Assembly

Where we stand and where we go

Cauchy Pu

Outline

Overview

Computers execute machine code, sequences of bytes encoding the low-level operations that manipulate data, manage memory, read and write data on storage devices, and communicate over networks.

Bits and Where

Information = Bits + Context

So we study what bits and in which we view them

Assembly!

Why assembly

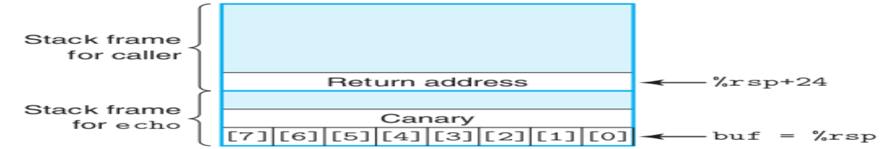
But, compilers do most of the work in generating assembly code, so why sould we spend our time?

1. Optimization. Sometimes we have to try the assembly code corresponding to various forms of upper language.
2. Some bugs more obvious in assembly. For exmaple, concurrent programs.
3. Security. Overwriting information is a common attacked method.
4. Some code must be assembly, context switch.
5. When you encounter core dump, what is the corresponding high level code?
6. And you told me, we are low-level engineers, right? So, here we go, assembly!
7. ......

Think like a computer

Why assembly

Example




```
echo:
    subq    $24, %rsp                Allocate 24 bytes on stack
    movq    %fs:40, %rax             Retrieve canary
    movq    %rax, 8(%rsp)            Store on stack
    xorl    %eax, %eax               Zero out register
    movq    %rsp, %rdi               Compute buf as %rsp
    call    gets                     Call gets
    movq    %rsp, %rdi               Compute buf as %rsp
    call    puts                     Call puts
    movq    8(%rsp), %rax            Retrieve canary
    xorq    %fs:40, %rax             Compare to stored value
    je      .L9                      If =, goto ok
    call    __stack_chk_fail         Stack corrupted!
  .L9:                             ok:
    addq    $24, %rsp                Deallocate stack space
    ret
```

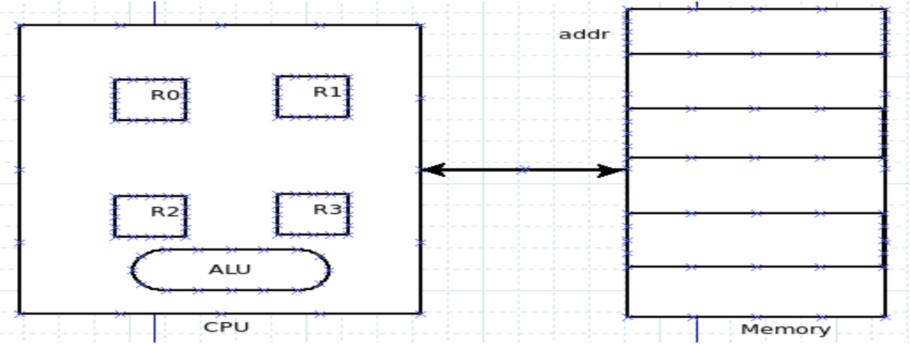
Why assembly

Example

```
GLOBAL_ENTRY(ia64_switch_to)
	.prologue
	alloc r16=ar.pfs,1,0,0,0
	DO_SAVE_SWITCH_STACK
	.body

	adds r22=IA64_TASK_THREAD_KSP_OFFSET,r13
	movl r25=init_task
	mov r27=IA64_KR(CURRENT_STACK)
	adds r21=IA64_TASK_THREAD_KSP_OFFSET,in0
	dep r20=0,in0,61,3		// physical address of "next"
	;;
	st8 [r22]=sp			// save kernel stack pointer of old task
	shr.u r26=r20,IA64_GRANULE_SHIFT
	cmp.eq p7,p6=r25,in0
	;;
	/*
	 * If we've already mapped this task's page, we can skip doing it again.
	 */
(p6)	cmp.eq p7,p6=r26,r27
(p6)	br.cond.dpnt .map
	;;
.done:
	ld8 sp=[r21]			// load kernel stack pointer of new task
	MOV_TO_KR(CURRENT, in0, r8, r9)		// update "current" application register
	mov r8=r13			// return pointer to previously running task
	mov r13=in0			// set "current" pointer
	;;
	DO_LOAD_SWITCH_STACK

#ifdef CONFIG_SMP
	sync.i				// ensure "fc"s done by this CPU are visible on other CPUs
#endif
	br.ret.sptk.many rp		// boogie on out in new context

.map:
	RSM_PSR_IC(r25)			// interrupts (psr.i) are already disabled here
	movl r25=PAGE_KERNEL
	;;
	srlz.d
	or r23=r25,r20			// construct PA | page properties
	mov r25=IA64_GRANULE_SHIFT<<2
```

A simple world

In the world of assembly...



