System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.96

Edited by
Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitchell[4]

June 14, 2005

[1]matz@suse.de
[2]jh@suse.cz
[3]aj@suse.de
[4]mark@codesourcery.com

List of Tables

List of Figures

0.93 Add sections about program headers, new section types and special sections for unwinding information. Thanks to Michael Walker.

0.92 Fix some typos (thanks to BryaoaFord-221ader32(aTd[(Aix)-237(sectiaoa-237(aboix)-tackaboix)la

Chapter 1

Introduction

The AMD64[1] architecture[2] architecture1

- Sizes of fundamental data types.
- Parameter-passing conventions.
- Floating-point computations.
- Removal of the GOT register.
- UseTd[(ž)]TJ/mLA(T)-250(locutations.)] 162.018 -466.75 0 Td917

Chapter 2

Software Installation

No changes required.

Chapter 3

Low Level System Information

3.1 Machine Interface

Figure 3.1: Scalar Types

Type	C		Alignment	AMD64

`double` requires 16 bytes of storage, only the first 10 bytes are significant. The remaining six bytes are tail padding, and the contents of these bytes are undefined.

The `__int128`

Figure 3.2: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed longS2el7l9`	`-25.334`	`-62.314 -31.38 cm q []0 199 m00.3 i`

Figure 3.3: Stack Frame with Base Pointer

Position	Contents	Frame
`8n+16(%rbp)`	argument eightbyte n	Previous
	...	
`16(%rbp)`	argument eightbyte `0`	
`8(%rbp)`	return address	
`0(%rbp)`	previous `%rbp` value	

The end of the input argument area shall be aligned on a 16 byte boundary. In other words, the value (`%rsp - 8`)

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

Classification

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.

Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
`%rax`	temporary register; with variable ar-	

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.

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, consMntiond.

Figure 3.5: Parameter Passing Example

3.3 Operating System Interface

3.3.1 Exception Interface

Table 3.1: Hardware Exceptions and Signals

Number	Exception name	Signal
0	divide error fault	SIGFPE
1		

Figure 3.11: Auxiliary Vector Types

Name	Value	a_un
AT_NULL	0	ignored

AT_PHENT The `a_val` member of this entry holds the size, in bytes, of one entry in the program header table to which the `AT_PHDR` entry points.

AT_PHNUM The `a_val` member of this entry holds the number of entries in the program header table to which the `AT_PHDR` entry points.

AT_PAGESZ If present, this entry's

3.5.1 Architectural Constraints

The AMD64 architecture usually does not allow an instruction to encode arbitrary 64-bit constants as immediate operand. Most instructions accept 32-bit immediates that are sign extended to the 64-bit ones. Additionally the 32-bit operations with register destinations implicitly perform zero extension making loads of 64-bit immediates with upper half set to 0 even cheaper.

Additionally the branch instructions accept 32-bit immediate operands that are sign extended and used to adjust the instruction pointer. Similarly an instruction

model and the large data section having no limits except for available addressing space. The program layout must be set in a way so that large data sections (`.ldata`, `.lrodata`, `.lbss`) come after the text and data sections.

Instead an unwind sequence consisting of `movabs`, `lea` and `add` needs to be used.

3.5.3 Position-Independent Function Prologue

In the small code model AMD64 does not need any function prologue for cal-

Because only the `movabs` instruction uses 64-bit addresses directly, depend-

Figure 3.14: Position-Independent Load and Store (Small PIC Model)

```
extern int src[65536];
extern int dst[65536];
extern int *ptr;
static int lsrc[65536];
```

```
.extern  src
.extern  dst
.extern  ptr
```

Medium models

JItrem AbsolutemMedium

Figure 3.16: Position-Independent Load and Store (Medium PIC Model)

```
extern int src[65536];
extern int dst[65536];
extern int *ptr;
static int lsrc[65536];

static int ldst[65536];

static int *lptr;


dst[0] = src[0];
```

```
.extern  src
.extern  dst
.extern  ptr
.local   lsrc
.comm    lsrc,262144,4
.local   ldst
.comm    ldst,262144,4
.local   lptr
.comm    lptr,8,8
.text
movq     src@GOTPCREL(%rip), %rax
movl     (%rax), %edx
```

Figure 3.18: Absolute Global Data Load and Store

```
static int src;
static int dst;
extern int *ptr;
```

```
Lsrc: .long
Ldst: .long
      .extern  ptr
```

```
static int src;
```

Figure 3.24: Absolute Direct and Indirect Function Call

`static void (*ptr) (void);`
`extern void foo (void);`
`static void bar (void);`
	`Lptr: .quad`
	`.globl foo`
	`Lbar: ...`

