Introduction to x86-64 (dis)assembly

aka. AMD64/x64

Toby Fleming

September 2016

Note

Talk only applies to executables that target SystemV AMD64 ABI. This means:

- Unix (AIX, Solaris), Linux, FreeBSD/macOS
- **not** Windows! (different ABI)
- OS running on 64-bit, x86-compatible processors
  (most modern Intel/AMD processors)
- ARM 64-bit (AArch64) ABI is also different!
  (e.g. Raspberry Pi, iOS)

Also very much for beginners.

Registers

Intel 8086

16-bit, 1978

registers
AX (primary accumulator)
BX (base)
CX (counter)
DX (other functions, cba with backcronyms)
Index registers
SI (Source Index)
DI (Destination Index)
BP (Base Pointer)
SP (Stack Pointer)

Intel 8086

16-bit, 1978

registers	Hi	Lo
AX (primary accumulator)	AH	AL
BX (base)	BH	BL
CX (counter)	CH	CL
DX (other functions, cba with backcronyms)	DH	DL
Index registers	16-bit	only
SI (Source Index)	-	-
DI (Destination Index)	-	-
BP (Base Pointer)	-	-
SP (Stack Pointer)	-	-

Intel 8086

16-bit, 1978

registers	Hi	Lo
AX (primary accumulator)	AH	AL
BX (base)	BH	BL
CX (counter)	CH	CL
DX (other functions, cba with backcronyms)	DH	DL
Index registers	16-bit	only
SI (Source Index)	-	-
DI (Destination Index)	-	-
BP (Base Pointer)	-	-
SP (Stack Pointer)	-	-

And some more, not so important for now

Intel 80386

32-bit, 1986

32-bit	16-bit	8-bit
EAX	AX	AL
EBX	BX	BL
ECX	CX	CL
EDX	DX	DL
ESI	SI	-
EDI	DI	-
EBP	BP	-
ESP	SP	-

AMD Athlon 64

64-bit, 2003

64-bit	32-bit	16-bit	8-bit	aka.
RAX	EAX	AX	AL	R0
RBX	EBX	BX	BL	R1
RCX	ECX	CX	CL	R2
RDX	EDX	DX	DL	R3
RSI	ESI	SI	SIL	R4
RDI	EDI	DI	DIL	R5
RSP	EBP	BP	BPL	R6
RBP	ESP	SP	SPL	R7
(**R**egister)	(**E**xtended)		(**L**ow byte)	

AMD Athlon 64

64-bit, 2003

New general purpose registers R8–15!

64-bit	32-bit	16-bit	8-bit
R8	R8D	R8W	R8B
…			
R15	R15D	R15W	R15B
(quadword)	(doubleword)*	(word)	(byte)

* Warning: aka. long word. This can be used in 32-bit asm instruction names (e.g. `movl`), and is why the new 8-bit registers don't use the **L** postfix.

AMD Athlon 64

64-bit, 2003

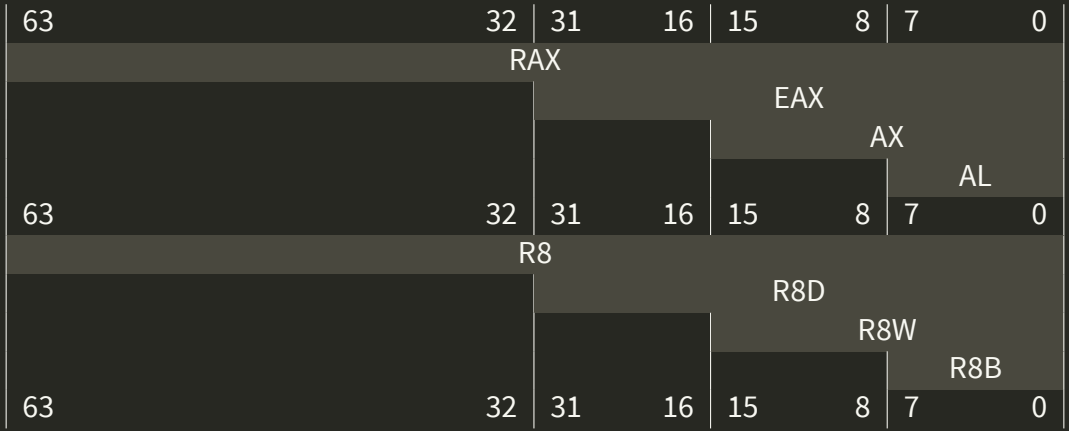
New general purpose registers R8–15!

