with `0b`
- Octal: Suffixed with `o`

These instructions show that assembly language allows programmers to express the same numerical values in different number systems, which can be useful for different contexts.

Memory Addressing Instructions

```
mov ebx, alfa      ; Loads address of alfa into EBX
lea ebx, [alfa]    ; Load Effective Address - same result as above
mov [alfa], ax     ; Store value of AX at memory location alfa
mov [alfa + esi], ax   ; Indexed addressing
```

Various memory addressing modes are demonstrated:

- Direct addressing: Using the variable name directly loads its address
- LEA (Load Effective Address): Calculates the effective address but doesn't access memory
- Indirect addressing: Using square brackets to access the memory at a given address
- Indexed addressing: Adding an index register (ESI) to a base address to compute the target address

Register Operations

```
mov cx, ax         ; Copy value from AX to CX
xchg ax, bx        ; Exchange values between AX and BX
```

Different register manipulation operations are shown:

- Direct register-to-register copy with MOV
- Register value exchange with XCHG, which swaps the values in two registers in a single instruction

Program Flow and Execution

The execution has only progressed through the first few instructions, which mainly demonstrate loading different numeric literals into the AX register. The final value loaded before stopping was octal 21 (decimal 17), which is shown in the register output.

Had the program continued, it would have demonstrated more complex memory addressing modes and register manipulation operations before properly exiting via the system call at the end:

```
mov eax, 1         ; sys_exit
mov ebx, 0         ; exit code 0
int 0x80           ; system call interrupt
```

This sequence is the standard way to exit a program in Linux assembly:

1. Load system call number 1 (sys_exit) into EAX
2. Load the exit code 0 (indicating successful completion) into EBX
3. Trigger the interrupt 0x80 to invoke the Linux kernel's system call handler

Conclusion

This program serves as an educational example of assembly language programming, demonstrating:

- Various number representation formats (decimal, binary, octal)
- Different memory addressing modes
- Register manipulation operations
- Proper program termination through system calls

The GDB debugging session provides insight into the program's execution flow and the effect of each instruction on the processor's state.

1.1.2 Floating-Point Assembly Program Analysis

Program Overview

This x86 assembly program calculates the mathematical expression $z = a + b^2 - \frac{36/b^2}{1+25/b^2}$ using the FPU (Floating-Point Unit). The program uses the FPU stack to perform calculations with floating-point values defined in the data section.

Complete Assembly Code

Listing 1.2: *Floating-Point Assembly Code*

```
section .data
 a dd 5.0       ; Example value for a (float)
 b dd 2.0       ; Example value for b (float)
 z dd 0.0       ; Result storage
 const36 dd 36.0
 const25 dd 25.0
 const1 dd 1.0

section .text
global _start

_start:
 ; Load values
 fld dword [b]          ; ST0 = b
 fmul st0, st0          ; ST0 = b*b

 ; Calculate denominator part (1 + (25/(b*b)))
 fld dword [const25]  ; ST0 = 25, ST1 = b*b
 fdiv st0, st1          ; ST0 = 25/(b*b)
 fadd dword [const1]  ; ST0 = 1 + (25/(b*b))

 ; Calculate numerator part (36/(b*b))
 fld dword [const36]  ; ST0 = 36, ST1 = denominator, ST2 = b*b
 fdiv st0, st2          ; ST0 = 36/(b*b)

```

```
; Divide numerator by denominator
fdiv st0, st1          ; ST0 = (36/(b*b))/(denominator)

; Calculate a + b*b
fld dword [a]          ; ST0 = a, ST1 = intermediate result, ST2 = denominator, ST3 = b*b
fadd st0, st3          ; ST0 = a + b*b

; Final subtraction
fsub st0, st1          ; ST0 = (a + b*b) - intermediate result

; Store result
fstp dword [z]         ; Store result in z and pop

; Exit the program
mov eax, 1
mov ebx, 0
int 0x80
```

Mathematical Expression

The program implements the following expression:

$$z = a + b^2 - \frac{36/b^2}{1 + 25/b^2}$$

Data Section

```
section .data
 a dd 5.0        ; Example value for a (float)
 b dd 2.0        ; Example value for b (float)
 z dd 0.0        ; Result storage
 const36 dd 36.0
 const25 dd 25.0
 const1 dd 1.0
```

The data section defines:

- `a` - a floating-point value initialized to 5.0
- `b` - a floating-point value initialized to 2.0
- `z` - a floating-point variable to store the result, initialized to 0.0
- Constants needed for the calculation: 36.0, 25.0, and 1.0

Note that `dd` stands for "define double" and creates 32-bit (4-byte) floating-point values.