Simple, right?

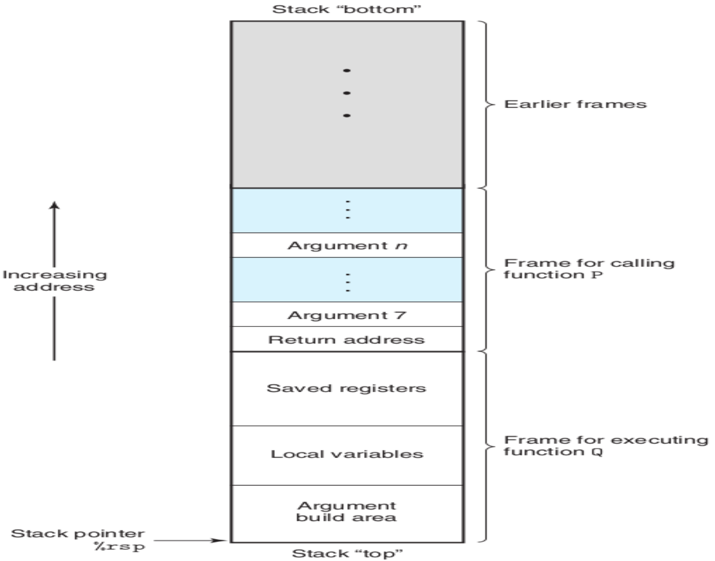
A simple world

The class of instructions:

1. assessing
   - mov
   - ldl
   - stl
2. arithmetic and logical operations
   - add
   - sub
3. control & procedures
   - jump
   - call
   - ret

A simple world

How stack operations...



A simple world

I just, want struct, array and many functions...

Expression	Type	Value	Assembly code
`E`	`int *`	x_E	`movl %rdx,%rax`
`E[0]`	`int`	$M[x_E]$	`movl (%rdx),%eax`
`E[i]`	`int`	$M[x_E + 4i]$	`movl (%rdx,%rcx,4),%eax`
`&E[2]`	`int *`	$x_E + 8$	`leaq 8(%rdx),%rax`
`E+i-1`	`int *`	$x_E + 4i - 4$	`leaq -4(%rdx,%rcx,4),%rax`
`*(E+i-3)`	`int`	$M[x_E + 4i - 12]$	`movl -12(%rdx,%rcx,4),%eax`
`&E[i]-E`	`long`	i	`movq %rcx,%rax`

Complete example:

rax = rdx + 4 * rcx - 12

This is the so-called addressing mode

A simple world

I just, want struct, array and many functions... So, our Starring!

Row	Element	Address
A[0]	A[0][0]	x_A
	A[0][1]	$x_A + 4$
	A[0][2]	$x_A + 8$
A[1]	A[1][0]	$x_A + 12$
	A[1][1]	$x_A + 16$
	A[1][2]	$x_A + 20$
A[2]	A[2][0]	$x_A + 24$
	A[2][1]	$x_A + 28$
	A[2][2]	$x_A + 32$
A[3]	A[3][0]	$x_A + 36$
	A[3][1]	$x_A + 40$
	A[4][2]	$x_A + 44$
A[4]	A[4][0]	$x_A + 48$
	A[4][1]	$x_A + 52$
	A[4][2]	$x_A + 56$

How calculation

T A[R][C]

&A[i][j] = $x_A + L(C * i + j)$

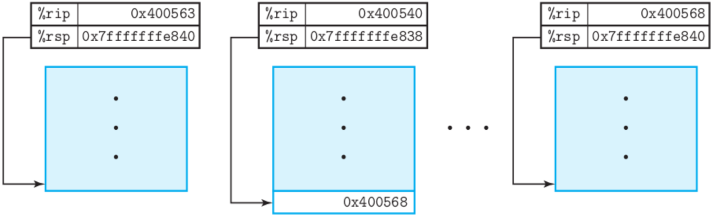
$L: the \ \ size \ \ of \ \ element \ \ of \ \ D$

```
A in %rdi, i in %rsi, and j in %rdx
leaq    (%rsi,%rsi,2), %rax     Compute 3i
leaq    (%rdi,%rax,4), %rax     Compute x_A + 12i
movl    (%rax,%rdx,4), %eax     Read from M[x_A + 12i + 4]
```

Really simple, right? ;-)

A simple world

Eventually, it’s the mysterious of function call!



(a) Executing `call` (b) After `call` (c) After `ret`

```
  Beginning of function multstore
  0000000000400540 <multstore>:
    400540:  53                    push   %rbx
    400541:  48 89 d3              mov    %rdx,%rbx
  . . .
  Return from function multstore
    40054d:  c3                    retq
  . . .
  Call to multstore from main
    400563:  e8 d8 ff ff ff        callq  400540 <multstore>
    400568:  48 8b 54 24 08        mov    0x8(%rsp),%rdx
```

A simple world

Now some real feed...

