The x86-64 processor (aka amd64, x64): Whirlwind tour

devblogs.microsoft.com/oldnewthing/20220831-00

August 31, 2022

Raymond Chen

I figure I'd tidy up the processor overview series by covering the last[1] processor on my list of "processors Windows has supported in its history," namely, the x86-64. Other names for this architecture are amd64 (because AMD invented it) and x64 (which is super-confusing because it doesn't correspond with x86, a common nickname for the x86-32).

This is going to be a quick overview because the x86-64 is a natural extension of the i386, which we covered some time ago. I'll just highlight the differences.

Each existing 32-bit general-purpose register has been extended from 32 bits to 64. The name of the 64-bit register is based on the name of the 32-bit register, but with the leading *e* changed to a leading *r*. Eight new 64-bit registers were introduced, bring the total to 16. Instead of giving quirky names to the new registers, they are just numbered: *r8* through *r15*. To match the existing classic registers, the new registers also have aliases for referring to partial registers, and partial register aliases were invented for some of the classic registers that lacked them.

Register	Aliases				Preserved?	Notes
	Bits 31:0	Bits 15:0	Bits 15:8	Bits 7:0		
rax	*eax*	*ax*	*ah*	*al*	No	Return value
rbx	*ebx*	*bx*	*bh*	*bl*	Yes	
rcx	*ecx*	*cx*	*ch*	*cl*	No	Parameter 1
rdx	*edx*	*dx*	*dh*	*dl*	No	Parameter 2
rsi	*esi*	*si*		*sil*	Yes	
rdi	*edi*	*di*		*dil*	Yes	
rsp	*esp*	*sp*		*spl*	Yes	Stack pointer

rbp	*ebp*	*bp*		*bpl*	Yes	Frame pointer
r8	*r8d*	*r8w*		*r8b*	No	Parameter 3
r9	*r9d*	*r9w*		*r9b*	No	Parameter 4
r10	*r10d*	*r10w*		*r10b*	No	
r11	*r11d*	*r11w*		*r11b*	No	
r12	*r12d*	*r12w*		*r12b*	Yes	
r13	*r13d*	*r13w*		*r13b*	Yes	
r14	*r14d*	*r14w*		*r14b*	Yes	
r15	*r15d*	*r15w*		*r15b*	Yes	

The *eip* and *eflags* registers are correspondingly expanded to 64-bit registers *rip* and *rflags*.

Additional restrictions have been imposed on the use of the *ah*, *bh*, *ch*, and *dh* registers. The details aren't important for reading code, so I won't bother digging into them.

Windows requires that the stack be 16-byte aligned at function call boundaries, and there is no red zone. Calling a function pushes the 8-byte return address onto the stack, so on entry to a function, the stack is misaligned. Functions typically realign the stack in their prologue.

The old 8087-based floating point registers are not used.[2] Instead, the SIMD XMM registers are used for floating point calculations. These registers are 128 bits wide and can be viewed as four single-precision floating point values or as two double-precision floating point values. When used to pass parameters or return floating point values, only the bottom lane is used.

Eight more XMM registers have been added, bringing the total to 16.

Register	**Preserved?**	**Notes**
XMM0	No	Parameter 1 and return value
XMM1	No	Parameter 2 and second return value
XMM2	No	Parameter 3
XMM3	No	Parameter 4
XMM4	No	
XMM5	No	

XMM6	Yes	
XMM7	Yes	
XMM8	Yes	
XMM9	Yes	
XMM10	Yes	
XMM11	Yes	
XMM12	Yes	
XMM13	Yes	
XMM14	Yes	
XMM15	Yes	

Calling convention

The calling convention is register-based for the first four parameters, with remaining parameters on the stack. In practice, the stack-based parameters are not `push`'d, but rather the values are `mov`'d into the preallocated stack space.

For register-based parameters, integer parameters go into the general-purpose registers and floating point parameters go into the floating point registers. When a register is used to hold a parameter, its counterpart register goes unused. For example, a function that takes an integer and a double will pass the integer in *rcx* and the double in *xmm1*.

There are always 4 × 8 = 32 bytes of home space for the register-based parameters, even if the function has fewer than four formal parameters. (If this bothers you, then you can reinterpret the home space as a 32-byte red zone that resides *above* the return address.)

Integer return values up to 64 bits go into *rax* If the return value is a 128-bit value, then the *rdx* register holds the upper 64 bits. Floating point return values are returned in *xmm0*.

The caller is responsible for cleaning the stack. In practice, the caller does not clean the stack after every call, but rather preallocates the stack space in the prologue, reuses the stack space for multiple calls, and then cleans it all up in the epilogue.

Exception handling is done by unwind tables, not by threading exception handlers through the stack at runtime.

Partial registers

When a 32-bit partial register is the destination of an operation, the upper 32 bits are set to zero. For example, consider

```
add     eax, ecx
```

On the 32-bit 80386, this adds the value of *ecx* to *eax* and puts the result back into *eax*. On x86-64, this performs the same calculation, but since the destination is the 32-bit partial register *eax*, the operation also zeroes out the upper 32 bits of *rax*.

Another way of looking at this is that writes to 32-bit partial registers are *zero-extended* to 64-bit values.[3]

Note, however, that operations on 16-bit and 8-bit partial registers leave the unused bits unchanged.

Addressing modes

The 32-bit addressing modes carry over to 64-bit, with these exceptions:

- Absolute addressing mode is limited to signed 32-bit addresses.
- There is a new *rip*-relative addressing mode.

The offsets in the memory addressing modes are 32-bit signed values, for a reach of ±2GB.

