Bluff your way in x64 assembler

Roger Orr

OR/2 Limited

How to survive an encounter with opcodes

– ACCU 2017 –

Where does assembler fit in?

- Most of us program in a high level language that is abstracted away from details of the hardware
- As Joel Spolsky wrote: “All non-trivial abstractions, to some degree, are leaky
- Let's take a “layered” look at an example program

Where does assembler fit in?

hello.cpp

```
#include <iostream>
int main() {
  std::cout << "Hello, world\n";
}
```

iostream

```
// Standard iostream objects -*- C++ -*-

// Copyright (C) 1997-2017 FSF, Inc.
//
// This file is part of the GNU ISO C++
...
```

Assembly language *(conceptually)*

```
main:   push    rbp
        mov     rbp, rsp
        mov     esi, OFFSET FLAT:.LC0
        ...
```

a.out (machine code)

```
... 55 48 89 E5  BE E5 07 40
    00 BF 60 10  60 00 E8 F6 ...
```

run-time libraries

libstdc++.so.6
libc.so.6
...

Operating System

Hardware

Where does assembler fit in?

- Three of the commonest uses of assembly language are:
  - During interactive debugging
  - Checking compiler optimisation
  - Understanding how a program runs

x86 introduction

- What does the CPU know about life?
    - ***Machine registers*** *and* ***Memory****
- The 8086 chip had 16-bit registers:
    - General: AX/BX/CX/DX/SI/DI/BP/SP
    - Segment: CS/DS/ES/SS
    - Special purpose: IP
    - Flags
- Memory: 16-bit segment + 16-bit offset
- *(*Ignoring all the details below that...)*

x86 introduction

- Additionally, four of the 16-bit general purpose registers (AX-DX) could be accessed as two 8-bit values:
    - Eg AH and AL
    - Where AX = AH<<8 + AL
- The 8087 floating point co-processor added a register *stack*, ST0 – ST7, holding 80-bit floating point values

x86 introduction

- The x86 16-bit architecture evolved into a 32-bit one for the 80386.
- The registers were Extended to 32bits:
  - General: EAX/EBX/etc.
  - Segment: added FS and GS
  - Special purpose: EIP
  - Eflags
- The old names for the general registers accessed the low 16-bit values

x86 introduction

- The x86 32-bit architecture evolved into a 64-bit one (originally designed by AMD)
- The general registers were widened to 64-bits and eight new ones added:
  - General: RAX/RBX/etc.
  - Addition of general registers R8-R15
  - Special purpose: RIP
- The old names for the general registers accessed the low 32-bit values
- Add 'D' to R8-R15 for the low 32-bit values

x86 introduction

- Meanwhile the floating point register stack was overlaid by 64-bit MMX registers
- Eight new 128-bit XMM registers were added, then increased to 16
- Widened to 256-bit YMM registers (32)
- Widened to 512-bit ZMM registers (32)
- There are also other registers, used for processor control, debug, and instrumentation

