Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014-2015
xeno@legbacore.com
All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64' class, available at http://OpenSecurityTraining.info/IntroX86-64.html"
ExampleSubroutine1.c

The stack frames in this example will be very simple.
Only saved return addresses (RIP).

//ExampleSubroutine1:
// using the stack & subroutine
to call subroutines
//New instructions:
push, pop, call, ret, mov
int func(){
    return 0xbeef;
}
int main(){
    func();
    return 0xf00d;
}
CALL - Call Procedure

- CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off
- First it pushes the address of the next instruction onto the stack
  - For use by RET for when the procedure is done
- Then it changes RIP to the address given in the instruction
- Destination address can be specified in multiple ways
  - Absolute address
  - Relative address (relative to the end of the instruction, or some other register)
RET - Return from Procedure

- Two forms
  - Pop the top of the stack into RIP (remember, pop increments stack pointer, RSP)
    - In this form, the instruction is just written as “ret”
  - Pop the top of the stack into RIP and also add a constant number of bytes to RSP
    - In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc
**Intel vs. AT&T Syntax**
(we'll come back to this again much later)

- **Intel: Destination <- Source(s)**
  - Windows. Think algebra or C: \( y = 2x + 1 \);
  - `mov rbp, rsp`
  - `add rsp, 0x14 ; (rsp = rsp + 0x14)`

- **AT&T: Source(s) -> Destination**
  - *nix/GNU. Think elementary school: \( 1 + 2 = 3 \)
  - `mov %rsp, %rbp`
  - `add $0x14, %rsp`
  - So registers get a % prefix and immediates get a $

- My classes will use Intel syntax except in this section
- But it's important to know both, so you can read documents in either format.
MOV - Move

• Can move:
  – register to register
  – memory to register, register to memory
  – immediate to register, immediate to memory
• Never memory to memory!
• Memory addresses are given in r/mX form talked about next
“r/mX” Addressing Forms

• Anywhere you see an r/mX it means it could be taking a value either from a register, or a memory address.
• I’m just calling these “r/mX forms” because anywhere you see “r/m16”, “r/m32”, or “r/m64” in the manual, the instruction can be a variation of the below forms.
• In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like dereferencing a pointer)
  – mov rax, rbx
  – mov rax, [rbx]
  – mov rax, [rbx+rcx*X] (X=1, 2, 4, 8)
  – mov rax, [rbx+rcx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
• Most complicated form is: [base + index*scale + disp]

More info: Intel v2a, Section 2.1.5 page 2-4
in particular Tables 2-2 and 2-3
ADD and SUB

- Adds or Subtracts, just as expected
- Destination operand can be r/mX or register
- Source operand can be r/mX or register or immediate
- No source and destination as r/mXs, because that could allow for memory to memory transfer, which isn’t allowed on x86
- Evaluates the operation as if it were on signed AND unsigned data, and sets flags as appropriate. Instructions modify OF, SF, ZF, AF, PF, and CF flags for what it’s worth
- add rsp, 8 == (rsp = rsp + 8)
- sub rax, [rbx*2] == (rax = rax - memorypointedtoby(rbx*2))
ExampleSubroutine1.c 1:
EIP = 00000001`40001010, but no instruction yet executed

<table>
<thead>
<tr>
<th>Key:</th>
</tr>
</thead>
<tbody>
<tr>
<td>⭕ executed instruction,</td>
</tr>
<tr>
<td>⬆️ modified value</td>
</tr>
<tr>
<td>⬇️ start value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>func:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001400001000 mov eax,0BEEFh</td>
</tr>
<tr>
<td>00000001400001005 ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001400001010 sub rsp,28h</td>
</tr>
<tr>
<td>00000001400001014 call func (0140001000h)</td>
</tr>
<tr>
<td>00000001400001019 mov eax,0F000h</td>
</tr>
<tr>
<td>0000000140000101E add rsp,28h</td>
</tr>
<tr>
<td>00000001400001022 ret</td>
</tr>
</tbody>
</table>

| Belongs to the frame *before* main() is called |
| Belongs to the frame *before* main() is called |

Colon notation means the full value is represented by the concatenation of the two values. If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444.
ExampleSubroutine1.c 2:

Key:
- Executed instruction
- Modified value
- Start value

Colon notation means the full value is represented by the concatenation of the two values.
If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

```c
func:
0000000140001000  mov   eax,0BEFH
0000000140001005  ret

main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```
ExampleSubroutine1.c 3:

rax  000007fe'f239c3a8
rsp  00000000'0012fe88

Key:
- executed instruction
- modified value
- start value

