Bundle of excerpts from AMD64 (x86-64) documentation

COMP 40

Fall 2010

This packet contains key excerpts from the voluminous documentation for the AMD64 (`x86_64`) architecture used in every 64-bit Intel and AMD CPU. The excerpts include

- Names of the integer registers and their legacy internal fields (sub-registers)
- Summary of conventions of how integer registers, SSE registers (`%xmm`$*$), and floating-point registers are used within the procedure calling convention.
- Details about how the stack frame is laid out and how parameters are passed in the calling convention.
- Sizes and alignments of the standard scalar types
- Information about conditional-branch and conditional-move instructions
- Information about which registers may be modified by instructions *without* mentioning those registers in the assembly code (implicit modification)

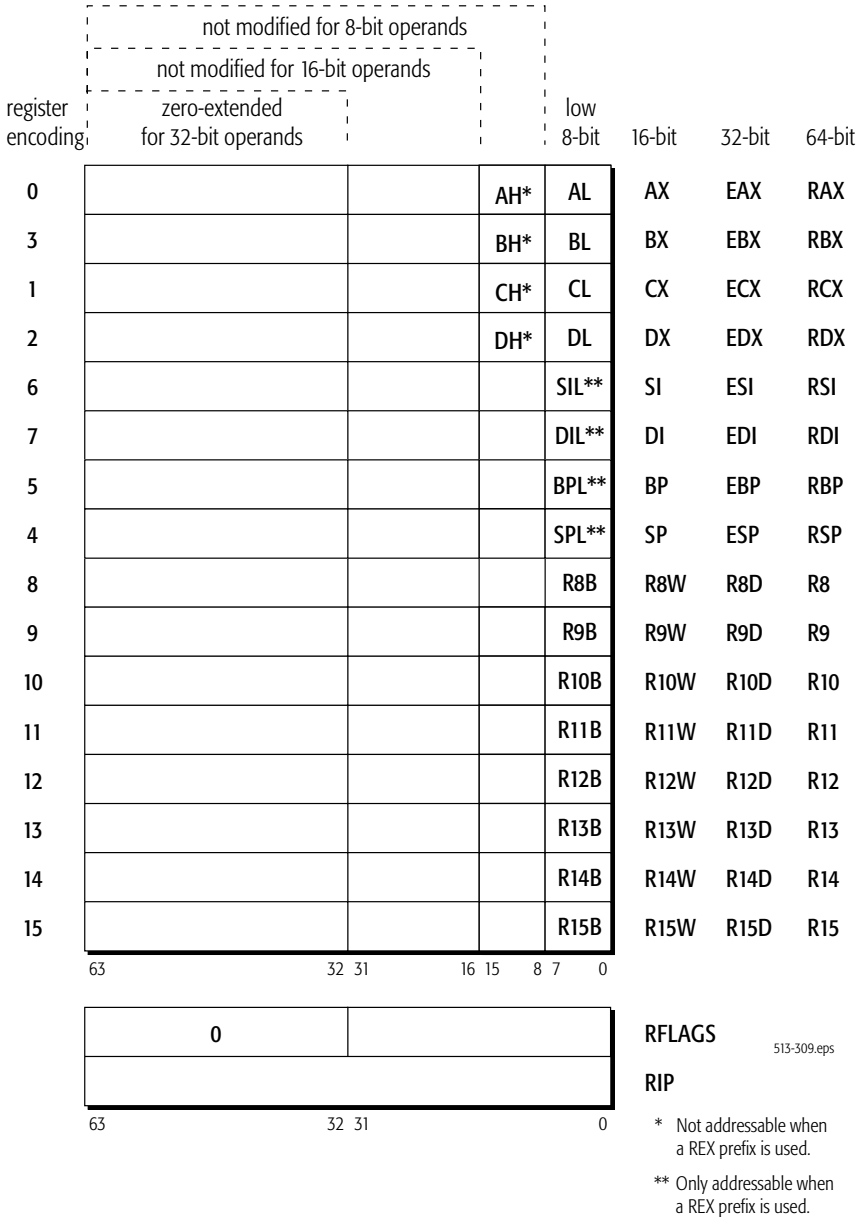


Figure 3-3. General Registers in 64-Bit Mode

Figure 3-4 on page 28 illustrates another way of viewing the 64-bit-mode GPRs, showing how the legacy GPRs overlap the extended GPRs. Gray-shaded bits are not modified in 64-bit mode.

Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

Figure 3.2: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed char`	1 to 8	-2^{w-1} to $2^{w-1}-1$
`char`		0 to 2^w-1
`unsigned char`		0 to 2^w-1
`signed short`	1 to 16	-2^{w-1} to $2^{w-1}-1$
`short`		0 to 2^w-1
`unsigned short`		0 to 2^w-1
`signed int`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`int`		0 to 2^w-1
`unsigned int`		0 to 2^w-1
`signed long`	1 to 64	-2^{w-1} to $2^{w-1}-1$
`long`		0 to 2^w-1
`unsigned long`		0 to 2^w-1

ative values), these bit-fields have the same range as a bit-field of the same size with the corresponding unsigned type. Bit-fields obey the same size and alignment rules as other structure and union members.

Also:

- bit-fields are allocated from right to left
- bit-fields must be contained in a storage unit appropriate for its declared type
- bit-fields may share a storage unit with other struct / union members

Unnamed bit-fields' types do not affect the alignment of a structure or union.

3.2 Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, parameter passing and so on.

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use dif-

ferent conventions. Nevertheless, it is recommended that all functions use the standard calling sequence when possible.

3.2.1 Registers and the Stack Frame

The AMD64 architecture provides 16 general purpose 64-bit registers. In addition the architecture provides 16 SSE registers, each 128 bits wide and 8 x87 floating point registers, each 80 bits wide. Each of the x87 floating point registers may be referred to in *MMX*/3DNow! mode as a 64-bit register. All of these registers are global to all procedures active for a given thread.

This subsection discusses usage of each register. Registers `%rbp`, `%rbx` and `%r12` through `%r15` "belong" to the calling function and the called function is required to preserve their values. In other words, a called function must preserve these registers' values for its caller. Remaining registers "belong" to the called function.[5] If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

The CPU shall be in x87 mode upon entry to a function. Therefore, every function that uses the *MMX* registers is required to issue an `emms` or `femms` instruction after using *MMX* registers, before returning or calling another function. [6] The direction flag `DF` in the `%rFLAGS` register must be clear (set to "forward" direction) on function entry and return. Other user flags have no specified role in the standard calling sequence and are *not* preserved across calls.

The control bits of the `MXCSR` register are callee-saved (preserved across calls), while the status bits are caller-saved (not preserved). The x87 status word register is caller-saved, whereas the x87 control word is callee-saved.

3.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Figure 3.3 shows the stack organization.

The end of the input argument area shall be aligned on a 16 byte boundary. In other words, the value $(\texttt{\%rsp} - 8)$ is always a multiple of 16 when control is

[5]Note that in contrast to the Intel386 ABI, `%rdi`, and `%rsi` belong to the called function, not the caller.

[6]All x87 registers are caller-saved, so callees that make use of the *MMX* registers may use the faster `femms` instruction.

Figure 3.3: Stack Frame with Base Pointer

Position	Contents	Frame
`8n+16(%rbp)`	memory argument eightbyte n	Previous
	...	
`16(%rbp)`	memory argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

transferred to the function entry point. The stack pointer, `%rsp`, always points to the end of the latest allocated stack frame. [7]

The 128-byte area beyond the location pointed to by `%rsp` is considered to be reserved and shall not be modified by signal or interrupt handlers.[8] Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.

3.2.3 Parameter Passing

After the argument values have been computed, they are placed either in registers or pushed on the stack. The way how values are passed is described in the following sections.

Definitions We first define a number of classes to classify arguments. The classes are corresponding to AMD64 register classes and defined as:

[7] The conventional use of `%rbp` as a frame pointer for the stack frame may be avoided by using `%rsp` (the stack pointer) to index into the stack frame. This technique saves two instructions in the prologue and epilogue and makes one additional general-purpose register (`%rbp`) available.

[8] Locations within 128 bytes can be addressed using one-byte displacements.

INTEGER This class consists of integral types that fit into one of the general purpose registers.

SSE The class consists of types that fits into a SSE register.

SSEUP The class consists of types that fit into a SSE register and can be passed and returned in the most significant half of it.

X87, X87UP These classes consists of types that will be returned via the x87 FPU.

COMPLEX_X87 This class consists of types that will be returned via the x87 FPU.

NO_CLASS This class is used as initializer in the algorithms. It will be used for padding and empty structures and unions.

MEMORY This class consists of types that will be passed and returned in memory via the stack.

