X86 64 Part 2

GDB & Instruction Format

Kumaravel Jagasivamani
Apr. 1-3, 2024

Contents

- Run-time and Syntax errors
- gdb commands
- gdb Tutorial
- Moving immediate data to memory
- Memory Endianness
- Moving data directly between memory
- Instruction Disassembly Details
- Instruction Operand Encoding
- Special Registers
- Pipelining/Dependency/Hazards Introduction
- Function calls

Run-time and Syntax errors

- Run-time errors:
  - Use debugger
- Syntax errors:
  - mov [rsi], [rdi]
  - Error:

```
cmsc313_proj2.asm:117: error: invalid combination of opcode and operands
```

  - Refer to instruction manual:
    - **https://cdrdv2.intel.com/v1/dl/getContent/671110**

Debugger

- Troubleshoot issue in code
- Have visibility to values of all registers
- Can step through code one instruction at a time

Problem code

```
    ; Clear [loop_index] with 0
    xor rax, rax
    mov [loop_index], rax
loop2:
    mov rdi, input_buffer
    mov rax, [loop_index]
    sal rax, 3
    add rdi, rax
    mov rbx, [rdi]
    mov rsi, o
    mov [rsi], rbx

    inc qword [loop_index]
    mov rax, [samples_cnt]
    cmp qword rax, [loop_index]
    jne loop2
```

Problem code

```
    ; Clear [loop_index] with 0
    xor rax, rax
    mov [loop_index], rax
loop2:
    mov rdi, input_buffer
    mov rax, [loop_index]
    sal rax, 3                  -> Need to shift by 5: each sample has 4*8bytes
    add rdi, rax                -> For o output, need to add offset of 24 bytes within sample
    mov rbx, [rdi]
    mov rsi, o
    mov [rsi], rbx              -> Need to add offset for each output sample

    inc qword [loop_index]
    mov rax, [samples_cnt]
    cmp qword rax, [loop_index]
    jne loop2
```

Running gdb

- gcc -c cmsc313_proj2.c -o cmsc313_proj2.c.o
- nasm -f elf64 -l cmsc313_proj2.lst cmsc313_proj2.asm -o cmsc313_proj2.o
- gcc cmsc313_proj2.c.o cmsc313_proj2.o **-g** -o cmsc313_proj2
- gdb cmsc313_proj2
  - Load the executable file as input
  - ‘Enable debuginfod for this session?’: **Enter ‘y’ to allow symbol access**

gdb Disassembly commands

- set disassembly-flavor intel
  - Use Intel x86 architecture
- disassemble main
  - List of low-level instructions
  - Shows RIP, instruction
    - RIP: Instruction Pointer Register (Program Counter)

Disassembly

- Assembly code:

```
main:
    mov rdi, input_buffer
    call load_input_buffer
    mov [samples_cnt], rax
```

- Disassembly output:

```
            RIP    RIP_Offset     Instruction
   0x0000000000401270 <+0>:movabs rdi,0x404034
   0x000000000040127a <+10>:call   0x401126 <load_input_buffer>
   0x000000000040127f <+15>:mov    QWORD PTR ds:0x404024,rax
```

gdb Breakpoint Commands

- break main
- break <label>
- break *0x<RIP>
  - Need the asterisk before the address
- info breakpoints: list of all breakpoints
- delete <index>: delete a breakpoint
- enable/disable <index>: enable/disable a breakpoint

gdb Flow Commands

- run
  - Run until the program hits a breakpoint
    - Start execution
- continue
  - Run until the program hits a breakpoint
    - Resume execution
- nexti
  - Run single instruction skipping function
- stepi
  - Run single instruction

Display registers and memory

- Registers:
  - info registers
  - print/x $rax
- Memory:
  - print (uint64_t)<label>
  - Printing N values in decimal format (option 1):
    - Find the address of the label: print &<label>
    - Use examine memory command: x/<N>**d**g 0x<addr>
  - Printing N values in hex format (option 2):
    - x/<N>**x**g (uint64_t*)&<label>

Lecture Action Items

gdb

- How to create breakpoint for a specific RIP value:
  - break *0x<addr>
  - Example: break *0x40127a

0x000000000040127a <+10>: call 0x401126 <load_input_buffer>

- How to resume code execution until the next breakpoint:
  - continue
- How to look at data memory location based on label instead of memory address
  - Use -g option for gcc
  - print (uint64_t)<label>
  - Printing N values in memory array:
    - print &<label>
    - x/<N>dg 0x<addr>
  - Printing N values:
    - x/<N>dg (uint64_t*)&<label>
- How to look at program memory:
  - x/8x1 $rip: display 8 bytes of program memory that will be executed next
  - x/8x1 0x40127a: display 8 bytes of program memory at provided memory address