GDB Debugging Analysis

Setting a Breakpoint

```
(gdb) break _start
```

```
Breakpoint 1 at 0x8048080: file program.asm, line 15.
(gdb) run
Starting program: /home/dumas/Documents/Lab7/Task2/program
...
Breakpoint 1, _start () at program.asm:15
15          fld dword [b]        ; ST0 = b
```

The program execution begins at the `_start` label and stops at the first instruction. This is achieved by:

1. Setting a breakpoint at the `_start` label with the `break _start` command
2. Starting program execution with the `run` command
3. GDB automatically stops at the first instruction of the `_start` label, which loads the value of `b` onto the FPU stack

Step 1: Loading b into FPU Stack

```
(gdb) stepi
16          fmul st0, st0        ; ST0 = b*b
```

The instruction `fld dword [b]` loads the floating-point value of `b` (2.0) from memory into the FPU stack:

- `fld` stands for "floating-point load"
- `dword` specifies that we are loading a 32-bit (4-byte) floating-point value
- `[b]` indicates we're loading from the memory location labeled `b`

The FPU stack is organized as a stack of 8 registers labeled ST0 through ST7, with ST0 being the top of the stack. After executing this instruction, ST0 contains the value 2.0.

FPU Register State After Loading b

```
(gdb) info float
=>R7: Valid   0x4000800000000000000 +2
  R6: Empty   0x00000000000000000000
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000
Status Word:         0x3800
                       TOP: 7
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
```

The `info float` command displays the current state of the FPU registers:

- The value 2.0 has been loaded into register R7 (which is currently ST0)
- `TOP: 7` means R7 is currently the top of the stack (ST0)
- The hexadecimal representation `0x40008000000000000000` corresponds to the IEEE 754 floating-point format for 2.0
- All other registers (R0-R6) are empty
- The FPU Control Word (0x037f) shows:
  - All exception masks are enabled (IM, DM, ZM, OM, UM, PM)
  - The precision control (PC) is set to Extended Precision (64-bits)
  - The rounding control (RC) is set to "Round to nearest"

Step 2: Squaring b

```
(gdb) stepi
19          fld dword [const25]  ; ST0 = 25, ST1 = b*b
```

The instruction `fmul st0, st0` multiplies ST0 by itself:

- `fmul` is the floating-point multiply instruction
- `st0, st0` specifies that we're multiplying the top of the stack by itself
- This effectively computes $b^2 = 2.0^2 = 4.0$

After this operation, ST0 contains 4.0, which is the square of the original value.

FPU Register State After Squaring b

```
(gdb) info float
=>R7: Valid   0x40018000000000000000 +4
  R6: Empty   0x00000000000000000000
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000
Status Word:         0x3800
                       TOP: 7
```

We can see that:

- R7 (ST0) now contains the value 4.0, which is b^2
- The hexadecimal representation has changed to `0x40018000000000000000`, representing 4.0
- TOP is still 7, indicating R7 is still the top of the stack

Step 3: Loading constant 25

```
(gdb) stepi
20          fdiv st0, st1        ; ST0 = 25/(b*b)
```

The instruction `fld dword [const25]` loads the constant 25.0 from memory onto the top of the FPU stack:

- This pushes the previous value ($b^2 = 4.0$) down one position in the stack
- After this operation:
  - ST0 = 25.0 (newly loaded value)
  - ST1 = 4.0 (b^2 from previous step)

FPU Register State After Loading constant 25

```
(gdb) info float
  R7: Valid   0x40018000000000000000 +4
=>R6: Valid   0x4003c800000000000000 +25
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000
Status Word:         0x3000
                       TOP: 6
```

We can see that:

- R6 is now the top of the stack (ST0) and contains 25.0
- R7 has moved down to ST1 and still contains 4.0 (b^2)
- TOP = 6 indicates that R6 is now the top of the stack (ST0)
- The status word has changed to 0x3000, reflecting the new stack position

Step 4: Division for 25/(b*b)

```
(gdb) stepi
21          fadd dword [const1]  ; ST0 = 1 + (25/(b*b))
```

The instruction `fdiv st0, st1` divides ST0 (25.0) by ST1 (4.0):

- `fdiv` is the floating-point division instruction
- This computes $\frac{25.0}{4.0} = 6.25$
- After this operation:
  - ST0 = 6.25 (result of 25.0/4.0)
  - ST1 = 4.0 (unchanged)

General Register State

```
(gdb) info registers
eax            0x0                 0
ecx            0x0                 0
edx            0x0                 0
ebx            0x0                 0
esp            0xffffcaa0          0xffffcaa0
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0x8048090           0x8048090 <_start+16>
eflags         0x202               [ IF ]
```

The general-purpose registers are mostly unused at this point since the calculation is being performed using the FPU stack:

- All general registers (EAX, ECX, EDX, EBX, etc.) are still at their initial values of 0
- Only the EIP (instruction pointer) has advanced as it points to the current instruction
- The stack pointer (ESP) is at its initial position, as we haven't used the main stack yet
- This shows that FPU operations do not affect the general-purpose registers, which is one of the benefits of using the FPU for floating-point calculations

