x64 assembly programming using C library

Arvind S Raj
(arvindsraj@am.amrita.edu)

CSE467 IntroSSOC

B.Tech CSE Jan-May 2017

What is assembly code?

- Machine specific code generated by compiler from higher level code(eg: C programs).
- Sequence of instructions understood by the CPU.
- Lowest possible level of code $\Longrightarrow$ very verbose.

Why learn assembly programming?

- C can't be used everywhere. Eg: Atomic mutex locking, initial stages of bootloader.
- Assembly code can be faster. Eg: graphics and game programming, time critical and real time systems.
- Compilers avoid inter-procedure optimization and ignore efficient instructions.

Why learn assembly programming?(cont.)

- Writing code to spawn a shell after compromising an application.
- Understand how programs work at low level - useful for reverse engineering legitimate apps and malware, debugging without source code and much more.

Does any software feature assembly code?

- Most operating systems - Linux, Unix, BSD, Haiku.
- GNU GRUB, TrueCrypt, libav(formerly ffmpeg), GNU GMP.
- KolibriOS, MS-DOS and variants, L4: OS written in x86 assembly only.

Outline

View of a computer: C version

- Memory: Collection of variable of various types: char, int, float, structures etc.
- Lots of commands to perform several kinds of actions: modify variables, call functions, modify multiple variables and use result directly etc.
- Several working details abstracted out(but still quite verbose compared to higher level languages).
- Abstracts out library functions as known to exist somewhere and directly usable in programmer code.

View of a computer: assembly version

- Memory: A very large collection of bytes. No datatypes associated with a collection of bytes.
- Instruction Set Architecture: Finite set of instructions understood by processor.
- Many details of program execution and processor state visible - registers, program stack, current instruction being executed etc.
- Doesn't distinguish between user code and library code.

C vs assembly datatypes in 64 bit

- (un)signed char, (u)int8
- (unsigned) short/(u)int16
- (unsigned) int/(u)int32
- (unsigned) long/(u)int64
- pointer(8 bytes)

- Byte - 8 bits
- Word - 16 bits
- Double word - 32 bits
- Quad word - 64 bits

Outline

Registers

- Assembly instructions fairly limited - single C line consists of multiple instructions.
- Registers - scratchpad area for processor to store intermediate results.
- Also used to pass first few arguments to a function being called.
- Why not use main memory for this?
  Reading/writing to memory is very slow.

Registers and their recommended uses

- **RAX** - Accumulator. Function/system call return value.
- **RBX** - Pointer to data section.
- **RCX** - Counter.
- **RDX** - I/O register.
- **RDI** - Destination index in repeated byte operations.
- **RSI** - Source index in repeated byte operations.

Registers and their recommended uses

- **RBP** - Base pointer. Points to bottom of current stack frame.
- **RSP** - Stack pointer. Points to top of current stack frame.
- **RIP** - Current Instruction pointer.
- **R8-R15** - Any computation/intermediate result storage purposes.
- **RFLAGS** - FLAGS registers. Holds information about result of certain operations. Used with conditional instructions.

Registers(cont.)

- All except RSP and RIP can store intermediate results.
- Above are general purpose registers(GPRs). More exist(not all may be present in all processors):
  - Segment registers: Point to process segments in memory.
  - Control registers: Manage processor and other states. Privileged.
  - 64 bit MMX registers: Floating point and SIMD instructions.
  - 128 bit XMM registers: SSE instructions.
  - 256 bit registers: AVX instructions.
- We won't discuss them though.

Accessing specific bits of GPRs

Almost all GPRs support accessing specific bits **starting from the least significant bit**.

64	**32**	**16**	**8**		**64**	**32**	**16**	**8**
rax	eax	ax	al		r8	r8d	r8w	r8b
rbx	ebx	bx	bl		r9	r9d	r9w	r9b
rcx	ecx	cx	cl		r10	r10d	r10w	r10b
rdx	edx	dx	dl		r11	r11d	r11w	r11b
rsi	esi	si	sil		r12	r12d	r12w	r12b
rdi	edx	dx	dil		r13	r13d	r13w	r13b
rbp	ebp	bp	bpl		r14	r14d	r14w	r14b
rsp	esp	sp	spl		r15	r15d	r15w	r15b

Outline

System V calling convention

- Calling convention: Set of rules caller and callee agree upon: how to pass arguments, how registers should be used, which registers should be saved by whom etc.
- 64 bit Linux and *nix distros use the System V calling convention.
- Windows follows a different calling convention. Many other conventions also exist.
- Calling convention varies with architecture and OS used.