Accessing memory contents

- print/x (uint64_t*)input_buffer
  - Display the value in memory pointed by input_buffer
- print/x (uint64_t)&input_buffer
  - Display input_buffer
- print/x $rdi
  - Display rdi register value
- print/x *$rdi
  - Display the value in memory pointed by rdi

Tutorial Commands

- set disassembly-flavor intel: instruction display format
- break main: set a breakpoint on main label
- run: Run till the main label
- break loop2: set a breakpoint on loop2 label
- continue: Continue execution until the next breakpoint
- disassemble loop2: Look at the assembly code with details on instructions and corresponding RIP/PC
- break *0x4012af: set a breakpoint at instruction with address 0x4012af
- print (uint64_t)loop_index: display the value at memory location with label of loop_index
- x/12dg (uint64_t*)&input_buffer: display 12 “giant” (8 byte) values in decimal format starting at memory location with input_buffer label
- print/x $rdi: print the value of rdi register
- Print *0x40404c: print the value at the memory location with address 0x40404c

Moving immediate data to memory

- For instruction list, find the format to specify the immediate data size
  - mov **qword** [rsi], 8
- Why size of immediate value needed for transfer to memory, but not for register?
  - Register name determines the size:
    - rax: 8 bytes, eax: 4 bytes, ax: 2 bytes, ah/al: 1 byte
  - Memory location doesn't convey the information
    - Same address can be used as the starting address to write to byte, 2 bytes, 4 bytes, 8 bytes
    - Example: If rsi is 0x100, the immediate value can overwrite bytes 0x100 (for 1 byte), 0x100~0x101 (2 bytes), 0x100~0x103 (4 bytes), 0x100~0x107 (8 bytes)

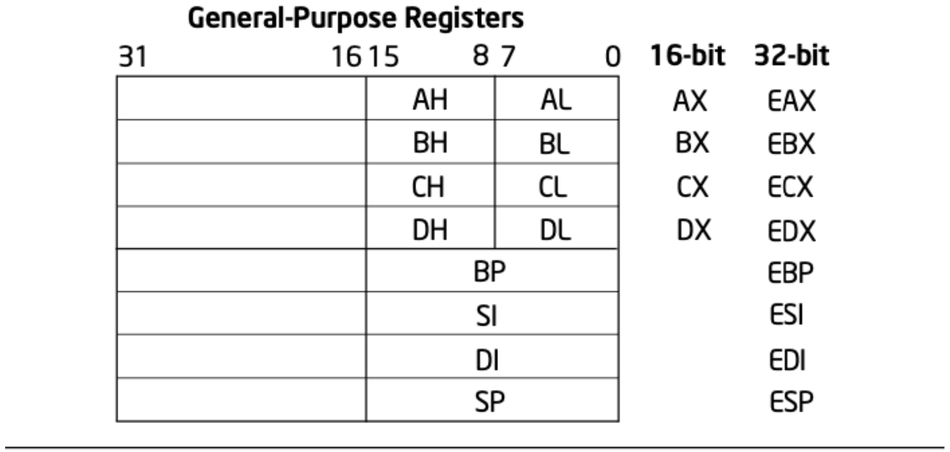


Figure 3-5. Alternate General-Purpose Register Names

Memory Endianness

- Data in little-endian format:
  - Lowest byte stored in lowest memory address
- Example:
  - Perform 8-byte write of 38483938 decimal to address input_buffer
    - mov rsi, input_buffer
    - mov qword [rsi], 0x24b37e2
      - In hexadecimal, 38483938 is 0x24b37e2

Address	Data
rsi + 0	0xe2
rsi + 1	0x37
rsi + 2	0x4b
rsi + 3	0x2
rsi + 4 .. rsi + 7	0