Classification The size of each argument gets rounded up to eightbytes.[9]
The basic types are assigned their natural classes:

- Arguments of types (signed and unsigned) `_Bool`, `char`, `short`, `int`, `long`, `long long`, and pointers are in the INTEGER class.
- Arguments of types `float`, `double`, `_Decimal32`, `_Decimal64` and `__m64` are in class SSE.
- Arguments of types `__float128`, `_Decimal128` and `__m128` are split into two halves. The least significant ones belong to class SSE, the most significant one to class SSEUP.
- The 64-bit mantissa of arguments of type `long double` belongs to class X87, the 16-bit exponent plus 6 bytes of padding belongs to class X87UP.
- Arguments of type `__int128` offer the same operations as INTEGERs, yet they do not fit into one general purpose register but require two registers. For classification purposes `__int128` is treated as if it were implemented as:

[9] Therefore the stack will always be eightbyte aligned.

```
typedef struct {
  long low, high;
} __int128;
```

with the exception that arguments of type `__int128` that are stored in memory must be aligned on a 16-byte boundary.

- Arguments of `complex T` where `T` is one of the types `float or double` are treated as if they are implemented as:

```
struct complexT {
  T real;
  T imag;
};
```

- A variable of type `complex long double` is classified as type COMPLEX_X87.

The classification of aggregate (structures and arrays) and union types works as follows:

1. If the size of an object is larger than two eightbytes, or it contains unaligned fields, it has class MEMORY.
2. If a C++ object has either a non-trivial copy constructor or a non-trivial destructor [10] it is passed by invisible reference (the object is replaced in the parameter list by a pointer that has class INTEGER). [11]
3. If the size of the aggregate exceeds a single eightbyte, each is classified separately. Each eightbyte gets initialized to class NO_CLASS.

[10] A de/constructor is trivial if it is an implicitly-declared default de/constructor and if:

- its class has no virtual functions and no virtual base classes, and
- all the direct base classes of its class have trivial de/constructors, and
- for all the nonstatic data members of its class that are of class type (or array thereof), each such class has a trivial de/constructor.

[11] An object with either a non-trivial copy constructor or a non-trivial destructor cannot be passed by value because such objects must have well defined addresses. Similar issues apply when returning an object from a function.

4. Each field of an object is classified recursively so that always two fields are considered. The resulting class is calculated according to the classes of the fields in the eightbyte:

   (a) If both classes are equal, this is the resulting class.

   (b) If one of the classes is NO_CLASS, the resulting class is the other class.

   (c) If one of the classes is MEMORY, the result is the MEMORY class.

   (d) If one of the classes is INTEGER, the result is the INTEGER.

   (e) If one of the classes is X87, X87UP, COMPLEX_X87 class, MEMORY is used as class.

   (f) Otherwise class SSE is used.

5. Then a post merger cleanup is done:

   (a) If one of the classes is MEMORY, the whole argument is passed in memory.

   (b) If SSEUP is not preceeded by SSE, it is converted to SSE.

Passing Once arguments are classified, the registers get assigned (in left-to-right order) for passing as follows:

1. If the class is MEMORY, pass the argument on the stack.

2. If the class is INTEGER, the next available register of the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` is used[12].

3. If the class is SSE, the next available SSE register is used, the registers are taken in the order from `%xmm0` to `%xmm7`.

4. If the class is SSEUP, the eightbyte is passed in the upper half of the last used SSE register.

[12]Note that `%r11` is neither required to be preserved, nor is it used to pass arguments. Making this register available as scratch register means that code in the PLT need not spill any registers when computing the address to which control needs to be transferred. `%rax` is used to indicate the number of SSE arguments passed to a function requiring a variable number of arguments. `%r10` is used for passing a function's static chain pointer.

5. If the class is X87, X87UP or COMPLEX_X87, it is passed in memory.

When a value of type `_Bool` is passed in a register or on the stack, the upper 63 bits of the eightbyte shall be zero.

If there are no registers available for any eightbyte of an argument, the whole argument is passed on the stack. If registers have already been assigned for some eightbytes of such an argument, the assignments get reverted.

Once registers are assigned, the arguments passed in memory are pushed on the stack in reversed (right-to-left[13]) order.

For calls that may call functions that use varargs or stdargs (prototype-less calls or calls to functions containing ellipsis (...) in the declaration) `%al` [14] is used as hidden argument to specify the number of SSE registers used. The contents of `%al` do not need to match exactly the number of registers, but must be an upper bound on the number of SSE registers used and is in the range 0–8 inclusive.