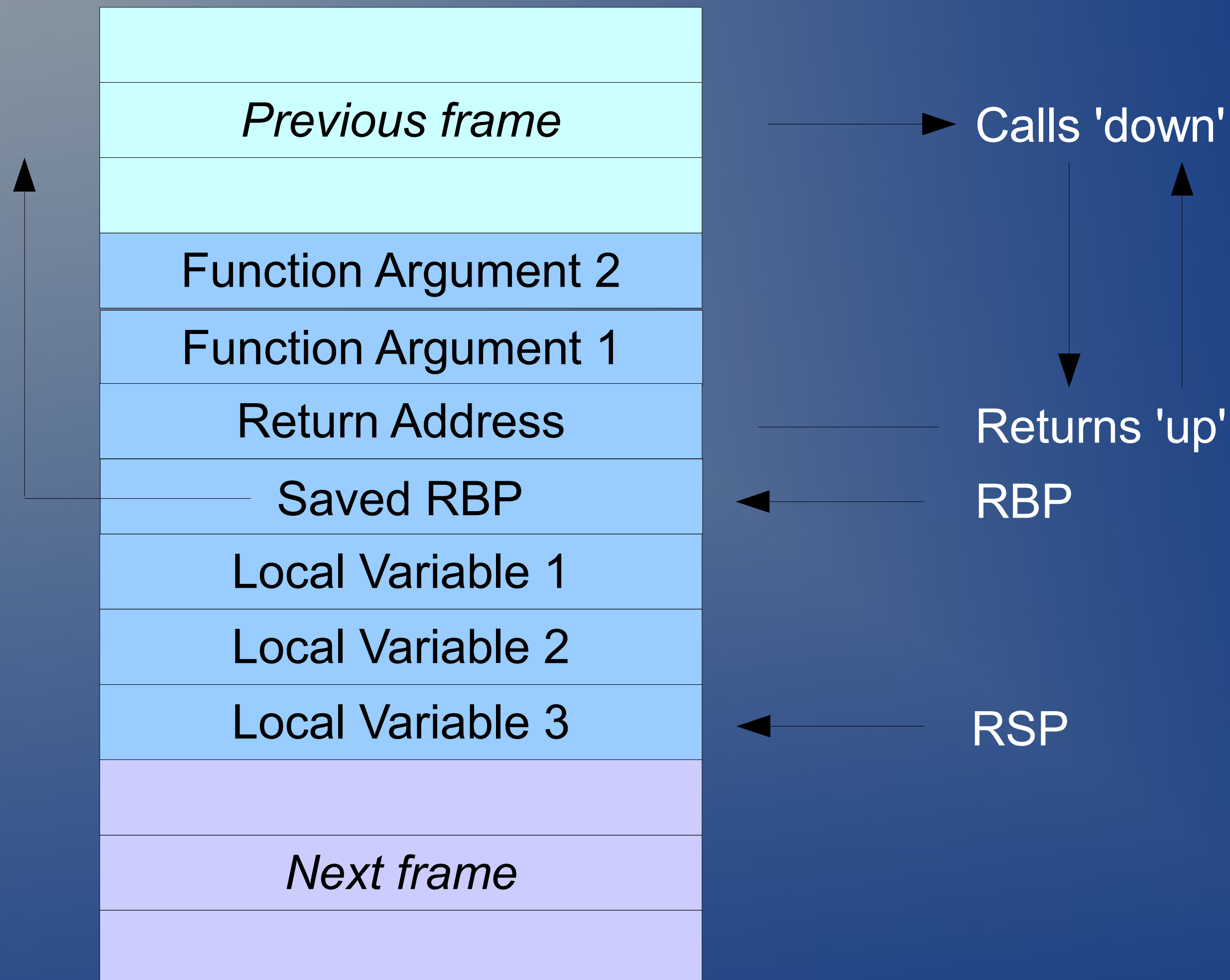
Don't Panic!!

- While there is a *lot* of complexity, much of it is not needed for the use cases we're focusing on:
  - control, debug & performance registers
  - segment registers (applications use a so-called “flat” address mode)
  - XMM, YMM, ZMM registers

The stack frame

- When a program makes a function call it:
    - Sets a new program location
    - Stores where it came from
    - Passes arguments from the caller
    - Reserves locations for local variables
- In x64 this is all managed by a “stack frame”, at least in principle