`foo ();`	

not enough to address the branch target. Therefore, a branch target address is calculated explicitly [21]. For absolute objects:

Figure 3.26: Absolute Branching Code

`if (!a)`
`{`
 	
`...`
`}`
	`testl %eax,%eax`
	`jnz 1f`
	`movabs $2f,%r11 ; R_X86_64`
	`jmpq *%r11`
	`1: ...`
	`2:`

`goto Label;`
 	
`...`
`Label:`
	`movabs $Label,%r11 ; R_X86_64`
	`jmpq *%r11`
	`...`
	`Label:`

Figure 3.31: Parameter Passing Example with Variable-Argument List

```
int a, b;
long double ld;
double m, n;

extern void func (int a, double m,...);

func (a, m, b, ld, n);
```

Figure 3.32: Register Allocation Example for Variable-Argument List

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rdi: a	%xmm0: m	0: ld
%rsi: b	%xmm1: n	
%rax: 2		

The Re-14.4 cm q [tf

overflow_arg_area

5. Set:

$$\texttt{l->gp\_offset} = \texttt{l->gp\_offset} + \texttt{num\_gp} \times 8$$

$$\texttt{l->fp\_offset} = \texttt{l->fp\_offset} + \texttt{num\_fp} \times 16.$$

6. Return the fetched `type`.

7. Align `l->overflow_arg_area` upwards to a 16 byte boundary if alignment needed by `type` exceeds 8 byte boundary.

8. Fetch `type` from `l->overflow_arg_area`.

Position independence In order to avoid load time relocations for position independent code, the FDE CIE offset pointer should be stored relative to the start of CIE table entry. Frames using this extension of the DWARF stan-

4.2 Sections

4.2.1 Section Flags

In order to allow linking object files of different code models, it is necessary to provide for a way to differentiate those sections which may hold more than 2GB

4.2.3 Special Sections

Table 4.4: Special sections

Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.eh_frame		

_PROGBITS

`.plt .init .fini .text .got .rodata .rodata1 .data .data1 .bss`
: These sections can have a combined size of up to 2GB.

`.lplt .ltext .lgot .lrodata .lrodata1 .ldata .ldata1 .lbss`
: These sections plus the above can have a combined size of up to 16EB.

4.2.4 EH_FRAME sections

The call frame information needed for unwinding the stack is output into one or more ELF sections of type `SHT_X86_64_UNWIND`. In the simplest case there will be one such section per object file and it will be named `.eh_frame`. An `.eh_frame`

Table 4.6: Common Information Entry (CIE)

Field	Length (byte)	Description
Length	4	Length of the CIE (not including this 4-byte field)

Table 4.7: CIE Augmentation Section Content

Char	Operands	Length (byte)	Description
z	size	uleb128	Length of the remainder of the Augmentation Section
P	personality_enc	1	

Table 4.8: Frame Descriptor Entry (FDE)

Field	Length (byte)	Description
Length	4	Length of the FDE (not including this 4-

Table 4.9: FDE Augmentation Section Content

Char	Operands	Length (byte)	Description
z			

GOT

Chapter 5

[redacted]
reside at the virtual addresses used to build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This lets a segments virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments' relative positions. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file.

5.1.1isoPrograme

Global Offset Table (GOT)

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and shareability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

global offset table entry used in the previous `jmp` instruction. The relocation entry contains a symbol table index that will reference the appropriate symbol, `name1` in the example.

6. After pushing the relocation index, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushq` instruction places the value of the second global offset table entry (GOT+8) on the stack, thus giving the dynamic linker one word of identifying information. The program then jumps to the address in the third global offset table entry (GOT+16),

The PLT is extended as shown in figure 5.3 with the assumption that the GOT address is in `%r15`

the large code model, only `-fPIC` is allowed. Using the option `-fpic` with the large code model remains reserved for future use.

5.2.1 Program Interpreter

Thei.18 d[(is18 donei.18 dvlo)2alidi.18710(8(ogram18 d[(Intpr8(etem18 d0(for18 d0(8(ogr(is18 dcon0(f

Chapter 6

Libraries

A further review of the Intel386 ABI is needed.

6.1 C Library

6.1.1 Global Data Symbols

The symbols

6.2 Unwind Library Interface

This section defines the Unwind Library interface [1], expected to be provided by

6.2.1 Exception Handler Framework

Reasons for Unwinding

There are two major reasons for unwinding the stack:

- exceptions, as defined by languages that support them (such as C++)
- "forced" unwinding (such as caused by `longjmp` or thread termination)

The interface described here tries to keep both similar. There is a major difference, however.