Returning of Values The returning of values is done according to the following algorithm:

1. Classify the return type with the classification algorithm.
2. If the type has class MEMORY, then the caller provides space for the return value and passes the address of this storage in `%rdi` as if it were the first argument to the function. In effect, this address becomes a "hidden" first argument.

   On return `%rax` will contain the address that has been passed in by the caller in `%rdi`.
3. If the class is INTEGER, the next available register of the sequence `%rax`, `%rdx` is used.
4. If the class is SSE, the next available SSE register of the sequence `%xmm0`, `%xmm1` is used.
5. If the class is SSEUP, the eightbyte is passed in the upper half of the last used SSE register.

[13] Right-to-left order on the stack makes the handling of functions that take a variable number of arguments simpler. The location of the first argument can always be computed statically, based on the type of that argument. It would be difficult to compute the address of the first argument if the arguments were pushed in left-to-right order.

[14] Note that the rest of `%rax` is undefined, only the contents of `%al` is defined.

Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

6. If the class is X87, the value is returned on the X87 stack in `%st0` as 80-bit x87 number.

7. If the class is X87UP, the value is returned together with the previous X87 value in `%st0`.

8. If the class is COMPLEX_X87, the real part of the value is returned in `%st0` and the imaginary part in `%st1`.

As an example of the register passing conventions, consider the declarations and the function call shown in Figure 3.5. The corresponding register allocation is given in Figure 3.6, the stack frame offset given shows the frame before calling the function.

Figure 3.5: Parameter Passing Example

```
typedef struct {
  int a, b;
  double d;
} structparm;
structparm s;
int e, f, g, h, i, j, k;
long double ld;
double m, n;

extern void func (int e, int f,
                  structparm s, int g, int h,
                  long double ld, double m,
                  double n, int i, int j, int k);

func (e, f, s, g, h, ld, m, n, i, j, k);
```

Figure 3.1: Scalar Types

Type	C	sizeof	Alignment (bytes)	AMD64 Architecture
Integral	`_Bool`†	1	1	boolean
	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed twobyte
	`unsigned short`	2	2	unsigned twobyte

	`int`
	`signed int`
	`enum`†††
		4	4	signed fourbyte

	`unsigned int`	4	4	unsigned fourbyte

	`long`
	`signed long`
	`long long`
	`signed long long`
		8	8	signed eightbyte

	`unsigned long`	8	8	unsigned eightbyte
	`unsigned long long`	8	8	unsigned eightbyte
	`__int128`††	16	16	signed sixteenbyte
	`signed __int128`††	16	16	signed sixteenbyte
	`unsigned __int128`††	16	16	unsigned sixteenbyte
Pointer	*any-type* `*` *any-type* `(*)()`	8	8	unsigned eightbyte
Floating-point	`float`	4	4	single (IEEE-754)
	`double`	8	8	double (IEEE-754)
	`long double`	16	16	80-bit extended (IEEE-754)
	`__float128`††	16	16	128-bit extended (IEEE-754)
Decimal-floating-point	`_Decimal32`	4	4	32bit BID (IEEE-754R)
	`_Decimal64`	8	8	64bit BID (IEEE-754R)
	`_Decimal128`	16	16	128bit BID (IEEE-754R)
Packed	`__m64`††	8	8	*MMX* and 3DNow!
	`__m128`††	16	16	SSE and SSE-2

† This type is called `bool` in C++.

†† These types are optional.

††† C++ and some implementations of C permit enums larger than an int. The underlying type is bumped to an unsigned int, long int or unsigned long int, in that order.

Table 3-1. Implicit Uses of GPRs

Registers[1]				Name	Implicit Uses
Low 8-Bit	16-Bit	32-Bit	64-Bit		
AL	AX	EAX	RAX[2]	Accumulator	• Operand for decimal arithmetic, multiply, divide, string, compare-and-exchange, table-translation, and I/O instructions. • Special accumulator encoding for ADD, XOR, and MOV instructions. • Used with EDX to hold double-precision operands. • CPUID processor-feature information.
BL	BX	EBX	RBX[2]	Base	• Address generation in 16-bit code. • Memory address for XLAT instruction. • CPUID processor-feature information.
CL	CX	ECX	RCX[2]	Count	• Bit index for shift and rotate instructions. • Iteration count for loop and repeated string instructions. • Jump conditional if zero. • CPUID processor-feature information.
DL	DX	EDX	RDX[2]	I/O Address	• Operand for multiply and divide instructions. • Port number for I/O instructions. • Used with EAX to hold double-precision operands. • CPUID processor-feature information.
SIL[2]	SI	ESI	RSI[2]	Source Index	• Memory address of source operand for string instructions. • Memory index for 16-bit addresses.