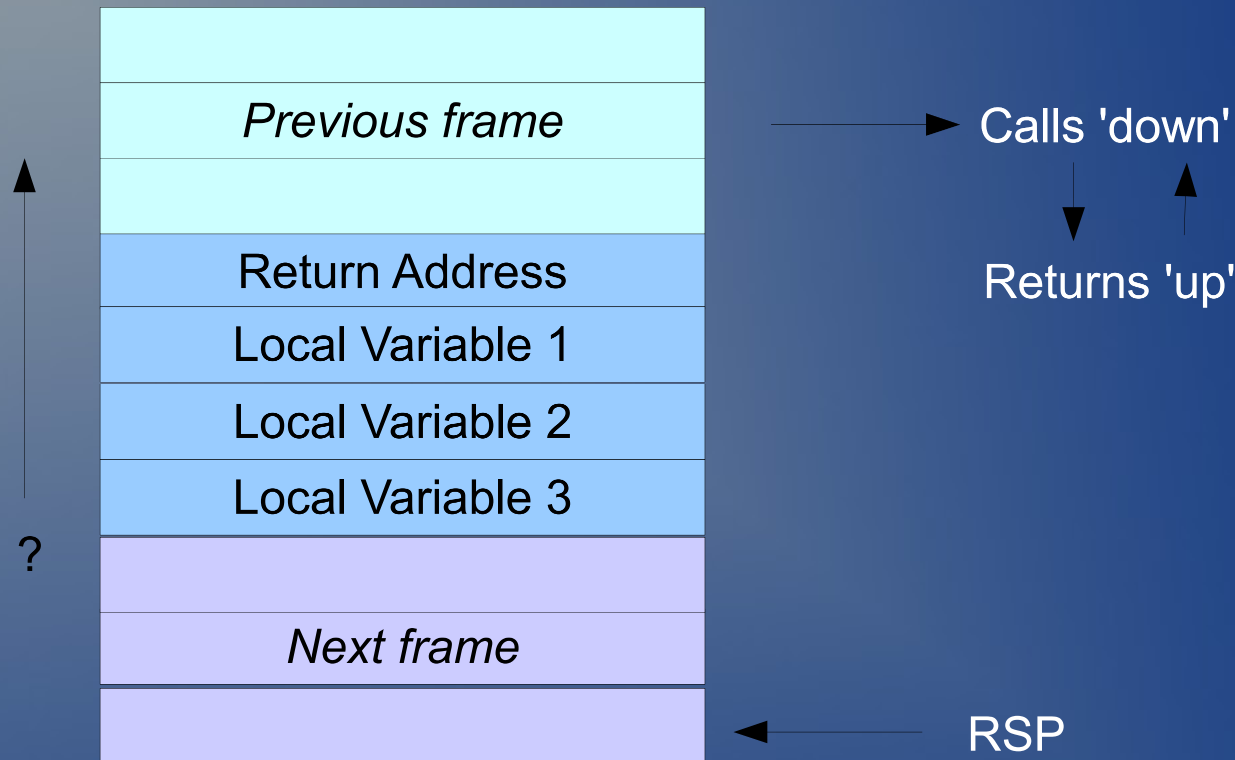
The ideal stack frame



The stack frame

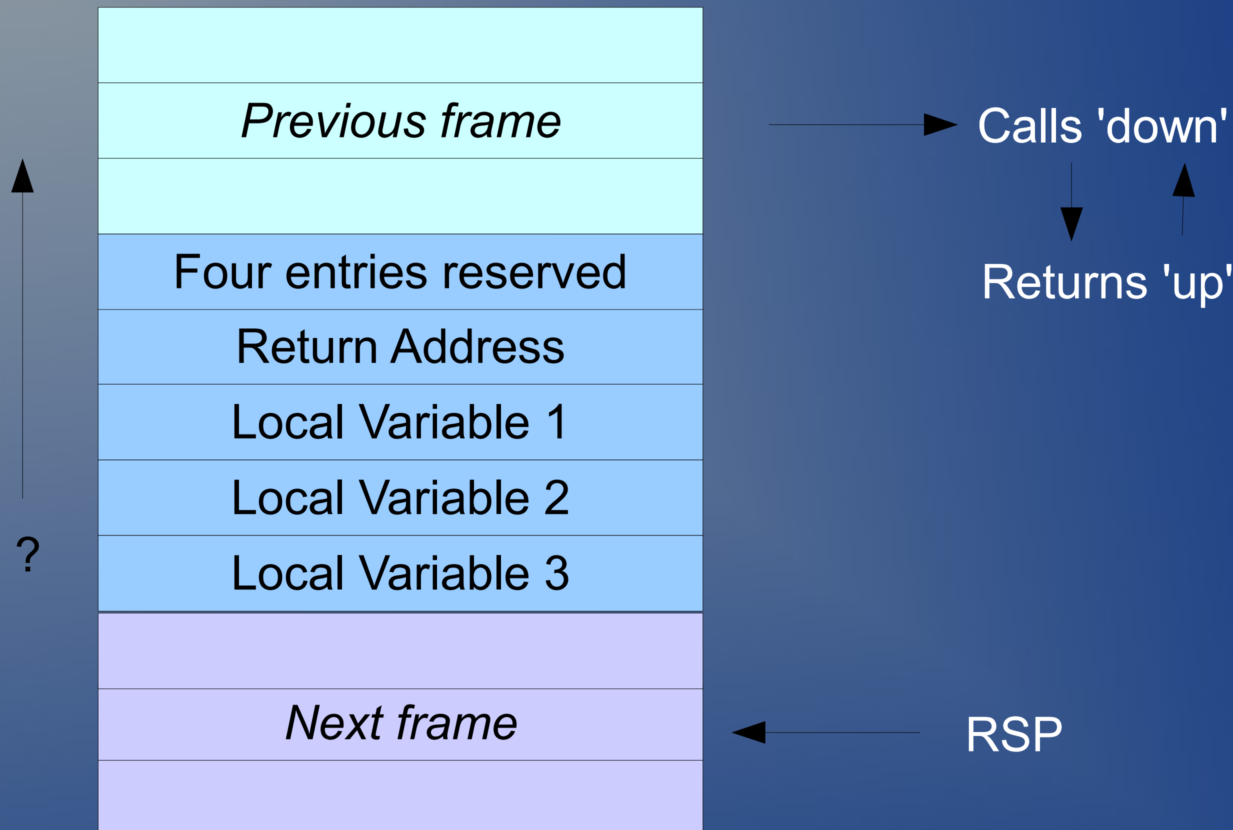
- However, the x64 conventions both pass some arguments in registers, for speed.
- Both conventions also define:
  - The register used for return values
  - The 'non-volatile' registers (that must be persisted over the call)
  - The 'volatile' registers (that a function can leave in an arbitrary state)

A real (Linux) stack frame



On Linux the first 6 arguments to a function are passed in registers:
RDI, RSI, RDX, RCX, R8, R9, so no stack might be needed for arguments
The compiler can move RSP to where the next call needs it, and fill in extra arguments by using addresses relative to RSP
The previous stack frame can be restored from RSP, so RBP can be used as a general purpose register

