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);
```