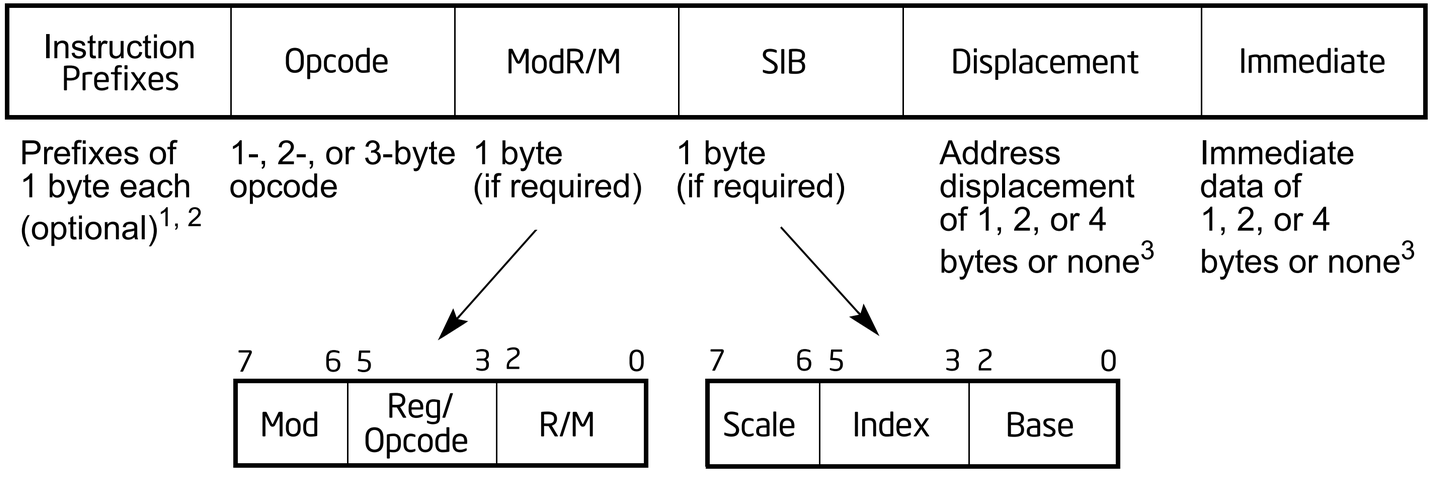
Moving data directly between memory

- movsq instruction
  - Use rsi as source memory address
  - Use rdi as destination memory address
  - Transfer quad-word (8 bytes)
- Smaller transfer: movsb (1 byte), movsw (2 bytes), movsd (4 bytes)

Instruction Format

Section 2.1

- Variable size of instruction:
  - 1~15 bytes
-



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, “REX Prefixes” for additional information.
2. For VEX encoding information, see Section 2.3, “Intel® Advanced Vector Extensions (Intel® AVX)”.
3. Some rare instructions can take an 8B immediate or 8B displacement.

Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Instruction Disassembly Details

- Assembly:
  - mov qword [rsi], 0x12345678
    - 32-bit immediate value sign-extended to 64-bits
- Disassembly:
  - 0000005B 48C70678563412 mov qword [rsi], 0x12345678
- Instruction Prefix (REX): 0x48
- Opcode: 0xC7
- ModR/M: 0x06
- Immediate: 0x78563412

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + C7 /0 id	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

Instruction Operand Encoding

- ModR/M: 0x06
- Operand 1: Destination
- Operation 2: Source
- Operand choices:
  - ModRM:r/m (w): register/memory (write)
  - ModRM:reg (r): register read
  - ModRM:reg (w): register write
  - ModRM: r/m (r): register/memory (read)
  - AL/AX/EAX/RAX: access 1/2/4/8 bytes of specified register
  - Moffs / Moffs (w): Specify address either as immediate value or offset to address register
    - mov rax, [0x404034]: read from hard-coded address (not recommended accessing a direct memory address)
    - mov [rdi + 0x8], rax: write to address location 8 bytes offset from value in rdi register
  - opcode + rd(w): register selection is encoded as part of opcode field instead of ModR/M field
  - imm8/16/32/64: immediate value provided as part of instruction
- General details in Section 3.1 of Volume 2

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Opera
MR	ModRM:r/m (w)	ModRM:reg (r)	N/
RM	ModRM:reg (w)	ModRM:r/m (r)	N/
FD	AL/AX/EAX/RAX	Moffs	N/
TD	Moffs (w)	AL/AX/EAX/RAX	N/
OI	opcode + rd (w)	imm8/16/32/64	N/
MI	ModRM:r/m (w)	imm8/16/32/64	N/

Special Registers

- “Normal” registers: RAX, RBX, RCX, RDX, R8~R15
- RDI: Destination Index Register
  - Used as destination memory address for instructions such as movsq
- RSI: Source Index Register
  - Used as source memory address for instructions such as movsq
- RSP: Stack Pointer
  - Stack addressing decrements
  - Points to location in stack that can be used for passing arguments to function and storing return address
- RBP: Base Pointer
  - Base of stack section used by current function for storing temporary values
  - Save RBP onto stack when entering function