A real (Windows) stack frame



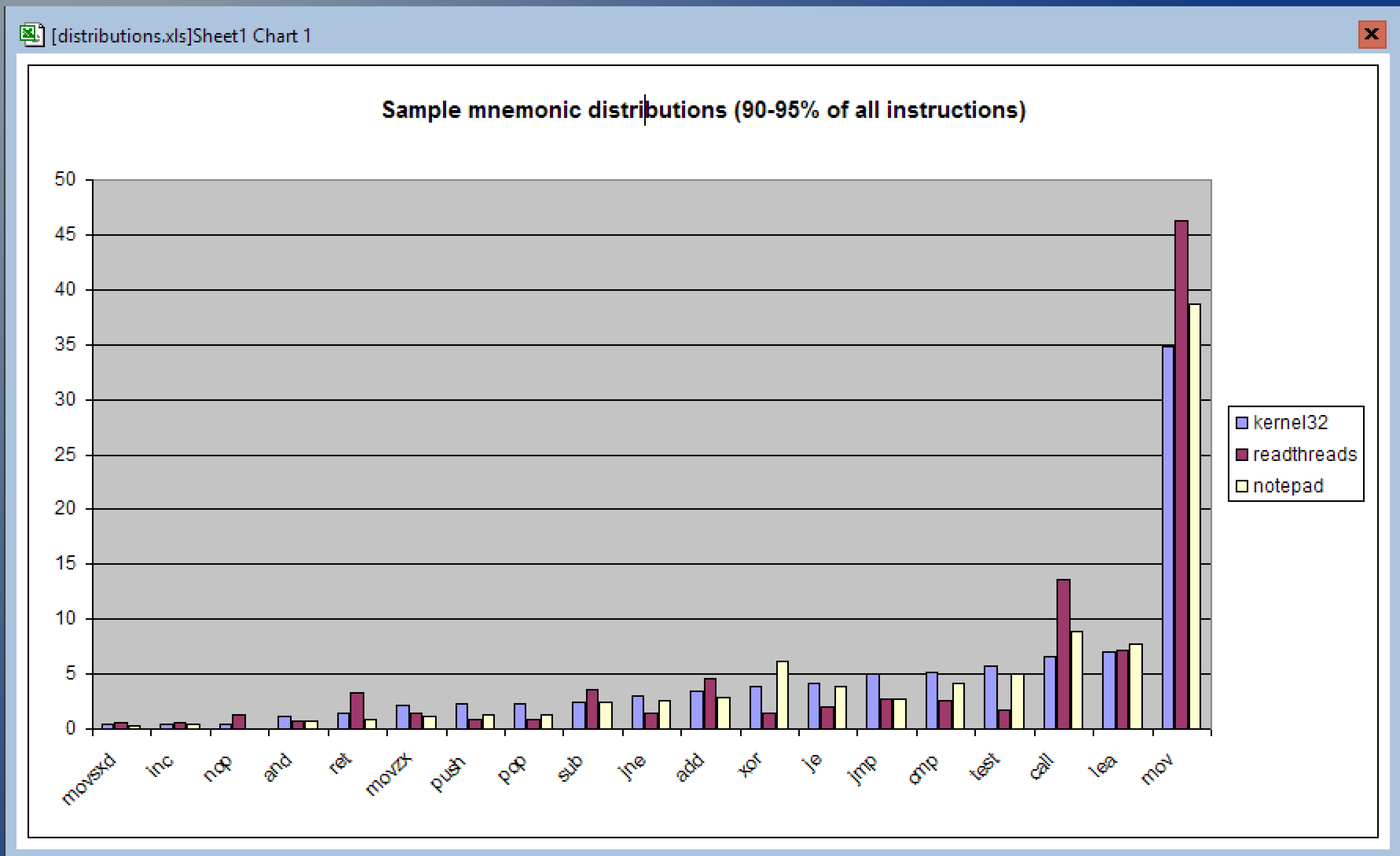
On Windows the first 4 arguments to a function are passed in registers: RCX, RDX, R8, R9. However 'shadow' stack space is always reserved, unless it is a 'leafl' function (makes no other calls).
As before, the frame address need not be saved, nor RBP used to hold it, provided it can be restored on exit

Instructions

- The x64 instruction set has a large number of different instructions
- Fortunately for bluffing, the vast majority of the assembler instructions in most compiled programs come from a very small subset
- I took the disassembly from three different binaries and analysed the opcodes used

  50-60% of the program was comprised of only 3 different instructions
- 90-95% of the program was comprised of less than 20 different instructions

Instructions



Top three instructions

- `mov` – roughly equivalent to assign. Can copy 64-bit (or other sized) values between registers and memory or between registers
- `lea` – a strange instruction: unpacked later
- `call` – make a function call. The target address can be a 32-bit relative address, or held in a register, or be referenced indirectly

Two different dialects

- Sadly the syntax has two different forms: Intel and AT&T. Here are a couple of `mov` examples showing some of the differences

AT&T

```
mov $5, %rax
movl $0x5,-0x44(%rbp)
```

Source before destination
Mnemonic indicates size
$/% prefix for numbers/registers
Round brackets for addressing

Common on Linux

Intel

```
mov rax, 5
mov dword ptr [rbp-0x44],0x5
```

Destination before source
Size inferred, or using keyword
No prefix for numbers and registers
Square brackets for addressing