Complete Program Analysis

Calculation Strategy

The program calculates the expression $z = a + b^2 - \frac{36/b^2}{1+25/b^2}$ in the following steps:

1. Calculate b^2 and keep it on the stack
2. Calculate the denominator part: $1 + \frac{25}{b^2}$
3. Calculate the numerator part: $\frac{36}{b^2}$
4. Divide numerator by denominator: $\frac{36/b^2}{1+25/b^2}$
5. Calculate $a + b^2$
6. Perform the final subtraction: $(a + b^2) - \frac{36/b^2}{1+25/b^2}$
7. Store the result in variable z

Code Analysis

```
; Load values
fld dword [b]        ; ST0 = b
fmul st0, st0        ; ST0 = b*b
```

First, the value of `b` (2.0) is loaded onto the FPU stack and squared to get b^2 (4.0):

- `fld` loads the float value from memory
- `fmul st0, st0` multiplies the value by itself (squaring)

```
; Calculate denominator part (1 + (25/(b*b)))
fld dword [const25] ; ST0 = 25, ST1 = b*b
fdiv st0, st1       ; ST0 = 25/(b*b)
fadd dword [const1] ; ST0 = 1 + (25/(b*b))
```

The denominator part $1 + \frac{25}{b^2}$ is calculated:

- Load 25.0 onto the stack (pushing b^2 down to ST1)
- Divide 25.0 by b^2 to get $\frac{25}{b^2} = \frac{25}{4} = 6.25$
- Add 1.0 to get $1 + \frac{25}{b^2} = 1 + 6.25 = 7.25$

```
; Calculate numerator part (36/(b*b))
fld dword [const36] ; ST0 = 36, ST1 = denominator, ST2 = b*b
fdiv st0, st2       ; ST0 = 36/(b*b)
```

The numerator part $\frac{36}{b^2}$ is calculated:

- Load 36.0 onto the stack (pushing previous values down)
- Divide 36.0 by b^2 (which is now in ST2) to get $\frac{36}{b^2} = \frac{36}{4} = 9.0$

```
; Divide numerator by denominator
fdiv st0, st1       ; ST0 = (36/(b*b))/(denominator)
```

The division $\frac{36/b^2}{1+25/b^2} = \frac{9.0}{7.25} \approx 1.24$ is performed by dividing the numerator (in ST0) by the denominator (in ST1).

```
; Calculate a + b*b
fld dword [a]       ; ST0 = a, ST1 = intermediate result, ST2 = denominator,
    ST3 = b*b
fadd st0, st3       ; ST0 = a + b*b
```

The sum $a + b^2 = 5.0 + 4.0 = 9.0$ is calculated:

- Load `a` (5.0) onto the stack
- Add to it the value of b^2, which is now in ST3

```
; Final subtraction
fsub st0, st1       ; ST0 = (a + b*b) - intermediate result
```

The final subtraction $(a + b^2) - \frac{36/b^2}{1+25/b^2} = 9.0 - 1.24 = 7.76$ is performed by subtracting the intermediate result (in ST1) from the sum (in ST0).

```
; Store result
fstp dword [z]      ; Store result in z and pop
```

The final result is stored in the memory variable `z` and removed from the FPU stack:

- `fstp` (floating-point store and pop) saves the value to memory and removes it from the stack
- This cleans up the FPU stack as part of good programming practice

Program Exit

```
; Exit
mov eax, 1
mov ebx, 0
int 0x80
```

The program terminates with a system call to exit:

- System call number 1 (exit) is loaded into EAX
- Return code 0 (indicating successful execution) is loaded into EBX
- Interrupt 0x80 is triggered to invoke the Linux kernel's system call handler