Example

```
.section .data
data_items:
.long 3,67,34,222,45,75,54,34,44,33,22,11,66,0
.section .text
.global _start
_start:
movl $0, $edi
movl data_items(,%edi, 4), %eax
movl %eax, %ebx

start_loop:
cmpl $0, $eax
```

A simple world

Example

```
je loop_exit
incl %edi
movl data_items(,$edi, 4), %eax
cmpl %ebx, %eax
jle start_loop
movl %eax, %ebx
jmp start_loop
loop_exit:
movl $1, %eax
int $0x80
```

A simple world

Example

```
je loop_exit
incl %edi
movl data_items(,$edi, 4), %eax
cmpl %ebx, %eax
jle start_loop
movl %eax, %ebx
jmp start_loop
loop_exit:
movl $1, %eax
int $0x80
```

1. Why cmpl $0, $eax?

A simple world

Example

```
je loop_exit
incl %edi
movl data_items(,$edi, 4), %eax
cmpl %ebx, %eax
jle start_loop
movl %eax, %ebx
jmp start_loop
loop_exit:
movl $1, %eax
int $0x80
```

1. Why cmpl $0, $eax? the last item of data_items

A simple world

Example

```
je loop_exit
incl %edi
movl data_items(,$edi, 4), %eax
cmpl %ebx, %eax
jle start_loop
movl %eax, %ebx
jmp start_loop
loop_exit:
movl $1, %eax
int $0x80
```

1. Why cmpl $0, $eax? the last item of data_items
2. How to check result?

A simple world

Example

```
je loop_exit
incl %edi
movl data_items(,$edi, 4), %eax
cmpl %ebx, %eax
jle start_loop
movl %eax, %ebx
jmp start_loop
loop_exit:
movl $1, %eax
int $0x80
```

1. Why cmpl $0, $eax? the last item of data_items
2. How to check result? echo $?

A simple world

Example

```
je loop_exit
incl %edi
movl data_items(,$edi, 4), %eax
cmpl %ebx, %eax
jle start_loop
movl %eax, %ebx
jmp start_loop
loop_exit:
movl $1, %eax
int $0x80
```

1. Why cmpl $0, $eax? the last item of data_items
2. How to check result? echo $?
3. How to pass the result to next function?

A simple world

Example

```
je loop_exit
incl %edi
movl data_items(,$edi, 4), %eax
cmpl %ebx, %eax
jle start_loop
movl %eax, %ebx
jmp start_loop
loop_exit:
movl $1, %eax
int $0x80
```

1. Why cmpl $0, $eax? the last item of data_items
2. How to check result? echo $?
3. How to pass the result to next function? Euh...next section

A simple world

Some caveat

1. SP is a freak!
2. You only have a limited number of registers.
3. Frame, frame, it's frame...
4. Alignment encounter with SP.
5. ......

Simple but frustrated...

Agreements(ABI)

Why Application Binary Interface(ABI)?

Definition

In computer software, an application binary interface (ABI) is an interface between two binary program modules

ABIs cover details such as:

1. a processor instruction set (with details like register file structure, stack organization, memory access types, ...)
2. the sizes, layouts, and alignments of basic data types that the processor can directly access
3. the calling convention
4. syscall and more

Something like TCP/IP?

Agreements(ABI)

Our interest, the calling convention

Definition

In computer science, a calling convention is an implementation-level (low-level) scheme for how subroutines receive parameters from their caller and how they return a result.

So, for X86-64, what the convention?

Operand size (bits)	Argument number 1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Notes

1, rax is the result

2, arguments on stack if more than six

Agreements(ABI)

Some examples...

(a) C code

```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

(a) C code for calling function

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

Imagine what the assembly code?

Agreements(ABI)

(b) Generated assembly code

```
void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
Arguments passed as follows:
  a1  in %rdi      (64 bits)
  a1p in %rsi      (64 bits)
  a2  in %edx      (32 bits)
  a2p in %rcx      (64 bits)
  a3  in %r8w      (16 bits)
  a3p in %r9       (64 bits)
  a4  at %rsp+8    ( 8 bits)
  a4p at %rsp+16   (64 bits)
proc:
  movq   16(%rsp), %rax    Fetch a4p     (64 bits)
  addq   %rdi, (%rsi)      *a1p += a1    (64 bits)
  addl   %edx, (%rcx)      *a2p += a2    (32 bits)
  addw   %r8w, (%r9)       *a3p += a3    (16 bits)
  movl   8(%rsp), %edx     Fetch a4      ( 8 bits)
  addb   %dl, (%rax)       *a4p += a4    ( 8 bits)
  ret                      Return
```

(b) Generated assembly code