Note:
1. *Gray-shaded registers have no implicit uses.*
2. *Accessible only in 64-bit mode.*

Table 3-1. Implicit Uses of GPRs (continued)

Registers[1]				Name	Implicit Uses
Low 8-Bit	16-Bit	32-Bit	64-Bit		
DIL[2]	DI	EDI	RDI[2]	Destination Index	• Memory address of destination operand for string instructions. • Memory index for 16-bit addresses.
BPL[2]	BP	EBP	RBP[2]	Base Pointer	• Memory address of stack-frame base pointer.
SPL[2]	SP	ESP	RSP[2]	Stack Pointer	• Memory address of last stack entry (top of stack).
R8B–R10B[2]	R8W–R10W[2]	R8D–R10D[2]	R8–R10[2]	None	No implicit uses
R11B[2]	R11W[2]	R11D[2]	R11[2]	None	• Holds the value of RFLAGS on SYSCALL/SYSRET.
R12B–R15B[2]	R12W–R15W[2]	R12D–R15D[2]	R12–R15[2]	None	No implicit uses

Note:
1. *Gray-shaded registers have no implicit uses.*
2. *Accessible only in 64-bit mode.*

Arithmetic Operations. Several forms of the add, subtract, multiply, and divide instructions use AL or rAX implicitly. The multiply and divide instructions also use the concatenation of rDX:rAX for double-sized results (multiplies) or quotient and remainder (divides).

Sign-Extensions. The instructions that double the size of operands by sign extension (for example, CBW, CWDE, CDQE, CWD, CDQ, CQO) use rAX register implicitly for the operand. The CWD, CDQ, and CQO instructions also uses the rDX register.

Special MOVs. The MOV instruction has several opcodes that implicitly use the AL or rAX register for one operand.

String Operations. Many types of string instructions use the accumulators implicitly. Load string, store string, and scan string instructions use AL or rAX for data and rDI or rSI for the offset of a memory address.

I/O-Address-Space Operations. The I/O and string I/O instructions use rAX to hold data that is received from or sent to a device located in the I/O-address space. DX holds the device I/O-address (the port number).

Table Translations. The table translate instruction (XLATB) uses AL for an memory index and rBX for memory base address.

Compares and Exchanges. Compare and exchange instructions (CMPXCHG) use the AL or rAX register for one operand.

Decimal Arithmetic. The decimal arithmetic instructions (AAA, AAD, AAM, AAS, DAA, DAS) that adjust binary-coded decimal (BCD) operands implicitly use the AL and AH register for their operations.

Shifts and Rotates. Shift and rotate instructions can use the CL register to specify the number of bits an operand is to be shifted or rotated.

Conditional Jumps. Special conditional-jump instructions use the rCX register instead of flags. The JCXZ and JrCXZ instructions check the value of the rCX register and pass control to the target instruction when the value of rCX register reaches 0.

Repeated String Operations. With the exception of I/O string instructions, all string operations use rSI as the source-operand pointer and rDI as the destination-operand pointer. I/O string instructions use rDX to specify the input-port or output-port number. For repeated string operations (those preceded with a repeat-instruction prefix), the rSI and rDI registers are incremented or decremented as the string elements are moved from the source location to the destination. Repeat-string operations also use rCX to hold the string length, and decrement it as data is moved from one location to the other.

Stack Operations. Stack operations make implicit use of the rSP register, and in some cases, the rBP register. The rSP register is used to hold the top-of-stack pointer (or simply, stack pointer). rSP is decremented when items are pushed onto the stack, and incremented when they are popped off the stack. The ENTER and LEAVE instructions use rBP as a stack-frame base pointer. Here, rBP points to the last entry in a data structure that is passed from one block-structured procedure to another.