System V calling convention(cont.)

- Argument order: RDI, RSI, RDX, RCX, R8, R9 and on stack **in reverse order**.
- Return value: RAX or (RDX:RAX) if value $>$ 64 bits.
- Caller save GPRs: RAX, RDI, RSI, RDX, RCX, R8, R9, R10 and R11.
- Callee save GPRs: RBX, RBP, RSP, R12, R13, R14, R15.

Sample programs!

Let's go through some sample programs to learn assembly programming.

Instructions we have seen

- **push**: Push a value onto program stack.
- **pop**: Pop a value from top of the program stack.
- **mov**: Move a value from one location to another.
- **call**: Save address of next instruction on top of program stack and start executing from address passed as argument.
- **ret**: Pop an address from top of program stack and start executing from that location.

Instructions we have seen(cont.)

- **leave**: Destroy stack frame and restore old *rbp*. Equivalent to *mov rsp, rbp ; pop rbp*.
- **add**: Add second argument to first argument. Store result in first argument.
- **inc**: Increment argument by 1.
- **cmp**: Compare the two arguments and set appropriate flags in RFLAGS register.

Instructions we have seen(cont.)

- **j****: Conditional or unconditional jump instructions.
- **xor**: XOR two arguments and store result in first argument.
- **test**: Compute bitwise AND of two arguments, set appropriate flags in RFLAGS and discard result of bitwise AND.

More useful instructions

- **nop**: Does nothing. Really - it just wastes a CPU cycle.
- **lea**: Load effective address. Eg: *lea rax, [rsi + 8]* same as $rax = rsi + 8$.
- **sub**: Subtract second argument from first argument and store the result in first argument.
- **imul, idiv**: Multiply/Divide the two arguments(second from first).
- **and, or, not**: Logical and, or and operations.

More useful instructions

- Several instructions exist in x64 instruction set.
- Impossible to learn and memorize all.
- Tip: Memorize most of the instructions we discussed. They are most commonly used.
- Any new instructions: Learn to read the processor manufacturer's manual to learn what the instruction does.

More about RFLAGS

- x64's FLAGS register. 64 bits in length.
- Used commonly in conditional statements.
- Set by arithmetic, logical and many other instructions.
- Most common flags: Zero, sign, overflow and carry. Many more exist but not important now.

Operand types

- 3 kinds of operands: Immediate, Register and Memory.
- **Immediate**: Absolute value. Eg: *push 0*.
- **Register**: Argument is name of a register. Eg: *not r15l*.
- **Memory**: Multiple ways exist to access a value in memory.
- All memory references must specify the size of value being accessed.

Operand types: memory

- **Absolute**: Modify a memory location using it's address. Eg: *not [0x400040]*.
- **Indirect**: Modify a memory location whose address is stored in a register. Eg: *not DWORD [rsp]*.
- **Base + displacement**: Eg: *not WORD [rsp + 4]*.
- **Indexed**: Eg: *not WORD [rsp + rcx]* and *not WORD [rsp + rcx + 1]*. Commonly used when accessing arrays in a loop.

Operand types: memory

Examples of scaled indexed memory access. Possible scale values: 1, 2, 4 and 8. Commonly used in accessing arrays of standard and custom datatypes in a loop.

- *not DWORD [rsp * 1]*
- *not DWORD [rsp + rcx * 2]*
- *not DWORD [rsp + rcx * 4 + 1]*
- *not DWORD [rsp + rcx * 8 - 1]*

Outline

Things to remember

- ASM programming: Writing CPU specific assembly code. Lower level $\Longrightarrow$ more details visible.
- Only 4 data types: BYTE, WORD, DWORD and QWORD.
- Many registers available: try to use them as much as possible.
- Sub-registers also available starting from LSB.
- Follow calling convention, specially when invoking library functions.

Things to remember(cont.)

- Memorize few common instructions and find out rest using processor manuals or other sources.
- *nasm -f elf64 <filname>.asm && gcc <filename>.o*.
- At most 1 memory location can be accessed in a single instruction. Specify size correctly.
- Multiple instructions can have same effect. Eg: *mov rax, 0* and *xor rax, rax*. Shorter instructions $\Longrightarrow$ reduced size.