How many registers does an x86-64 CPU have?

Nov 30, 2020 **Tags: programming, x86**

x86 is back in the general programmer discourse, in part thanks to Apple's M1 and Rosetta 2. As such, I figured I'd do *yet another* x86-64 post.

Just like the last one, I'm going to cover a facet of the x86-64 ISA that sets it apart as unusually complex among modern ISAs: the number and diversity of registers available.

Like instruction counting, register counting on x86-64 is subject to debates over methodology. In particular, for this blog post, I'm going to lay the following ground rules:

- I **will** count sub-registers (e.g., `EAX` for `RAX`) as distinct registers. My justification: they have different instruction encodings, and both Intel and AMD optimize/pessimize particular sub-register use patterns in their microcode.
- I **will** count registers that are present on x86-64 CPUs, but that can't be used in long mode.
- I **won't** count registers that are *only* present on older x86 CPUs, like the 80386 and 80486 test registers.
- I **won't** count microarchitectural implementation details, like shadow registers.
- I **will** count registers that aren't directly addressable, like MSRs that can only be accessed through `RDMSR`. However, I **won't** (or will try not to) double-count registers that have multiple access mechanisms (like `RDMSR` and `RDTSC`).
- I **won't** count model-specific registers that fall into these categories:
  - MSRs that are only present on niche x86 vendors (Cyrix, Via)
  - MSRs that aren't widely available on recent-ish x86-64 CPUs
    - **Errata**: I accidentally included AVX-512 in some of the original counts below, not realizing that it hadn't been released on any AMD CPUs. The post has been updated.

    - MSRs that are *completely* undocumented (both officially and unofficially)

In addition to the rules above, I'm going to use the following considerations and methodology for grouping registers together:

- Many sources, both official and unofficial, use "model-specific register" as an umbrella term for any non-core or non-feature-set register supplied by an x86-64 CPU. Whenever possible, I'll try to avoid this in favor of more specific categories.
- Both Intel and AMD provide synonyms for registers (e.g. `CR8` as the "task priority register," or `TPR`). Whenever possible, I'll try to use the more generic/category conforming name (like `CR8` in the case above).
- In general, the individual cores of a multicore processor have independent register states. Whenever this **isn't** the case, I'll make an effort to document it.

General-purpose registers

The general-purpose registers (or GPRs) are **the** primary registers in the x86-64 register model. As their name implies, they are the **only** registers that are *general purpose*: each has a set of conventional uses[1], but programmers are generally free to ignore those conventions and use them as they please[2].

Because x86-64 evolved from a 32-bit ISA which in turn evolved from a 16-bit ISA, each GPR has a set of *subregisters* that hold the lower 8, 16 and 32 bits of the full 64-bit register.

As a table:

64-bit	32-bit	16-bit	8-bit (low)
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B

R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Some of the 16-bit subregisters are also special: the original 8086 allowed the **high** byte of `AX`, `BX`, `CX`, and `DX` to be accessed indepenently, so x86-64 preserves this for some encodings:

16-bit	8-bit (high)
AX	AH
BX	BH
CX	CH
DX	DH

So that's 16 full-width GPRs, fanning out to another 52 subregisters.

Registers in this group: 68.

Running total: 68.

Special registers

This is sort of an artificial category: like every ISA, x86-64 has a few "special" registers that keep things moving along. In particular:

- The *instruction pointer*, or `RIP`.

  x86-64 has 32- and 16-bit variants of `RIP` (`EIP` and `IP`), but I'm **not** going to count them as separate registers: they have identical encodings and can't be used in the same CPU mode[3].

- The *status register*, or `RFLAGS`.

  Just like `RIP`, `RFLAGS` has 32- and 16-bit counterparts (`EFLAGS` and `FLAGS`). Unlike `RIP`, these counterparts can be partially mixed: `PUSHF` and `PUSHFQ` are both valid in long mode, and `LAHF`/`SAHF` can operate on the bits of `FLAGS` on some x86-64 CPUs outside of compatiblility mode[4]. So I'm going to go ahead and count them.

Registers in this group: 4.

Running total: 72.

Segment registers

x86-64 has a total of 6 segment registers: `CS`, `SS`, `DS`, `ES`, `FS`, and `GS`. The operation varies with the CPU's mode:

- In all modes except for long mode, each segment register holds a *selector*, which indexes into either the GDT or LDT. That yields a segment *descriptor* which, among other things, supplies the base address and extent of the segment.
- In long mode all but `FS` and `GS` are treated as having a base address of zero and a 64-bit extent, effectively producing a flat address space. `FS` and `GS` are retained as special cases, but no longer use the segment descriptor tables: instead, they access base addresses that are stored in the `FSBASE` and `GSBASE` model-specific registers[5]. More on those later.