1.2 Analysis of Assembly Code and Debugging Output

1.2.1 Source Code with Comments

Below is the analyzed assembly program with detailed comments:

```
section .data
    mes1 db "Enter the X:", 0         ; Prompt message for X
    mes2 db "Enter the Y:", 0         ; Prompt message for Y
    mes3 db "Result: ", 0             ; Message to print before result
    newline db 10, 0                  ; Newline character
    vrx dd 0                          ; Variable to store X
    vry dd 0                          ; Variable to store Y
    rez dd 0                          ; Variable to store result

section .bss
    input_buffer resb 16              ; Buffer for keyboard input

section .text
    global _start
_start:
    ; Prompt for X
    mov eax, 4           ; sys_write
    mov ebx, 1           ; stdout
    mov ecx, mes1        ; message to display
    mov edx, 12          ; length of message
    int 0x80

    ; Read X from stdin
    mov eax, 3           ; sys_read
    mov ebx, 0           ; stdin
```

```
mov ecx, input_buffer
mov edx, 16
int 0x80

; Convert input to integer and store in vrx
mov esi, input_buffer
call atoi
mov [vrx], eax

; Prompt for Y
mov eax, 4
mov ebx, 1
mov ecx, mes2
mov edx, 12
int 0x80

; Read Y
mov eax, 3
mov ebx, 0
mov ecx, input_buffer
mov edx, 16
int 0x80

; Convert Y and store in vry
mov esi, input_buffer
call atoi
mov [vry], eax

; Calculate Z = (X/8 + 32 - Y) if X < 2Y, else Z = 2Y - 60
xor eax, eax
xor edx, edx
mov eax, [vry]
mov ebx, 2
mul ebx                 ; EAX = 2Y
cmp [vrx], eax
jb con1                 ; if X < 2Y, jump to con1

; Case: X >= 2Y → Z = 2Y - 60
mov eax, [vry]
mov ebx, 2
mul ebx
sub eax, 60
```

```
    mov [rez], eax
    jmp ex

con1:
    ; Case: X < 2Y → Z = X / 8 + 32 - Y
    mov eax, [vrx]
    mov ebx, 8
    xor edx, edx
    div ebx
    add eax, 32
    sub eax, [vry]
    mov [rez], eax

ex:
    ; Display result
    mov eax, 4
    mov ebx, 1
    mov ecx, mes3
    mov edx, 8
    int 0x80

    ; Convert result to ASCII and print it
    mov eax, [rez]
    call itoa

    ; Newline
    mov eax, 4
    mov ebx, 1
    mov ecx, newline
    mov edx, 1
    int 0x80

    ; Exit
    mov eax, 1
    xor ebx, ebx
    int 0x80

; ASCII to integer
atoi:
    xor eax, eax
    xor ecx, ecx
    mov ebx, 10
```

```
.next_digit:
    mov cl, [esi]
    cmp cl, '0'
    jb .done
    cmp cl, '9'
    ja .done
    sub cl, '0'
    mul ebx
    add eax, ecx
    inc esi
    jmp .next_digit
.done:
    ret

; Integer to ASCII
itoa:
    mov ebx, 10
    xor ecx, ecx
    mov edi, input_buffer + 15
    mov byte [edi], 0
    dec edi
.convert_loop:
    xor edx, edx
    div ebx
    add dl, '0'
    mov [edi], dl
    dec edi
    inc ecx
    test eax, eax
    jnz .convert_loop
    inc edi
    mov eax, 4
    mov ebx, 1
    mov edx, ecx
    mov ecx, edi
    int 0x80
    ret
```

1.2.2 Debugging Analysis Using `gdb`

The program was debugged using `gdb` with the following steps:

1. **Set Breakpoints:**

```
(gdb) break _start
(gdb) break atoi
(gdb) break itoa
```

2. **Start Program Execution:**

```
(gdb) run
Breakpoint 1, _start () at program.asm:18
```

3. **Step Through Initial Instructions:**

```
(gdb) ni            ; Move to next instruction
(gdb) si            ; Step into instruction
(gdb) stepi         ; Execute instruction by instruction
```

4. **Check Memory Contents:**

```
(gdb) x/wx &vrx     ; Check value of X (initially 0)
(gdb) x/wx &vry     ; Check value of Y (initially 0)
```

5. **Enter Input When Prompted:**

```
Enter the X:2
Enter the Y:4
```

6. **Break in atoi Function (X Conversion):**

```
Breakpoint 2, atoi () at program.asm:110
(gdb) print $eax     ; Will be 2 after conversion
```

7. **Break in atoi Function (Y Conversion):**

```
Breakpoint 2, atoi () at program.asm:110
(gdb) print $eax     ; Will be 4 after conversion
```

8. **Condition Check X < 2Y:**

```
X = 2, Y = 4 → 2 < 8 → Jump to con1 executed
Z = (2 / 8) + 32 - 4 = 0 + 32 - 4 = 28
```

9. **Break in itoa:**

```
Breakpoint 3, itoa () at program.asm:131
(gdb) print $eax      ; Shows 28 as final result
```

10. **Final Output:**

```
Result: 28
```

1.2.3 Conclusion

Through debugging, we validated the program's logic:

- The program correctly prompts the user and reads values for `X` and `Y`.
- It performs a conditional check: if `X < 2Y`, it uses the first formula; otherwise, it uses the second.
- The result is calculated, converted to ASCII, and displayed.

For inputs `X = 2` and `Y = 4`, the output was:

```
Result: 28
```

This matches the expected value of $Z = \frac{2}{8} + 32 - 4 = 28$, confirming correct implementation and logic.

1.3 Assembly Program Analysis and Debugging Part 2

1.3.1 Task Overview

This lab assignment involves implementing an Assembly program in two versions:

- **Version 1**: Accepts input from the keyboard for values of X and Y.
- **Version 2**: Generates values of X and Y randomly using `rand`, `RandomRange`, and `time`.