Common on Windows

mov addressing

- The mov instruction can use some simple arithmetic when calculating the memory address.
- Given
  - `struct s { int v1; int v2; };`
  - `s *p;`
- Then accessing '`p[idx].v2`' could make use of an expression like: `[rdi+4+rsi*8]`
- This means some well-meaning attempts to optimise high level code may be ineffective (or even counter-productive)

Other popular instructions

- `test/cmp` – compare two operands. Test is like `and` and `cmp` is like `sub`, but the result is discarded
- `push/pop` – write value to stack and decrement the stack pointer/increment the stack pointer
- `jmp` – like goto. The target address specified like with `call`
- `add/sub/mul/div` – I expect you can guess

How do we get assembler?

- Writing it ourselves (not really suitable for bluffing...)
- Compiling high-level language source code
  - Most C++ compilers have a switch
  - The output may or may not be complete
- Disassembling binary data
  - For example, in a debugger when source code can't be found
  - Not always straightforward

Assembler from source

- g++ -S hello.cpp
  - Generates 98 lines
  - gcc hello.s -o hello.exe -lstdc++
- cl -FA hello.cpp
  - Generates 6,704 lines (!)
  - Output won't assemble (with ml.exe)
- In both cases finding the opcodes we are interested in can be non-trivial

Assembler from binary

- For example, debugging a program with no symbolic information using gdb:

```
                 [ No Source Available ]

>  0x7ffff7b049b0 <__read_nocancel+7>     cmp    $0xfffffffffffff001,%rax
   0x7ffff7b049b6 <__read_nocancel+13>    jae    0x7ffff7b049e9 <read+73>
   0x7ffff7b049b8 <__read_nocancel+15>    retq
   0x7ffff7b049b9 <read+25>               sub    $0x8,%rsp
   0x7ffff7b049bd <read+29>               callq  0x7ffff7b22810 <__libc_enabl
   0x7ffff7b049c2 <read+34>               mov    %rax,(%rsp)

native process 7959 In: __read_nocancel                L84   PC: 0x7ffff7b049b0

(gdb) layout next
(gdb)
```

Assembler from binary

- For example, debugging a program with no symbolic information using WinDbg:

```
Breakpoint 1 hit
cmd!main:
00000000`4a158494 488bc4          mov     rax,rsp
0:000> u
cmd!main:
00000000`4a158494 488bc4          mov     rax,rsp
00000000`4a158497 48895808        mov     qword ptr [rax+8],rbx
00000000`4a15849b 48896810        mov     qword ptr [rax+10h],rbp
00000000`4a15849f 48897018        mov     qword ptr [rax+18h],rsi
00000000`4a1584a3 48897820        mov     qword ptr [rax+20h],rdi
00000000`4a1584a7 4155            push    r13
00000000`4a1584a9 4156            push    r14
00000000`4a1584ab 4157            push    r15

0:000>
```

Assembler from binary

- Sometimes disassembly can be inaccurate
  - x64 instructions are of variable length
  - code and data can be mixed
- In particular disassembling *backwards* to find how you got to where you are may not be simple
- It may be simpler to disassemble *forwards* from the start of the current function

Assembler from binary

- Example output when starting disassembly **not** on an instruction boundary:

```
Breakpoint 1 hit
cmd!main:
00000000`4a158494 488bc4          mov     rax,rsp
0:000> u rip+5
cmd!main+0x5:
00000000`4a158499 58              pop     rax
00000000`4a15849a 084889          or      byte ptr [rax-77h],cl
00000000`4a15849d 6810488970      push    70894810h
00000000`4a1584a2 184889          sbb     byte ptr [rax-77h],cl
00000000`4a1584a5 7820            js      cmd!main+0x33 (00000000`4a1584c7)
00000000`4a1584a7 4155            push    r13
00000000`4a1584a9 4156            push    r14
00000000`4a1584ab 4157            push    r15

0:000>
```

Assembler from binary

- Usually much better when you don't need to use assembler when debugging.
- Check your tool chain for how to build with symbolic information – and how to make sure this is associated correctly in the debugger with the code being examined
- May wish to split out debugging information to reduce disk space usage and/or control access
- Find out options for debugging symbols for operating system and third party components.