```
func:
0000000140001000  mov   eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func(0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```

Colon notation means the full value is represented by the concatenation of the two values. If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444.
ExampleSubroutine1.c 4:

```
func:
0000000140001000  mov   eax,0BEEFh ⬇
0000000140001005  ret

main:
0000000140001010  sub   rsp,28h
0000000140001014  call  func (0140001000h)
0000000140001019  mov   eax,0F00Dh
000000014000101E  add   rsp,28h
0000000140001022  ret
```

Note that it "zero extended" the reg (meaning it filled in the upper 32 bits of RAX with zeros)

Colon notation means the full value is represented by the concatenation of the two values.
If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444

Key:
- ⬇ executed instruction
- ⬆ modified value
- ⬆ start value
ExampleSubroutine1.c 4:

Key:
- executed instruction,
- modified value
- start value

Note that it "zero extended" the reg (meaning it filled in the upper 32 bits of RAX with zeros)

From section 3.4.1.1 in the June 2014 Manual included with class materials:

When in 64-bit mode, operand size determines the number of valid bits in the destination general-purpose register:

- 64-bit operands generate a 64-bit result in the destination general-purpose register.
- 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.
- 8-bit and 16-bit operands generate an 8-bit or 16-bit result. The upper 56 bits or 48 bits (respectively) of the destination general-purpose register are not modified by the operation. If the result of an 8-bit or 16-bit operation is intended for 64-bit address calculation, explicitly sign-extend the register to the full 64-bits.
ExampleSubroutine1.c:
STACK FRAME TIME OUT

func:
  mov   eax,0BEEFh
  ret

main:
  sub   rsp,28h
  call  func (0140001000h)
  mov   eax,0F00Dh
  add   rsp,28h
  ret

"Function-before-main"'s frame

main's frame

func never even makes a stack frame

Colon notation means the full value is represented by the concatenation of the two values.
If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444
ExampleSubroutine1.c 5:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, 0BEEFh</td>
<td>00000001`40001000</td>
<td>def</td>
</tr>
<tr>
<td>ret</td>
<td>00000001`40001005</td>
<td>executed</td>
</tr>
</tbody>
</table>

Key:
- ⌫ executed instruction
- ⭪ modified value
- ⬆ start value

func:
- mov eax, 0BEEFh
- ret

main:
- sub rsp, 28h
- call func (0140001000h)
- mov eax, 0F00Dh
- add rsp, 28h
- ret

Colon notation means the full value is represented by the concatenation of the two values. If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444.
## ExampleSubroutine1.c 6:

### Key:
- ⬤ executed instruction
- ⬤ modified value
- ⬤ start value

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>00000000</code></td>
<td><code>-0012FEB8</code></td>
<td><code>mov</code> eax,0BEEFh</td>
</tr>
<tr>
<td><code>00000000</code></td>
<td><code>-0012FE88</code></td>
<td><code>ndef</code></td>
</tr>
<tr>
<td><code>00000000</code></td>
<td><code>-0012FE80</code></td>
<td><code>ndef</code></td>
</tr>
<tr>
<td><code>00000000</code></td>
<td><code>-0012FE78</code></td>
<td><code>undef</code></td>
</tr>
</tbody>
</table>

### func:
- `mov` eax,0BEEFh
- `ret`

### main:
- `sub` rsp,28h
- `call` func (0140001000h)
- `mov` eax,0F00Dh
- `add` rsp,28h
- `ret`

---

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444
ExampleSubroutine1.c 7:

```
rax    000000000000f00d
rsp    0000000000012feb8

func:
0000000140001000 mov  eax,0BEEFh
0000000140001005 ret

main:
0000000140001010 sub  rsp,28h
0000000140001014 call func (0140001000h)
0000000140001019 mov  eax,0F00Dh
000000014000101E add  rsp,28h ⬆
0000000140001022 ret
```

Key:
- ⬇ executed instruction
- ⬆ modified value
- ⬇ start value

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000`0012FEB8</td>
<td>00000000`400012ed</td>
</tr>
<tr>
<td>00000000`0012FE88</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FE80</td>
<td>undef</td>
</tr>
<tr>
<td>00000000`0012FE78</td>
<td>undef</td>
</tr>
</tbody>
</table>
```

Colon notation means the full value is represented by the concatenation of the two values.

If rdx = 0x11112222 and eax = 0x33334444, then rdx:eax is the quadword 0x1111222233334444.
ExampleSubroutine1.c 8:

```
rax  00000000'0000f00d
rsp  00000000'0012fec0

func:
  0000000140001000  mov   eax,0BEEFh
  0000000140001005  ret

main:
  0000000140001010  sub   rsp,28h
  0000000140001014  call  func (0140001000h)
  0000000140001019  mov   eax,0F00Dh
  000000014000101E  add   rsp,28h
  0000000140001022  ret

Execution would continue at the value ret removed from the stack: 00000001'400012ed
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000'0012FEB8</td>
<td>undef</td>
<td>executed instruction, modified value, start value</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000'0012FE90</td>
<td>undef</td>
<td></td>
</tr>
<tr>
<td>00000000'0012FE88</td>
<td>undef</td>
<td></td>
</tr>
<tr>
<td>00000000'0012FE80</td>
<td>undef</td>
<td></td>
</tr>
<tr>
<td>00000000'0012FE78</td>
<td>undef</td>
<td></td>
</tr>
</tbody>
</table>

Key:
- ⏯️ executed instruction,
- 🚳 modified value
- ⬆️ start value
ExampleSubroutine1 Notes

• func() is dead code - its return value is not used for anything, and main() always returns 0xF00D. If optimizations were turned on in the compiler, it would remove func()
• We don’t yet understand why main() does “sub rsp,28h” & “add rsp,28h”… We will figure that out later.
Let’s do that in a tool

- Visual C++ 2012 Express edition (which I will shorthand as “VisualStudio” or VS)
- Standard Windows development environment
- Available for free, but missing some features that pro developers might want
- Keep in mind you can’t move express-edition-compiled applications to other systems and get them to run without first installing the “redistributable libraries”
Adding files to the project
Creating a new project - 4
Right click, select Add
ExampleSubroutine1.cpp
ExampleSubroutine1.h
Add
Cancel
Unfortunately the debug information format alters the code which gets generated too much, making it not as simple as I would like for this class.

Set SDL checks to **No** ("/sdl") in VS2013.
These would just add extra complexity to the asm which we don't want for now.
This shouldn't matter, but setting it just in case…
Different options can be set for release vs debug builds.

The GUI is just a wrapper to set command line options.

Click this to change which config set is active.
Click this box to change which config set is active.

And then select “New” here.
Pull down to select x64 build platform option

Very important! This box should be un-checked

Once it's selected, hit OK
Setting project properties - 6

This adds an extra jump between a call and the target function.
Disable Address Space Layout Randomization (ASLR) so that we see the same addresses in our labs.
2 ways to build the project

Right click on project and select build

Or select Build Only ... from the menu bar
Information about whether the build succeeded will be in the Output window. If it fails, a separate Error tab will open up.
Setting breakpoints & start debugger

Click to the left of the line to break at. A red circle will appear.
Debugging interface
Showing assembly

Right click: Only available while debugging

Current stopped location
Watching registers ("watch")

In the "Watch" tab you can enter register names or variable names.
Watching registers ("autos")

Note that it knows the RSP register is going to be modified by this instruction.
Watching the stack change - 1
Watching the stack change - 2

1. Right click on the body of the data in the window and make sure everything's set like this.
2. Set address to rsp (will always be the top of the stack).
3. Set to 8 byte.
4. Click “Reevaluate Automatically” so that it will change the display as rsp.
5. Set to 1.
ExampleSubroutine1.c takeaways

- In VS (when optimization is turned off), there is an over-allocation of stack space as a result of calling a function
  - 0x28 reserved with no apparent storage of data on the stack
  - More about this later once we start passing function parameters

```c
int func(){
    return 0xbeef;
}

int main(){
    func();
    return 0xf00d;
}
```

```
<table>
<thead>
<tr>
<th></th>
<th>func:</th>
<th>main:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000140001000</td>
<td>mov</td>
<td>mov</td>
</tr>
<tr>
<td>0000000140001005</td>
<td>ret</td>
<td>ret</td>
</tr>
<tr>
<td>0000000140001010</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>0000000140001014</td>
<td>call</td>
<td></td>
</tr>
<tr>
<td>0000000140001019</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>000000014000101E</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>0000000140001022</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
```

```assembly
func:
0000000140001000  mov      eax,0BEEFh
0000000140001005  ret
main:
0000000140001010  sub      rsp,28h
0000000140001014  call     func (0140001000h)
0000000140001019  mov      eax,0F00Dh
000000014000101E  add      rsp,28h
0000000140001022  ret
```
Instructions we now know (8)

- NOP
- PUSH/POP
- CALL/RET
- MOV
- ADD/SUB