The goal is to compute the value of Z based on the condition:

$$Z = \begin{cases} \frac{Y-X}{2} + 102, & \text{if } X < 2Y \\ 4X - Y, & \text{if } X \geq 2Y \end{cases}$$

1.3.2 Full Source Code with Comments

```
section .data
    prompt_x     db "Enter value for X: ", 0
    prompt_y     db "Enter value for Y: ", 0
    result_msg   db "Result Z = ", 0
```

```
    formula1_msg db "Using formula (Y - X)/2 + 102 (X < 2Y)", 10, 0
    formula2_msg db "Using formula 4X - Y (X >= 2Y)", 10, 0
    x_value_msg  db "X = ", 0
    y_value_msg  db "Y = ", 0
    newline      db 10, 0
    random_msg   db "Random mode values:", 10, 0
    mode_prompt  db "Choose mode (1-keyboard input, 2-random values): ", 0
    format_in    db "%d", 0
    format_out   db "%d", 0

section .bss
    x       resd 1    ; Variable for X
    y       resd 1    ; Variable for Y
    z       resd 1    ; Variable for Z
    mode    resd 1    ; Mode selection (1 or 2)

section .text
    global main
    extern printf, scanf, srand, rand, time

main:
    push rbp
    mov rbp, rsp

    ; Prompt user for input mode
    mov rdi, mode_prompt
    xor rax, rax
    call printf

    ; Read mode selection (1 or 2)
    mov rdi, format_in
    mov rsi, mode
    xor rax, rax
    call scanf

    ; Compare input and jump to corresponding mode
    mov eax, [mode]
    cmp eax, 1
    je user_input_mode
    jmp random_mode

user_input_mode:
```

```
    ; Request X
    mov rdi, prompt_x
    xor rax, rax
    call printf

    ; Read X
    mov rdi, format_in
    mov rsi, x
    xor rax, rax
    call scanf

    ; Request Y
    mov rdi, prompt_y
    xor rax, rax
    call printf

    ; Read Y
    mov rdi, format_in
    mov rsi, y
    xor rax, rax
    call scanf
    jmp calculate_z

random_mode:
    ; Initialize RNG
    xor rdi, rdi
    call time
    mov rdi, rax
    call srand

    ; Generate X in range [-100, 100]
    call rand
    mov edx, 0
    mov ecx, 201
    div ecx
    sub edx, 100
    mov [x], edx

    ; Generate Y in range [-100, 100]
    call rand
    mov edx, 0
    mov ecx, 201
```

```
    div ecx
    sub edx, 100
    mov [y], edx

    ; Display values
    mov rdi, random_msg
    xor rax, rax
    call printf

    mov rdi, x_value_msg
    xor rax, rax
    call printf
    mov rdi, format_out
    mov esi, [x]
    xor rax, rax
    call printf

    mov rdi, newline
    xor rax, rax
    call printf

    mov rdi, y_value_msg
    xor rax, rax
    call printf
    mov rdi, format_out
    mov esi, [y]
    xor rax, rax
    call printf

    mov rdi, newline
    xor rax, rax
    call printf

calculate_z:
    ; Compare: if X < 2Y
    mov eax, [y]
    add eax, eax      ; 2Y
    cmp [x], eax
    jge second_formula

    ; Formula 1: Z = (Y - X)/2 + 102
    mov rdi, formula1_msg
```

```
    xor rax, rax
    call printf
    mov eax, [y]
    sub eax, [x]
    cdq
    mov ecx, 2
    idiv ecx
    add eax, 102
    mov [z], eax
    jmp print_result

second_formula:
    ; Formula 2: Z = 4X - Y
    mov rdi, formula2_msg
    xor rax, rax
    call printf
    mov eax, [x]
    imul eax, 4
    sub eax, [y]
    mov [z], eax

print_result:
    ; Print Z
    mov rdi, result_msg
    xor rax, rax
    call printf

    mov rdi, format_out
    mov esi, [z]
    xor rax, rax
    call printf

    mov rdi, newline
    xor rax, rax
    call printf

    xor eax, eax
    leave
    ret
```

1.3.3 Debugging Analysis

Using `gdb`, the following steps and checks are expected during debugging:

1. **Breakpoint at `main`**: Allows step-by-step tracing from program start.
2. **User Input Mode (mode = 1)**:
   - Step through prompts for `X` and `Y`.
   - Use `print/x $eax`, `print/x $esi` to view input values.
   - Step into `calculate_z` and observe branching based on comparison `X < 2Y`.
3. **Random Mode (mode = 2)**:
   - Observe call to `time` and `srand`.
   - Track outputs from `rand` and how values are scaled to [-100, 100].
   - Use `x/wx x`, `x/wx y` to check memory values.
4. **Z Calculation**:
   - Set breakpoints before and after both formulas.
   - Track EAX before storing to [z].
   - Final output value is printed using `printf`, and you can observe it in the output window.