Compiler Explorer

Compiler Explorer - C++

Secure | https://gcc.godbolt.org

Compiler Explorer C++ Editor Diff View More Share Help

C++ source #1

```cpp
// Type your code here, or load an example.
int square(int num) {
    return num * num;
}
```

x86-64 gcc 6.3 (Editor #1, Compiler #1)

x86-64 gcc 6.3 | compiler options...

11010 .LX0: .text // Intel

```asm
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

g++ (GCC-Explorer-Build) 6.3.0- *cached*

Compiler Explorer

- Left pane – type your source code
  - Automatically compiled as you type
  - Set compiler options as you choose
- Right pane – displays the assembler output
  - Compiler Explorer filters down the output to focus on the important part
- Supports many different compilers
- Can be fetched from git and run locally
  - Finds local compilers
  - Can configure non-standard locations

Compiler Explorer - locally

- Download & install VirtualBox from www.virtualbox.org
- Download & install Ubuntu from www.ubuntu.com
- `sudo apt install npm`
- `sudo apt install node.js`
- `sudo apt install git`
- `git clone https://github.com/mattgodbolt/compiler-explorer`
- `make`
- Then visit `http://localhost:10240`
- `sudo apt install clang`
  - Compiler explorer automatically finds it

Compiler Explorer

- Demonstrate Compiler Explorer
- Live demo includes:
    - See the stack frame set up
    - Look at arguments and local variables
    - Look at some trivial optimiser use
    - Where do the local variables go?

LEA ("Load Effective Address")

- The `lea` instruction allows memory addressing calculations without actually accessing any memory
- As with `mov`, the various addressing modes allow use of two or three operands, and to put the result anywhere (unlike `add`)
- Does not affect flags, and so friendly to re-ordering /pipe-lining

LEA (“Load Effective Address”)

- The 'addresses' are not used to access memory and so the instruction can be used for addition of any integer values
- This can allow the compiler to use the `lea` instruction for basic arithmetic

Walking the stack

- The *call stack* is an important piece of information. It is useful when debugging to see how you got to the current location and it is required when transferring control by exception.
- In general this is a hard problem; each function only knows how to return its caller
- The solution for X64 is additional meta-data which is held in the binary and allows the stack frames to be deduced at runtime

Walking the stack

- The extra data is generated in the assembler using various directives
- The gcc and clang directives are prefixed with `.cfi_`, the MSVC directives have no common prefix
- While it **is** possible to walk the stack manually this is not 'bluff your way' material.
- You fortunately do not normally need to do so; debuggers use the meta-data to work up the stack identifying each stack frame and return address

Walking the stack

- Note that since debuggers use the meta-data from the binaries themselves, examining dump files may require access to the binaries matching those on the machine where the dump was taken
- Better to check this *before* rather than *after* you receive your first important production dump file...

More information - Linux

- The data structures are held in the `.eh_frame` section of the binary
- There are libraries to help unwind the stack (e.g. http://www.nongnu.org/libunwind/)
- You find the meta-data for a target address, which identifies the register and offset for the current frame address and where the non-volatile registers are stored
- Repeat using the return address from the frame as the target address

More information - Windows

- The data structures are held in the `.pdata` section and this can be examined with `dumpbin /unwindinfo`
- There is a debugging method, `StackWalk64` from dbghelp.dll to help with unwinding the stack
- There is a lower level call, `RtlVirtualUnwind`
- You can even add entries dynamically using `RtlAddFunctionTable` (few people need to do this; it's needed to support dynamically creating functions at runtime)

Conclusion

- While mastering the full range of X64 assembler is hard, a basic understanding of
  - common registers,
  - the calling convention, and
  - a handful of instructions
- will enable you to get a long way in those cases where you do need to delve into assembler
- There is a *lot* of data in assembler mode, tools such as compiler explorer may help you focus on the relevant parts