Registers in this group: 6.

Running total: 78.

SIMD and FP registers

The x86 family has gone through *several* generations of SIMD and floating-point instruction groups, each of which has introduced, extended, or re-contextualized various registers:

- x87
- MMX
- SSE (SSE2, SSE3, SSE4, SSE4, …)
- AVX (AVX2, AVX512)

Let's do them in rough order.

x87

Originally a discrete coprocessor with its own instruction set and register file, the x87 instructions have been regularly baked into x86 cores themselves since the 80486.

Because of its coprocessor history, x87 defines both normal registers[6] (akin to GPRs) and a variety of special registers needed to control the FPU state:

- `ST0` through `ST7`: 8 80-bit floating-point registers
- `FPSW`, `FPCW`, `FPTW` [7]: Control, status, and tag-word registers

- “Data operand pointer”: I don’t know what this one does, but the Intel SDM specifies it[8]
- Instruction pointer: the x87 state machine apparently holds its own copy of the current x87 instruction
- Last instruction opcode: this is apparently distinct from the x87 opcode, and has its own register

Registers in this group: 14.

Running total: 92.

MMX

MMX was Intel’s first attempt at consumer SIMD in their x86 chips, released back in 1997.

For design reasons that are a complete mystery to me, the MMX registers are actually sub-registers of the x87 `STn` registers: each 64-bit `MMn` occupies the mantissa component of its corresponding `STn`. Consequently, x86 (and x86-64) CPUs cannot execute MMX and x87 instructions at the same time.

Edit: This section incorrectly included `MXCSR`, which was actually introduced with SSE. Thanks to /u/Skorezore for pointing out the error.

Registers in this group: 8.

Running total: 100.

SSE and AVX

For simplicity’s sake, I’m going to wrap SSE and AVX into a single section: they use the same sub-register pattern as the GPRs and x87/MMX do, so they fit well into a single table:

AVX-512 (512-bit)	AVX-2 (256-bit)	SSE (128-bit)
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8

ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

In other words: the lower half of each `ZMMn` is `YMMn`, and the lower half of each `YMMn` is `XMMn`. There's no direct way register access for just the upper half of `YMMn`, nor does `ZMMn` have direct 256- or 128-bit access for the thunks of its upper half.

SSE also defines a new status register, `MXCSR`, that contains flags roughly parallel to the arithmetic flags in `RFLAGS` (along with floating-point flags in the x87 status word). SSE also introduces a load/store instruction pair for manipulating it (`LDMXCSR` and `STMXCSR`).

AVX-512 **also** introduces eight *opmask* registers, `k0` through `k7`. `k0` is a special case that behaves much like the "zero" register on some RISC ISAs: it can't be stored to, and loads from it always produce a bitmask of all ones.

Errata: The table above includes AVX-512, which isn't available on any AMD CPUs as of 2020. I've updated the counts below to only include SSE and AVX2-introduced registers.

Registers in this group: 33.

Running total: 133.

Bounds registers

Intel added these with MPX, which was intended to offer hardware-accelerated bounds checking. Nobody uses it, since it doesn't work very well. But x86 is eternal and slow to fix mistakes, so we'll probably have these registers taking up space for at least a while longer:

- `BND0` — `BND3`: Individual 128-bit registers, each containing a pair of addresses for a bound.
- `BNDCFG`: Bound configuration, kernel mode.
- `BNDCFU`: Bound configuration, user mode.
- `BNDSTATUS`: Bound status, after a `#BR` is raised.

Registers in this group: 7.

Running total: 140.

Debug registers

These are what they sound like: registers that aid and accelerate software debuggers, like GDB.

There are 6 debug registers of two types:

- `DR0` through `DR3` contain linear addresses, each of which is associated with a breakpoint condition.
- `DR6` and `DR7` are the debug status and control registers. `DR6`'s lower bits indicate which debug conditions were encountered (upon entering the debug exception handler), while `DR7` controls which breakpoint addresses are enabled and their breakpoint conditions (e.g., when a particular address is written to).

What about `DR4` and `DR5`? For reasons that are unclear to me, they don't (and have never) existed[9]. They *do* have encodings but are treated as `DR6` and `DR7`, respective, or produce an `#UD` exception when `CR4.DE[bit 3] = 1`.