The *rip*-relative addressing mode greatly reduces the number of fixups required to relocate a module. The enormous ±2GB reach means that any reasonably-sized module can use it to access all of its static data, be it a read-only table embedded in the code segment or read-write data in the data segment.

The disassembler automatically performs the necessary calculations to convert the *rip*-relative address to an absolute one at disassembly time, so you are unlikely even to realize that anything has changed.

Immediates

In general, immediates are capped at 32 bits. The exception is that you can use a 64-bit immediate in the `mov reg, imm64` instruction.

Segments

Segmentation is architecturally dead. The processor is always in flat mode. The *fs* and *gs* selectors have been repurposed as two additional registers that add an operating-system-defined value to the effective address.

```
mov   rax, qword ptr gs:[rcx*8+1480h]
```

The *base address* assigned to the *gs* register is added to the effective address *rcx * 8 + 0x1480*, producing a final address that is the target of the memory operation.

Windows sets the *gs* register's base address to a block of per-thread data. During context switches, the base address of the *gs* register is updated to point to the per-thread data of the incoming thread. The *fs* register has not yet been assigned a meaning and should not be used.[4] The Windows ABI forbids modifying either of these segment registers.

Instruction set changes

Some rarely-used instructions have been removed, primarily the binary-coded decimal instructions, `BOUND`, and `PUSHAD` / `POPAD` instructions.

New instructions for dealing with 64-bit registers:

```
; sign-extend 32-bit to 64-bit
movsxd  r64, r32/m32
```

There is no need for a zero-extend instruction because operations on 32-bit registers automatically zero-extend to 64-bit values, so if the value was the result of a calculation, you probably got the zero-extended value anyeway. If you want to wipe out the top 32 bits of an existing 64-bit value, you could do

```
; zero-extend 32-bit to 64-bit
mov     r32, r32
```

This can result in some odd-looking instructions like

```
mov     eax, eax        ; zero-extend eax to rax
```

On its face, the instruction looks pointless, but we're performing for the zero-extending side effect.[5]

There are also specialized instruction for certain sign-extending scenarios:

```
cwqe                    ; sign-extend eax to rax
cqd                     ; sign-extend rax to rdx:rax
```

Lightweight leaf functions and exception handling

A lightweight leaf function is one which can perform all of its work using only non-preserved registers, the inbound parameter home space, and stack space occupied by stack-based inbound parameters (if any). Preserved registers and the stack pointer must remain unchanged for the entire lifetime of the function, and the return address must remain at the top of the stack.

The inability to move the stack pointer means that the stack pointer is *not* at a multiple of 16 for the lifetime of a lightweight leaf function.

The x86-64 ABI abandons the stack-based exception handling model of its 32-bit older brother and joins the RISC crowd by using table-based exception handling. With the exception of lightweight leaf functions, all functions must declare unwind codes that allow the exception unwinder to restore registers from the stack and find the return address. Any function that does not have unwind codes is assumed to be a lightweight leaf function.

Annotated disassembly

I'll defer to the existing documentation (which I wrote).

Encoding notes

Instructions that operate on the classic 32-bit or 8-bit registers tend to have the most compact encodings. Using any of the new registers (*r8* through *r15*, or *xmm8* through *xmm15*, or the new aliases *sil*, *dil*, *spl* or *bpl*) typically requires a one-byte prefix. An instruction that operates on word-sized data typically incurs an additional byte encoding. And fancy addressing modes (involving scaling or multiple registers contributing to the effective address) also require yet another byte for the encoding.

I'm not sure how aggressively the compiler allocates registers and chooses instructions which have compact encodings. It certainly didn't stand out to me.

Bonus reading: x64 software conventions.

Bonus chatter: Now that I've exhausted my list of processors that Windows has supported over the years, I'll have to start branching out into other processors. I'm open to suggestions. Though I probably won't be as detailed as these processor overviews have been, since the original goal of these overviews was to give you enough information to get started debugging on Windows. For other processors, I'll probably just focus on the one or two things that make them interesting, like SPARC register windows, or 68000's separate data and address registers.

[1] Early versions of Windows CE allegedly supported the StrongARM and possibly even M32R and other architectures, but I can't find any binaries for those versions, so I have nothing to investigate.

[2] They are still physically present and usable, but in practice, nobody uses them,[6] and they are not part of the calling convention.

[3] I strongly suspect this design decision was made to avoid introduce spurious register dependencies due to partial register operations.

[4] On x86-32, the *fs* register is used to access the per-thread data. Why did Windows switch to using *gs* on x86-64? One theory is that there is a special instruction on x86-64 called `SWAPGS` that lets the kernel exchange the *gs* base address with another internal register. This instruction is used on transitions to and from user mode, so the kernel can quickly switch from user-mode thread data to kernel-mode thread data on entry and to switch it back on exit. No such courtesy instruction exists for the *fs* register. Another theory is that `fs` is reserved for the 32-bit emulation layer.

[5] It also means that the x86-32 pun of interpreting `nop` as `xchg eax, eax` does not work in x86-64. The self-exchange zeroes out the high 32 bits as a side effect. The Windows debugger doesn't realize this, and if you ask it to assemble `xchg eax, eax`, it encodes it as `90`, using the one-byte encoding of `xchg eax, r32`, unaware that this doesn't work if the other register is *also eax*. The correct encoding of `xchg eax, eax` is `87 c0`, using the larger two-byte encoding.

[6] Apparently, gcc and clang do use them for the 80-bit floating point `long double` type.

Raymond Chen

Follow