1.3.4 Example Output

```
Choose mode (1-keyboard input, 2-random values): 1
Enter value for X: 10
Enter value for Y: 20
Using formula (Y - X)/2 + 102 (X < 2Y)
Result Z = 107

Choose mode (1-keyboard input, 2-random values): 2
Random mode values:
X = -38
Y = 91
Using formula (Y - X)/2 + 102 (X < 2Y)
Result Z = 156
```

1.4 Assembly Program: Swapping Pairs in an Array

1.4.1 Task

Write a program with a loop and indexed addressing that exchanges every pair of values in an array with an even number of elements. Item i should exchange with item $i+1$, item $i+2$ with $i+3$, and so on.

1.4.2 Source Code with Comments

```
section .data
    ; Sample array with 10 elements (even count)
    array       dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    array_len   equ ($ - array) / 4  ; Number of elements in the array

    ; Messages for output
    msg_before  db "Array before swapping: ", 0
    msg_after   db "Array after swapping:  ", 0
    msg_space   db " ", 0
    msg_newline db 10, 0

    ; Printf format
    format      db "%d", 0

section .bss
    ; No dynamic variables needed

section .text
    global main
    extern printf

main:
    push rbp
    mov rbp, rsp

    ; Display the original array
    mov rdi, msg_before
    xor rax, rax
    call printf

    mov r12, 0                   ; Index variable
    call print_array             ; Print before swapping

    ; Start swapping loop
    xor r12, r12                 ; r12 = index (start at 0)

swap_loop:
    cmp r12, array_len           ; Check if end of array is reached
    jge swap_done                ; If done, exit loop
```

```
    ; Calculate byte offset of index
    mov rax, r12
    shl rax, 2                  ; Multiply index by 4 (32-bit elements)

    ; Load elements to swap
    mov ecx, [array + rax]      ; Load array[i]
    mov edx, [array + rax + 4]  ; Load array[i+1]

    ; Perform swap
    mov [array + rax], edx      ; Store array[i+1] at array[i]
    mov [array + rax + 4], ecx  ; Store array[i] at array[i+1]

    ; Move to next pair (increment by 2)
    add r12, 2
    jmp swap_loop

swap_done:
    ; Print newline and swapped array
    mov rdi, msg_newline
    xor rax, rax
    call printf

    mov rdi, msg_after
    xor rax, rax
    call printf

    mov r12, 0
    call print_array

    ; Exit program
    xor eax, eax
    leave
    ret

; Procedure: print_array
; Prints the contents of the array
print_array:
print_loop:
    cmp r12, array_len
    jge print_done

    ; Calculate offset
```

```
    mov rax, r12
    shl rax, 2

    ; Print current element
    mov rdi, format
    mov esi, [array + rax]
    xor rax, rax
    call printf

    ; Print space
    mov rdi, msg_space
    xor rax, rax
    call printf

    ; Next index
    inc r12
    jmp print_loop

print_done:
    mov rdi, msg_newline
    xor rax, rax
    call printf
    ret
```

1.4.3 Explanation

- The array is declared statically with 10 elements.
- The loop iterates over the array two elements at a time.
- Indexed addressing is used to access and swap values using `[array + rax]` and `[array + rax + 4]`.
- The value at index i is stored in `ecx`, and $i + 1$ in `edx`, then swapped in memory.

1.4.4 Sample Output

```
Array before swapping: 1 2 3 4 5 6 7 8 9 10
Array after swapping:  2 1 4 3 6 5 8 7 10 9
```

1.4.5 Debugging Notes

- You can set a breakpoint at `swap_loop` to observe each iteration of the pair swap.
- Use `print r12`, `x/2dw array + rax` in GDB to monitor progress.
- The loop condition `cmp r12, array_len` ensures we don't exceed bounds.

1.5 Assembly Program: Copying a 16-bit Array into a 32-bit Array

1.5.1 Task

Write a program that uses a loop to copy all the elements from an unsigned Word (16-bit) array into an unsigned doubleword (32-bit) array.