Registers in this group: 6.

Running total: 146.

Control registers

x86-64 defines a set of *control registers* that can be used to manage and inspect the state of the CPU.

There are 16 “main” control registers, all of which can be accessed with a `MOV` variant:

Name	Purpose
CR0	Basic CPU operation flags
CR1	Reserved
CR2	Page-fault linear address
CR3	Virtual addressing state
CR4	Protected mode operation flags
CR5	Reserved
CR6	Reserved
CR7	Reserved
CR8	Task priority register (TPR)
CR9	Reserved
CR10	Reserved
CR11	Reserved
CR12	Reserved
CR13	Reserved
CR14	Reserved
CR15	Reserved

All reserved control registers result in an `#UD` when accessed, which makes me inclined to not count them in this post.

In addition to the “main” `CRn` control registers there are also the “extended” control registers, introduced with the XSAVE feature set. As of writing, `XCR0` is the only specified extended control register.

The extended control registers use `XGETBV` and `XSETBV` instead of a `MOV` variant.

Registers in this group: 6.

Running total: 152.

“System table pointer registers”

That’s what the Intel SDM calls these[8]: these registers hold sizes and pointers to various protected mode tables.

As best I can tell, there are four of them:

- `GDTR`: Holds the size and base address of the GDT
- `LDTR`: Holds the size and base address of the LDT
- `IDTR`: Holds the size and base address of the IDT
- `TR`: Holds the TSS selector and base address for the TSS

The `GDTR`, `LDTR`, and `IDTR` each seem to be 80 bits in 64-bit modes: 16 lower bits for the size of the register's table, and then the upper 64 bits for the table's starting address.

`TR` is likewise 80 bits: 16 bits for the selector (which behaves identically to a segment selector), and then another 64 for the base address of the TSS[10].

Registers in this group: 4.

Running count: 156.

Memory-type-ranger registers

These are an interesting case: unlike all of the other registers I've covered so far, these are **not** unique to a particular CPU in a multicore chip; instead, they're shared across all cores[11].

The number of MTTRs seems to vary by CPU model, and have been largely superseded by entries in the page attribute table, which is programmed with an MSR[12].

Registers in this group:

Running count: >156.

Model specific registers

Model-specific registers are where things get fun.

Like extended control registers, they're accessed indirectly (by identifier) through a pair of instructions: `RDMSR` and `WRMSR`. MSRs themselves are 64-bits but originated during the 32-bit era, so `RDMSR` and `WRMSR` read from and write to *two* 32-bit registers: `EDX` and `EAX`.

By way of example: here's the setup and `RDMSR` invocation for accessing the `IA32_MTRRCAP` MSR, which includes (among other things) that actual number of MTRRs available on the system:

```
1 MOV ECX, 0xFE ; 0xFE = IA32_MTRRCAP
2 RDMSR
3 ; The bits of IA32_MTRRCAP are now in EDX:EAX
```

`RDMSR` and `WRMSR` are privileged instructions, so normal ring-3 code can't access MSRs directly[13]. The one (?) exception that I know of is the timestamp counter (`TSC`), which is stored in the `IA32_TSC` MSR but can be read from non-privileged contexts with `RDTSC` and `RDTSCP`.

Two other interesting (but still privileged[14]) cases are `FSBASE` and `GSBASE`, which are

stored as `IA32_FS_BASE` and `IA32_GS_BASE`, respectively. As mentioned in the segment register section, these store the `FS` and `GS` segment bases on x86-64 CPUs. This makes them targets of relatively frequent use (by MSR standards), so they have their own dedicated R/W opcodes:

- `RDFSBASE` and `RDGSBASE` for reading
- `WRFSBASE` and `WRGSBASE` for writing

But back to the meat of things: how many MSRs *are* there?

Using the standards laid out at the beginning of this post, we're interested in counting what Intel calls "architectural" MSRs. From the SDM[15]:

> Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these "architectural MSRs" were given the prefix "IA32_".

According to the subsequent table[16], the highest architectural MSR is `6097/17D1H`, or `IA32_HW_FEEDBACK_CONFIG`. So, the naïve answer is over 6000.

However, there are significant gaps in the documented MSR ranges: Intel's documentation jumps directly from `3506/DB2H` (`IA32_THREAD_STALL`) to `6096/17D0H` (`IA32_HW_FEEDBACK_PTR`). On top of the empty ranges, there are also ranges that are explicitly marked as reserved, either generally or explicitly for later expansion of a particular MSR family.