64-bit	32-bit	16-bit	8-bit
R8	R8D	R8W	R8B
…			
R15	R15D	R15W	R15B
(quadword)	(doubleword)*	(word)	(byte)

* Warning: aka. long word. This can be used in 32-bit asm instruction names (e.g. `movl`), and is why the new 8-bit registers don't use the **L** postfix.

Pop quiz: what's half a byte?

Register overlap



Calling conventions

Calling conventions

- Part of the ABI (Application Binary Interface, c.f. API)
- Don’t bother with x86 32-bit calling conventions: too many, inconsistent, old
- ABI on most servers/desktops/laptops: SystemV AMD64
- ...except Windows, ARM, very old 32-bit kit

System V AMD64 ABI calling convention

- First six integer args: RDI, RSI, RDX, RCX, R8, R9
- Any more? Use stack
- Floating point has other registers
- Caller owns RBX, RBP, R12-R15
- Called function must restore these if overwritten (use stack)
- Return value in RAX (and RDX)

Tedious, see handout!

Stack

What's the call stack?

Warning, simplification!

- last in, first out queue (push/pop)
- way to "allocate memory"
- stack is limited in size and protected
- use stack for local vars/short lived
- use heap for large vars/long lived
- stack grows downwards
- heap grows upwards
- ignore heap, managed by runtime (`malloc`)

What's the call stack?

Warning, simplification!

- last in, first out queue (push/pop)
- way to "allocate memory"
- stack is limited in size and protected
- use stack for local vars/short lived
- use heap for large vars/long lived
- stack grows downwards
- heap grows upwards
- ignore heap, managed by runtime (`malloc`)

e.g. 16 GiB

0 GiB

What’s the call stack?

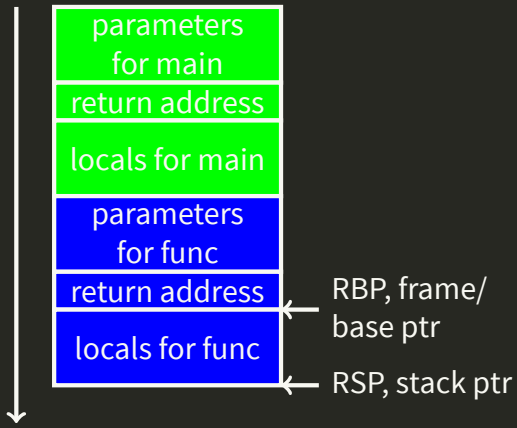
Warning, simplification!

- stack remembers program flow
- each function has a stack frame
- stack overflow = bad
- can overwrite return address
- boom, changed program execution

What's the call stack?

Warning, simplification!

- stack remembers program flow
- each function has a stack frame
- stack overflow = bad
- can overwrite return address
- boom, changed program execution



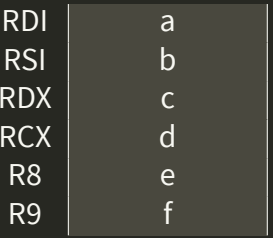
Call stack example

```
uint64_t func(
    uint64_t a, uint64_t b, uint64_t c, uint64_t d,
    uint64_t e, uint64_t f, uint64_t g, uint64_t h)
{
    uint64_t x = a + b + c + d;
    uint64_t y = e + f + g + h;
    uint64_t z = x + y;
    return z;
}
```

examples/stack.c

Call stack example

Just called `func(...)`



Manipulating the stack

```
push rax
; equivalent
sub rsp, 8
mov [rsp], rax
```

```
pop rax
; equivalent
mov rax, [rsp]
add rsp, 8
```

```
call fn
; equivalent
push rip
jmp fn
```

```
ret
; equivalent
pop rip
```

These instructions exist for a reason, try not to mess with `rsp` and `rip` manually.