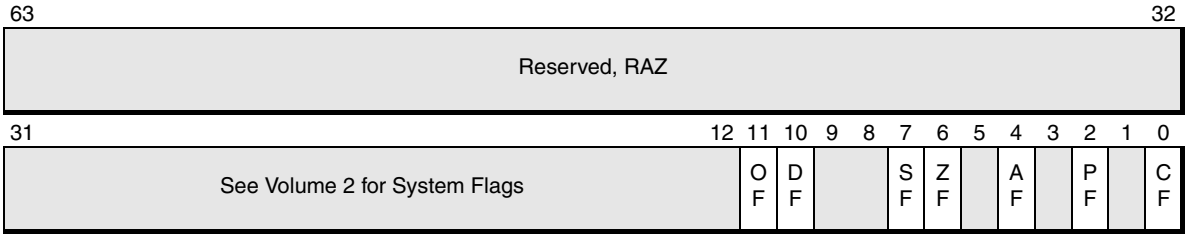
The use of rSP or rBP as a base register in an address calculation implies the use of SS (stack segment) as the default segment. Using any other GPR as a base register without a segment-override prefix implies the use of the DS data segment as the default segment.

The push all and pop all instructions (PUSHA, PUSHAD, POPA, POPAD) implicitly use all of the GPRs.

CPUID Information. The CPUID instruction makes implicit use of the EAX, EBX, ECX, and EDX registers. Software loads a function code into EAX, executes the CPUID instruction, and then reads the associated processor-feature information in EAX, EBX, ECX, and EDX.

3.1.4 Flags Register

Figure 3-5 on page 34 shows the 64-bit RFLAGS register and the flag bits visible to application software. Bits 15–0 are the FLAGS register (accessed in legacy real and virtual-8086 modes), bits 31–0 are the EFLAGS register (accessed in legacy protected mode and compatibility mode), and bits 63–0 are the RFLAGS register (accessed in 64-bit mode). The name *rFLAGS* refers to any of the three register widths, depending on the current software context.



Bits	Mnemonic	Description	R/W
11	OF	Overflow Flag	R/W
10	DF	Direction Flag	R/W
7	SF	Sign Flag	R/W
6	ZF	Zero Flag	R/W
4	AF	Auxiliary Carry Flag	R/W
2	PF	Parity Flag	R/W
0	CF	Carry Flag	R/W

Figure 3-5. rFLAGS Register—Flags Visible to Application Software

The low 16 bits (FLAGS portion) of rFLAGS are accessible by application software and hold the following flags:

- One control flag (the direction flag DF).
- Six status flags (carry flag CF, parity flag PF, auxiliary carry flag AF, zero flag ZF, sign flag SF, and overflow flag OF).

The direction flag (DF) flag controls the direction of string operations. The status flags provide result information from logical and arithmetic operations and control information for conditional move and jump instructions.

Bits 31–16 of the rFLAGS register contain flags that are accessible only to system software. These flags are described in “System Registers” in Volume 2. The highest 32 bits of RFLAGS are reserved. In 64-bit mode, writes to these bits are ignored. They are read as zeros (RAZ). The rFLAGS register is initialized to 02h on reset, so that all of the programmable bits are cleared to zero.

The effects that rFLAGS bit-values have on instructions are summarized in the following places:

- Conditional Moves (CMOV*cc*)—Table 3-4 on page 43.
- Conditional Jumps (J*cc*)—Table 3-5 on page 55.
- Conditional Sets (SET*cc*)—Table 3-6 on page 59.

The effects that instructions have on rFLAGS bit-values are summarized in “Instruction Effects on RFLAGS” in Volume 3.

target offset of the JMP instruction is ignored, and the new values loaded into CS and rIP are taken from the call gate or from the TSS.

Conditional Jump

- J*cc*—Jump if *condition*

Conditional jump instructions jump to an instruction specified by the operand, depending on the state of flags in the rFLAGS register. The operands specifies a signed relative offset from the current contents of the rIP. If the state of the corresponding flags meets the condition, a conditional jump instruction passes control to the target instruction, otherwise control is passed to the instruction following the conditional jump instruction. The flags tested by a specific J*cc* instruction depend on the opcode. In several cases, multiple mnemonics correspond to one opcode.

Table 3-6 shows the rFLAGS values required for each J*cc* instruction.

Table 3-6. rFLAGS for J*cc* Instructions

Mnemonic	Required Flag State	Description
JO	OF = 1	Jump near if overflow
JNO	OF = 0	Jump near if not overflow
JB JC JNAE	CF = 1	Jump near if below Jump near if carry Jump near if not above or equal
JNB JNC JAE	CF = 0	Jump near if not below Jump near if not carry Jump near if above or equal
JZ JE	ZF = 1	Jump near if 0 Jump near if equal
JNZ JNE	ZF = 0	Jump near if not zero Jump near if not equal
JNA JBE	CF = 1 or ZF = 1	Jump near if not above Jump near if below or equal
JNBE JA	CF = 0 and ZF = 0	Jump near if not below or equal Jump near if above
JS	SF = 1	Jump near if sign
JNS	SF = 0	Jump near if not sign
JP JPE	PF = 1	Jump near if parity Jump near if parity even
JNP JPO	PF = 0	Jump near if not parity Jump near if parity odd
JL JNGE	SF <> OF	Jump near if less Jump near if not greater or equal