To count the *actual* number of MSRs, I did a bit of pipeline ugliness:

- Extract just table 2-2 from Volume 4 of the SDM (link):

```
$ pdfjam 335592-sdm-vol-4.pdf 19-67 -o 2-2.pdf
```

- Use `pdftotext` to convert it to plain text and manually trim the next table from the last page:

```
$ pdftotext 2-2.pdf table.txt
# edit table.txt by hand
```

- Split the plain text table into a sequence of words, filter by `IA32_`, remove cruft, and do a standard sort-unique-count:

```
$ tr -s '[:space:]' '\n' < table.txt \
    | grep 'IA32_' \
    | tr -d '.' \
    | sed 's/\[.*$//' \
    | sort | uniq | wc -l
404
```

  (Output preserved for posterity here).

That pipeline left a bit of cruft towards the end thanks to quoted variants, so I count the actual number at 400 architectural MSRs. That's a lot more reasonable than 6096!

Registers in this group: 400

Running count: >556.

Other bits and wrapup

The footnotes at the bottom of this post cover most of my notes, but I also wanted to dump some other resources that I found useful while discovering registers:

- sandpile.org has a nice visualization of many of the architectural MSRs, including field breakdowns.
- Vol. 3A § 8.7.1 ("State of the Logical Processors") of the Intel SDM has a useful list of nearly all of the registers that are either unique to or shared between x86-64 cores.
- The OSDev Wiki has collection of helpful pages on various x86-64 registers, including a great page on the behavior of the segment base MSRs.

All told, I think that there are *roughly* **557** registers on the average (relatively recent) x86-64 CPU core. With that being said, I have some peripheral cases that I'm not sure about:

- Modern Intel CPUs use integrated APICs as part of their SMT implementation. These APICs have their own register banks which can be memory-mapped for reading and potential modification by an x86 core. I didn't count them because (1) they're memory mapped, and thus behave more like mapped registers from an arbitrary piece of hardware than CPU registers, and (2) I'm not sure whether AMD uses the same mechanism/implementation.

- The Intel SDM implies that Last Branch Records are stored in discrete, non-MSR registers. AMD's developer manual, on the other hand, specifies a range of MSRs. As such, I didn't attempt to count these separately.
- Both Intel and AMD have their own (and incompatible) virtualization extensions, as well as their own enclave/hardened execution extensions. My intuition is that each introduces some additional registers (or maybe just MSRs), but their vendor-specificity made me inclined to not look too deeply.

Information on these (and any other) registers would be deeply appreciated.

---

1. Both ISA and OS specified.
2. With a few exceptions: some x86 instructions have their register(s) baked into their encodings, preventing programmers from directly substituting another GPR. Examples: the stack operations (with `rsp`/`rbp`) and some of the rep-prefix operations (with `rcx`/`rsi`/`rdi`).
3. 64-bit kernels can run 32-bit userspace processes, but 64-bit and 32-bit code can't be mixed in the same process.
4. Specifically, `CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1`.
5. There's also a `KERNELGSBASE` MSR, which can be used with `SWAPGS` to quickly switch between user- and kernel-space `GS` base addresses.
6. "Normal" in the sense that they're for data processing, but they're actually in a weird stack structure for reasons that are lost to me.
7. My names; Intel doesn't abbreviate these.
8. Intel SDM Vol. 1 § 3.7.2: "Register Operands" [2]
9. Educated guess: there wasn't enough space in the original 32-bit control register for them, and the debug registers are niche enough for it to be not worth fixing.
10. Based on my reading of the SDM, but I'm less sure about this last part.
11. Intel SDM Vol. 3A § 8.7.1: "State of the Logical Processors" and § 8.7.3: "Memory Type Range Registers (MTRR)"
12. Specifically, `IA32_PAT`.
13. Linux provides `msr(4)`, which can be loaded to provide userspace R/W access to MSRs via devfs.
14. Unless support for `FSGSBASE` is enabled, in which case `FSBASE` and `GSBASE`

can be modified directly from ring 3. Linux enabled FSGSBASE in 5.9, which was released a bit over a month ago.

15. Intel SDM Vol. 4 § 2.1: “Architectural MSRs”
16. Intel SDM Vol. 4, Table 2-2: “IA-32 Architectural MSRs”

Discussions: Reddit

Previously Newer