1.5.2 Source Code with Comments

```
section .data
    ; Source array with 16-bit elements
    src_array    dw 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000
    src_len      equ ($ - src_array) / 2   ; Total number of elements

    ; Output strings
    msg_src      db "Source array (16-bit): ", 0
    msg_dest     db "Destination array (32-bit): ", 0
    msg_space    db " ", 0
    msg_newline  db 10, 0

    ; Printf format strings
    format_word  db "%hu", 0       ; Unsigned 16-bit
    format_dword db "%u", 0        ; Unsigned 32-bit

section .bss
    ; Destination array with 32-bit elements
    dest_array   resd src_len      ; Allocate same number of 32-bit elements

section .text
    global main
    extern printf

main:
    push rbp
    mov rbp, rsp

    ; Print original 16-bit source array
    mov rdi, msg_src
    xor rax, rax
    call printf
```

```
    mov r12, 0
    call print_src_array

    ; Loop to copy from 16-bit array to 32-bit array
    xor rsi, rsi                   ; Index = 0

copy_loop:
    cmp rsi, src_len
    jge copy_done

    ; Zero-extend and copy from 16-bit to 32-bit
    movzx eax, word [src_array + rsi*2]
    mov [dest_array + rsi*4], eax

    inc rsi
    jmp copy_loop

copy_done:
    ; Print newline and destination array
    mov rdi, msg_newline
    xor rax, rax
    call printf

    mov rdi, msg_dest
    xor rax, rax
    call printf

    mov r12, 0
    call print_dest_array

    ; Exit program
    xor eax, eax
    leave
    ret

; Print the 16-bit source array
print_src_array:
print_src_loop:
    cmp r12, src_len
    jge print_src_done

    mov rdi, format_word
```

```
    movzx esi, word [src_array + r12*2]
    xor rax, rax
    call printf

    mov rdi, msg_space
    xor rax, rax
    call printf

    inc r12
    jmp print_src_loop

print_src_done:
    mov rdi, msg_newline
    xor rax, rax
    call printf
    ret

; Print the 32-bit destination array
print_dest_array:
print_dest_loop:
    cmp r12, src_len
    jge print_dest_done

    mov rdi, format_dword
    mov esi, [dest_array + r12*4]
    xor rax, rax
    call printf

    mov rdi, msg_space
    xor rax, rax
    call printf

    inc r12
    jmp print_dest_loop

print_dest_done:
    mov rdi, msg_newline
    xor rax, rax
    call printf
    ret
```

1.5.3 Explanation

- The source array consists of 16-bit unsigned integers (declared with `dw`).
- The destination array is allocated in the `.bss` section with 32-bit entries (`resd`).
- In the loop, each 16-bit value is loaded using `movzx` to zero-extend to 32-bit.
- The value is then written to the destination array.
- Two procedures, `print_src_array` and `print_dest_array`, display the contents of the arrays.

1.5.4 Sample Output

```
Source array (16-bit): 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000
Destination array (32-bit): 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000
```

1.5.5 Debugging Tips

- Use `print rsi`, `x/hw src_array + rsi*2` and `x/wd dest_array + rsi*4` in GDB to verify values during the loop.
- Ensure `movzx` is used for zero-extension; otherwise upper bits may contain garbage.
- The element size scaling factors (2 and 4) are important for correct memory access.

1.6 Procedure random string

Create an x86-64 assembly program that:

1. Implements a procedure `generate_random_string` to create a random string of capital letters (A–Z) of given length.
2. Generates and displays 20 such random strings on the console.

Program Overview

This program uses the following components:

- `generate_random_string` uses `rdtsc` to produce pseudo-random values and convert them into capital letters.
- `int_to_string` converts a number (1–20) into a printable ASCII string for labeling.
- The strings are printed to the console using direct Linux system calls (`sys_write`).

Source Code

```
section .data
    newline db 10
    str_len equ 20
    num_strings equ 20
    msg_prefix db "String ", 0
    msg_prefix_len equ $ - msg_prefix

section .bss
    random_string resb 256
    display_buffer resb 8

section .text
    global _start

generate_random_string:
    push rcx
    push rdx
    push rax
    mov rcx, rax

.loop:
    rdtsc
    xor edx, eax
    mov al, dl
    and al, 0x1F
    cmp al, 26
    jl .valid_letter
    sub al, 6

.valid_letter:
    add al, 'A'
    mov [rdi], al
    inc rdi
    loop .loop

    mov byte [rdi], 0
    pop rax
    pop rdx
    pop rcx
    ret

int_to_string:
    push rbx
    push rcx
```

```
    push rdx
    mov rbx, 10
    mov rcx, 0

.push_digits:
    xor rdx, rdx
    div rbx
    add dl, '0'
    push rdx
    inc rcx
    test rax, rax
    jnz .push_digits

.pop_digits:
    pop rdx
    mov [rdi], dl
    inc rdi
    loop .pop_digits

    pop rdx
    pop rcx
    pop rbx
    ret

print_string:
    push rax
    push rdi
    mov rax, 1
    mov rdi, 1
    syscall
    pop rdi
    pop rax
    ret

_start:
    rdtsc
    mov rbx, 1

.generate_strings:
    mov rsi, msg_prefix
    mov rdx, msg_prefix_len
    call print_string

    mov rax, rbx
    mov rdi, display_buffer
    call int_to_string

```

```
    mov rsi, display_buffer
    mov rdx, rdi
    sub rdx, display_buffer
    call print_string

    mov rsi, newline
    mov rdx, 1
    call print_string

    mov eax, str_len
    mov rdi, random_string
    call generate_random_string

    mov rsi, random_string
    mov rdx, str_len
    call print_string

    mov rsi, newline
    mov rdx, 1
    call print_string

    inc rbx
    cmp rbx, num_strings + 1
    jle .generate_strings

    mov rax, 60
    xor rdi, rdi
    syscall
```