Table 3-6. rFLAGS for J*cc* Instructions (continued)

Mnemonic	Required Flag State	Description
JGE JNL	SF = OF	Jump near if greater or equal Jump near if not less
JNG JLE	ZF = 1 or SF <> OF	Jump near if not greater Jump near if less or equal
JNLE JG	ZF = 0 and SF = OF	Jump near if not less or equal Jump near if greater

Unlike the unconditional jump (JMP), conditional jump instructions have only two forms—*near conditional jumps* and *short conditional jumps*. To create a far-conditional-jump code sequence corresponding to a high-level language statement like:

```
IF A = B THEN GOTO FarLabel
```

where FarLabel is located in another code segment, use the opposite condition in a conditional short jump before the unconditional far jump. For example:

```
      cmp   A,B               ; compare operands
      jne   NextInstr         ; continue program if not equal
      jmp   far ptr WhenNE    ; far jump if operands are equal
NextInstr:                    ; continue program
```

Three special conditional jump instructions use the rCX register instead of flags. The JCXZ, JECXZ, and JRCXZ instructions check the value of the CX, ECX, and RCX registers, respectively, and pass control to the target instruction when the value of rCX register reaches 0. These instructions are often used to control safe cycles, preventing execution when the value in rCX reaches 0.

Loop

- LOOP*cc*—Loop if *condition*

The LOOP*cc* instructions include LOOPE, LOOPNE, LOOPNZ, and LOOPZ. These instructions decrement the rCX register by 1 without changing any flags, and then check to see if the loop condition is met. If the condition is met, the program jumps to the specified target code.

LOOPE and LOOPZ are synonyms. Their loop condition is met if the value of the rCX register is non-zero and the zero flag (ZF) is set to 1 when the instruction starts. LOOPNE and LOOPNZ are also synonyms. Their loop condition is met if the value of the rCX register is non-zero and the ZF flag is cleared to 0 when the instruction starts. LOOP, unlike the other mnemonics, does not check the ZF flag. Its loop condition is met if the value of the rCX register is non-zero.

Call

- CALL—Procedure Call

The CALL instruction performs a call to a procedure whose address is specified in the operand. The return address is placed on the stack by the CALL, and points to the instruction immediately following

In most instructions that take two operands, the first (left-most) operand is both a source operand and the destination operand. The second (right-most) operand serves only as a source. Instructions can have one or more prefixes that modify default instruction functions or operand properties. These prefixes are summarized in Section 3.5, "Instruction Prefixes," on page 71. Instructions that access 64-bit operands in a general-purpose register (GPR) or any of the extended GPR or XMM registers require a REX instruction prefix.

Unless otherwise stated in this section, the word *register* means a general-purpose register (GPR). Several instructions affect the flag bits in the RFLAGS register. "Instruction Effects on RFLAGS" in Volume 3 summarizes the effects that instructions have on rFLAGS bits.

3.3.2 Data Transfer

The data-transfer instructions copy data between registers and memory.

Move

- MOV—Move
- MOVSX—Move with Sign-Extend
- MOVZX—Move with Zero-Extend
- MOVD—Move Doubleword or Quadword
- MOVNTI—Move Non-Temporal Doubleword or Quadword

MOV*x* copies a byte, word, doubleword, or quadword from a register or memory location to a register or memory location. The source and destination cannot both be memory locations. An immediate constant can be used as a source operand with the MOV instruction. For MOV, the destination must be of the same size as the source, but the MOVSX and MOVZX instructions copy values of smaller size to a larger size by using sign-extension or zero-extension. The MOVD instruction copies a doubleword or quadword between a general-purpose register or memory and an XMM or MMX register.

The MOV instruction is in many aspects similar to the assignment operator in high-level languages. The simplest example of their use is to initialize variables. To initialize a register to 0, rather than using a MOV instruction it may be more efficient to use the XOR instruction with identical destination and source operands.

The MOVNTI instruction stores a doubleword or quadword from a register into memory as "non-temporal" data, which assumes a single access (as opposed to frequent subsequent accesses of "temporal data"). The operation therefore minimizes cache pollution. The exact method by which cache pollution is minimized depends on the hardware implementation of the instruction. For further information, see Section 3.9, "Memory Optimization," on page 92.