```
long call_proc()
call_proc:
  Set up arguments to proc
  subq    $32, %rsp            Allocate 32-byte stack frame
  movq    $1, 24(%rsp)         Store 1 in &x1
  movl    $2, 20(%rsp)         Store 2 in &x2
  movw    $3, 18(%rsp)         Store 3 in &x3
  movb    $4, 17(%rsp)         Store 4 in &x4
  leaq    17(%rsp), %rax       Create &x4
  movq    %rax, 8(%rsp)        Store &x4 as argument 8
  movl    $4, (%rsp)           Store 4 as argument 7
  leaq    18(%rsp), %r9        Pass &x3 as argument 6
  movl    $3, %r8d             Pass 3 as argument 5
  leaq    20(%rsp), %rcx       Pass &x2 as argument 4
  movl    $2, %edx             Pass 2 as argument 3
  leaq    24(%rsp), %rsi       Pass &x1 as argument 2
  movl    $1, %edi             Pass 1 as argument 1
  Call proc
  call    proc
  Retrieve changes to memory
  movslq  20(%rsp), %rdx       Get x2 and convert to long
  addq    24(%rsp), %rdx       Compute x1+x2
  movswl  18(%rsp), %eax       Get x3 and convert to int
  movsbl  17(%rsp), %ecx       Get x4 and convert to int
  subl    %ecx, %eax           Compute x3-x4
  cltq                         Convert to long
  imulq   %rdx, %rax           Compute (x1+x2) * (x3-x4)
  addq    $32, %rsp            Deallocate stack frame
  ret                          Return
```

How many architectures

So, each architecture corresponds to an ABI. How many architectures theres is?

Well, Of course, I don't know ;-)

But for common architecture:

1. arm32
   1. arguments: r0-r3
   2. results: the same as arguments
2. X86
   1. arguments: it depends. On the stack?
   2. results: eax
3. SW64(Yes, your old friend!)
   1. arguments: r16-r21
   2. results: r0 & r1

Inline Assembly

When you travel happily in the kernel source...

Damn!

```
__asm__ ("swp %0, %0, [%1]" : : "r"(val), "r"(addr));
```

This is inline assembly, a syntax that allows you to write assembly in C(or other high level language?)

Example

```
int a=10, b;
asm ("movl %1, %%eax;
movl %%eax, %0;"
:"=r"(b) /* output */
:"r"(a) /* input */
:"%eax" /* clobbered register */
);
```

Reverse Engineering

Some friends you need...

1. man objdump
2. man readelf
3. man gdb
4. man gcc
5. ELF(a really big topic)
6. last but not least, Google

Reverse Engineering

Try the real game!

```
a at %ebp+8, b at %ebp+12, c at %ebp+16
    movl    8(%ebp), %eax
    cmpl    $7, %eax
    ja      .L2
    jmp     *.L7(,%eax,4)
  .L2:
    movl    12(%ebp), %eax
    jmp     .L8
  .L5:
    movl    $4, %eax
    jmp     .L8
  .L6:
    movl    12(%ebp), %eax
    xorl    $15, %eax
    movl    %eax, 16(%ebp)
  .L3:
    movl    16(%ebp), %eax
    addl    $112, %eax
    jmp     .L8
  .L4:
    movl    16(%ebp), %eax
    addl    12(%ebp), %eax
    sall    $2, %eax
  .L8:
```

```
.L7:
   .long   .L3
   .long   .L2
   .long   .L4
   .long   .L2
   .long   .L5
   .long   .L6
   .long   .L2
   .long   .L4
```

```
int switcher(int a, int b, int c)
{
     int answer;
     switch(a) {
     case __________:           /* Case A */
          c = __________;
          /* Fall through */
     case __________:           /* Case B */
          answer = __________;
          break;
     case __________:           /* Case C */
     case __________:           /* Case D */
          answer = __________;
          break;
     case __________:           /* Case E */
          answer = __________;
          break;
     default:
          answer = __________;
     }
     return answer;
}
```

Questions

ANY
QUESTIONS?

References I

- Randal E. Bryant, David R. O'Hallaron
  Computer Systems, A prorgammer's perspective.
  Pearson, 2018

- Jonathan Bartlett
  Programming from the Ground Up.

- John R. Levine
  Linkers & Loaders.

- Oracle
  x86 Assembly Language Reference Manual.
  2010

- Sandeep.S
  GCC-Inline-Assembly-HOWTO, 2003.
  https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

References II

Wikipedia
Application binary interface, 2020.
https://en.wikipedia.org/wiki/Application_binary_interface

Wikipedia
Calling convention, 2020.
https://en.wikipedia.org/wiki/Calling_convention

Thanks

Thank you ;)!

pqy7172@gmail.com