- In the case where an exception is thrown, the stack is unwound while the exception propagates, but it is expected that the personality routine for each stack frame knows whether it wants to catch the exception or pass it through.

The Unwind Process

stops at each catch clause, and if it needs to restart, restarts at phase 1. This process is not needed for destructors (cleanup code), so the phase 1 can safely process all destructor-only frames at once and stop at the next enclosing catch clause.

```
typedef void (*_Unwind_Exception_Cleanup_Fn)
  (_Unwind_Reason_Code reason,
   struct _Unwind_Exception *exc);
struct _Unwind_Exception {
   uint64                          exception_class;
   _Unwind_Exception_Cleanup_Fn    exception_cleanup;
   uint64                          private_1;
   uint64                          private_2;
};
```

An _Unwind_Exception

The private unwinder state (`private_1` and `private_2`) in an exception object should be neither read by nor written to by personality routines or other parts of the language-specific runtime. It is used by the specific implementation of the unwinder on the host to store internal information, for instance to remember the final handler frame between unwinding phases.

In addition to the above information, a typical runtime such as the C++ runtime will add language-specific information used to process the exception. This is expected to be a contiguous area of memory after the `_Unwind_Exception` object, but this is not required as long as the matching personality routines know how to deal with it, and the `exception_cleanup`

rameters described for the usual personality routine below, plus an additional `stop_parameter`.

When the `stop`

_Unwind_Resume

```
void _Unwind_Resume
  (struct _Unwind_Exception *exception_object);
```

Resume propagation of an existing exception e.g. after executing cleanup code

_Unwind_GetGR

```
uint64 _Unwind_GetGR
  (struct _Unwind_Context *context, int index);
```

This function returns the 64-bit value of the given general register. The register is identified by its index as given in 3.36.

During the two phases of unwinding, no registers have a guaranteed value.

_Unwind_SetGR

```
void _Unwind_SetGR
  (struct _Unwind_Context *context,
   int index,
   uint64 new_value);
```

This function sets the 64-bit value of the given register, identified by its index as for `_Unwind_GetGR`.

The behavior is guaranteed only if the function is called during phase 2 of unwinding, and applied to an unwind context representing a handler frame, for which the personality routine will return `_URC_INSTALL_CONTEXT`, and that the landing pads.

_Unwind_GetIP

```
uint64 _Unwind_GetIP
```

The behavior is guaranteed only when this function is called for an unwind context representing a handler frame, for which the personality routine will return `_URC_INSTALL_CONTEXT`

6.2.6 Personality Routine

```
_Unwind_Reason_Code (*__personality_routine)
  (int version,
   _Unwind_Action actions,
   uint64 exceptionClass,
   str2duU4Eint64*uint64
```

return value The return value from the personality routine indicates how further unwind should happen, as well as possible error conditions. See the following section.

Personality Routine Actions

Transferring Control to a Landing Pad

If the personality routine determines that it should transfer control to a landing

a foreign language are mapped to the native language in that frame.

If a runtime resumes normal execution, and the caught excepaugh9fandmaame.-

`.cfi_rel_offset REGISTER, OFFSET` saves the previous value of REGISTER at offset OFFSET from the current CFA register. This is transformed to `.cfi_offset` using the known displacement of the CFA register from the CFA. This is often easier to use, because the number will match the code it is annotating.

`.cfi_escape EXPRESSION[, ...]` allows the user to add arbitrary bytes to the unwind info. One might use this to add OS-specific CFI opcodes, or generic CFI opcodes that the assembler does not support.

Chapter 7

Development Environment

During compilation of C or C++ code at least the symbols in table 7.1 are defined

Chapter 8

Chapter 9

Conventions

9.3 Fortran

Appendix A

Linux Conventions

This chapter describes some details that are only relevant to GNU/Linux systems and the Linux kernel.

A.1 Execution of 32-bit Programs

l eI oparticular40(v,-250(the)

The AMD64 processors are able to execute 64-bit AMD64 and also 32-bit ia32 programs. Libraries conforming to the Intel386 ABI will live in the normal places like `/lib`, `/usr/lib` and `/usr/bin`. Libraries following the AMD64, will use `lib64` subdirectories for the libraries, e.g `/lib64` and `/usr/lib64`le-]TJ -131.3318-14.446 Td[(

`/usr/bin`

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`

Figure A.1: Required Processor Features

Feature	Comment
Features need for programs	
fpu	Necessary for `long double`, MMX
tsc	User-visible

Index