Conditional Move

- CMOV*cc*—Conditional Move If *condition*

The CMOV*cc* instructions conditionally copy a word, doubleword, or quadword from a register or memory location to a register location. The source and destination must be of the same size.

The CMOV*cc* instructions perform the same task as MOV but work conditionally, depending on the state of status flags in the RFLAGS register. If the condition is not satisfied, the instruction has no effect and control is passed to the next instruction. The mnemonics of CMOV*cc* instructions indicate the condition that must be satisfied. Several mnemonics are often used for one opcode to make the mnemonics easier to remember. For example, CMOVE (conditional move if equal) and CMOVZ (conditional move if zero) are aliases and compile to the same opcode. Table 3-4 shows the RFLAGS values required for each CMOV*cc* instruction.

In assembly languages, the conditional move instructions correspond to small conditional statements like:

```
IF a = b THEN x = y
```

CMOV*cc* instructions can replace two instructions—a conditional jump and a move. For example, to perform a high-level statement like:

```
IF ECX = 5 THEN EAX = EBX
```

without a CMOV*cc* instruction, the code would look like:

```
cmp ecx, 5        ; test if ecx equals 5
jnz Continue      ; test condition and skip if not met
mov eax, ebx      ; move
Continue:         ; continuation
```

but with a CMOV*cc* instruction, the code would look like:

```
cmp ecx, 5        ; test if ecx equals to 5
cmovz eax, ebx    ; test condition and move
```

Replacing conditional jumps with conditional moves also has the advantage that it can avoid branch-prediction penalties that may be caused by conditional jumps.

Support for CMOV*cc* instructions depends on the processor implementation. To find out if a processor is able to perform CMOV*cc* instructions, use the CPUID instruction.

Table 3-4. rFLAGS for CMOV*cc* Instructions

Mnemonic	Required Flag State	Description
CMOVO	OF = 1	Conditional move if overflow
CMOVNO	OF = 0	Conditional move if not overflow
CMOVB CMOVC CMOVNAE	CF = 1	Conditional move if below Conditional move if carry Conditional move if not above or equal
CMOVAE CMOVNB CMOVNC	CF = 0	Conditional move if above or equal Conditional move if not below Conditional move if not carry
CMOVE CMOVZ	ZF = 1	Conditional move if equal Conditional move if zero

Table 3-4. rFLAGS for CMOV*cc* Instructions (continued)

Mnemonic	Required Flag State	Description
CMOVNE CMOVNZ	ZF = 0	Conditional move if not equal Conditional move if not zero
CMOVBE CMOVNA	CF = 1 or ZF = 1	Conditional move if below or equal Conditional move if not above
CMOVA CMOVNBE	CF = 0 and ZF = 0	Conditional move if not below or equal Conditional move if not below or equal
CMOVS	SF = 1	Conditional move if sign
CMOVNS	SF = 0	Conditional move if not sign
CMOVP CMOVPE	PF = 1	Conditional move if parity Conditional move if parity even
CMOVNP CMOVPO	PF = 0	Conditional move if not parity Conditional move if parity odd
CMOVL CMOVNGE	SF <> OF	Conditional move if less Conditional move if not greater or equal
CMOVGE CMOVNL	SF = OF	Conditional move if greater or equal Conditional move if not less
CMOVLE CMOVNG	ZF = 1 or SF <> OF	Conditional move if less or equal Conditional move if not greater
CMOVG CMOVNLE	ZF = 0 and SF = OF	Conditional move if greater Conditional move if not less or equal

Stack Operations

- POP—Pop Stack
- POPA—Pop All to GPR Words
- POPAD—Pop All to GPR Doublewords
- PUSH—Push onto Stack
- PUSHA—Push All GPR Words onto Stack
- PUSHAD—Push All GPR Doublewords onto Stack
- ENTER—Create Procedure Stack Frame
- LEAVE—Delete Procedure Stack Frame

PUSH copies the specified register, memory location, or immediate value to the top of stack. This instruction decrements the stack pointer by 2, 4, or 8, depending on the operand size, and then copies the operand into the memory location pointed to by SS:rSP.

POP copies a word, doubleword, or quadword from the memory location pointed to by the SS:rSP registers (the top of stack) to a specified register or memory location. Then, the rSP register is incremented by 2, 4, or 8. After the POP operation, rSP points to the new top of stack.