Sample Output

```
String 1
YIUYCMQUGYKWIUEIUYIU
String 2
CMYIMYKWOASWCGQCOASE
String 3
UYQUEIUEWIUYCUYKWASE
...
String 20
IUUMYYQWOUMYEWWUUMYE
```

Debugging with GDB

Compiling with Debug Info

```
nasm -f elf64 -g -F dwarf random_string.asm -o random_string.o
ld random_string.o -o random_string
```

Basic GDB Usage

- Launch GDB:

  ```
  gdb ./random_string
  ```

- Set breakpoints:

  ```
  b _start
  b generate_random_string
  ```

- Run and step through:

  ```
  run
  si  ; step instruction
  x/s random_string   ; view generated string
  ```

Conclusion

This program demonstrates:

- Use of `rdtsc` for pseudo-random generation.
- Assembly-level procedures with system calls for I/O.
- Manual ASCII string manipulation and printing.
- Number-to-string conversion without libraries.

If extended, this could be enhanced with:

- Better randomness (e.g., from `/dev/urandom`)
- Dynamic memory or varying string lengths
- Integration with C or a higher-level interface

Explanation of Assembly Commands

ADD Adds two operands.
ADC Add with carry; adds two operands and the carry flag.
SUB Subtracts the second operand from the first.
SBB Subtract with borrow; subtracts the second operand and the carry flag from the first.
CBW Convert byte to word; sign-extends AL into AX.
CWD Convert word to double word; sign-extends AX into DX:AX.
CDQ Convert double word to quad word; sign-extends EAX into EDX:EAX.
MUL Unsigned multiplication.
IMUL Signed multiplication.
DIV Unsigned division.
IDIV Signed division.
MOV Moves data from source to destination.
MOVZX Moves with zero-extension.
MOVSX Moves with sign-extension.
XCHG Exchanges two operands.
XLAT Table lookup translation using AL and BX.
IN Reads data from a port.
OUT Sends data to a port.
LEA Load effective address into register.
LAHF Load lower byte of EFLAGS into AH.
SAHF Store AH into lower byte of EFLAGS.
PUSH Pushes operand onto stack.
POP Pops top of stack into operand.
PUSHFD Pushes EFLAGS register onto stack.
POPFD Pops the top of stack into EFLAGS.
PUSHAD Pushes all general-purpose registers onto the stack.
PUSHA Pushes general-purpose registers (16-bit).
POPAD Pops general-purpose registers (32-bit) from the stack.
POPA Pops general-purpose registers (16-bit).
INC Increments operand by one.
DEC Decrements operand by one.
NEG Negates operand (two's complement).
CMP Compares two operands (subtracts but doesn't store result).

JMP Unconditional jump.
JE, JZ Jump if equal / zero flag set.
JNE, JNZ Jump if not equal / zero flag not set.
JL, JNGE Jump if less (signed).
JLE, JNG Jump if less or equal (signed).
JG, JNLE Jump if greater (signed).
JGE, JNL Jump if greater or equal (signed).
JB, JNAE Jump if below (unsigned).
JBE, JNA Jump if below or equal (unsigned).
JA, JNBE Jump if above (unsigned).
JAE, JNB Jump if above or equal (unsigned).
JCXZ Jump if CX register is zero.
AAA ASCII adjust after addition.
AAS ASCII adjust after subtraction.
DAS Decimal adjust after subtraction.
AAM ASCII adjust after multiply.

2
Conclusion

2.1 Conclusion

This lab provided extensive hands-on experience with x86 assembly programming, covering a range of topics including data manipulation, memory addressing, and system calls. Through various exercises, we explored both fundamental concepts and advanced techniques, such as:

1. **Basic Assembly Operations**: We learned to manipulate registers, perform arithmetic operations, and manage memory effectively using different addressing modes.
2. **Debugging with GDB**: We utilized GDB to debug assembly programs, stepping through instructions, examining register states, and inspecting memory contents. This enhanced our understanding of program flow and the impact of individual instructions.
3. **Complex Data Handling**: The lab involved creating complex data structures, such as arrays, and implementing algorithms to manipulate them. Tasks included swapping elements, copying data between different data types, and generating random strings.
4. **System Interaction**: We practiced using system calls for input/output operations, enabling us to interact with the console and handle user input dynamically.
5. **Mathematical Computations**: The lab included floating-point calculations using the Floating-Point Unit (FPU), allowing us to perform complex mathematical operations with precision.
6. **Procedural Programming**: We implemented procedures for generating random strings and converting integers to strings, demonstrating the modularity and reusability of code in assembly language.

Overall, this lab reinforced the importance of low-level programming concepts and provided a solid foundation for understanding how software interacts with hardware. The skills acquired are essential for further studies in systems programming and embedded systems development.