Practical time!

Howto disassemble

- `clang -O1 -S -masm=intel foo.c -o foo.s`
  (recommended if you have the source)
- `gcc` also works (same flags), worse assembly output IMO
- `gdb`
- `objdump -M intel -S foo > foo.s`
- macOS users add `-target x86_64-pc-linux-elf` to cross compile and follow along

Either way, will probably need some clean-up. So I've added them to a git repo:
`https://github.com/tobywf/talk-x86-64-asm`

Minimal program

```c
int main(void)
{
    return 0;
}
```

examples/main.c

Minimal program

```
main: ; -O0
    push    rbp                     ; save old stack frame
    mov rbp, rsp                    ; make new stack frame (rbp
    xor eax, eax                    ; rax = 0 (return val)
    mov dword ptr [rbp - 4], 0      ; set local var to 0
    ; n.b. explicit operand length
    ; cannot be inferred from value (0)
    pop rbp                         ; restore old stack frame
    ret

main: ; -O1
    xor eax, eax                    ; rax = 0 (return val)
    ret
```

examples/main.asm

Helloworld

```c
#include <stdio.h>

int main(void)
{
    puts("Hello World");
    return 0;
}
```

examples/puts.c

Helloworld

```
main: ; -O1
    push    rax      ; save rax
    mov edi, .L.str  ; load parameter (ptr)
    call    puts     ; call puts
    xor eax, eax     ; rax = 0 (return value)
    pop rdx          ; pop stack
    ret

.L.str:
    .asciz  "Hello World"
    .size   .L.str, 12
```

examples/puts.asm

Helloworld

```
main: ; -O0
    push    rbp                     ; save old stack frame
    mov rbp, rsp                    ; make new stack frame
    ; (rbp points to top of stack)
    sub rsp, 16                     ; "allocate" more stack sp
    ; (has to be 16 byte aligned)
    movabs  rdi, .L.str             ; load pointer to string
    mov dword ptr [rbp - 4], 0      ; save 0 to stack (why?)
    call    puts
    xor ecx, ecx                    ; rcx = 0 (why?)
    mov dword ptr [rbp - 8], eax    ; save eax to stack (why?)
    mov eax, ecx                    ; set eax = ecx = 0 (why?)
    add rsp, 16                     ; "deallocate" stack space
    pop rbp                         ; restore old stack frame
    ret
```

Let’s have some fun

Why not printf?

```c
#include <stdio.h>

int main(void)
{
    const char *string = NULL;
    printf("string: %s\n", string); // doesn't segfault
    printf("string: ");
    printf("%s\n", string); // segfault on -O1 and higher
    return 0;
}
```

examples/print.c

Why? Have a look at the `-O0` and `-O1` disassembly!

Why not printf?

```
main: ; -O1
    push    rax
    mov edi, .L.str
    xor esi, esi
    xor eax, eax
    call    printf
    mov edi, .L.str.1
    xor eax, eax
    call    printf
    xor edi, edi
    call    puts  ; <-- oops
    xor eax, eax
    pop rcx
    ret
```

Why not printf?

- Actual `gcc` bug: “too agressive [sic] printf optimization”
- `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=25609`
- Bug status?

Why not printf?

- Actual `gcc` bug: “too agressive [sic] printf optimization”
- `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=25609`
- Bug status? Won’t fix/Invalid

Final example

- `examples/stack.c` and `examples/stack.asm`

Final example

- `examples/stack.c` and `examples/stack.asm`
- Hope this proves how good `clang` & LLVM is
- -O0 isn't without trade-offs, even for debugging!
- DWARF makes debugging at higher optimisations okay(ish)

Fin

Questions?