

В связи с тем, что в некоторых современных моделях ПК нет возможности из коробки запускать ассемблерные команды под архитектуру IA-32, то возникает необходимость писать сразу под AMD64. Ниже представлены их основные различия и примеры кода. Соглашения о вызовах тоже довольно существенно меняются.

Что именно сравниваем

Архитектура + ABI	Где встречается	Что «входит в пакет»
System V AMD64	Linux, macOS, BSD (-m64)	правила для x86-64: регистры XMM/GPR, 16-байтовое выравнивание стека, <i>red zone</i> 128 B, динамическая линковка ELF/Mach-O, DWARF unwinding
cdecl (IA-32)	POSIX-сборки GCC/Clang, смешанный код под Windows	стек-ориентированное соглашение «caller-clean», порядок push right-to-left
stdcall (IA-32)	Win32 API, MSVC, Delphi	то же, но стек чистит <i>callee</i> ; имя экспортируется как <code>_Func@N</code>

Ниже — 10 главных пунктов, где эти соглашения ведут себя по-разному, и почему.

1. Как передаются аргументы

Тип	System V AMD64	cdecl	stdcall
int/ptr	RDI, RSI, RDX, RCX, R8, R9 → затем стек	все на стеке (RTL)	то же
double/float	XMM0 ... XMM7	x87 ST(0)... или XMM0 (если включён SSE)	то же
структуры ≤ 16 B	присылаются в рег. пару	на стеке (копия)	на стеке

Тип	System V AMD64	cdecl	stdcall
variadic	регистры и дубли на стеке	нет дублирования — всё уже на стеке	то же

Почему: у 64-битного ABI есть полноценный регистровый файл (16 GPR + 16 XMM) — дешевле держать первые аргументы там. В IA-32 регистров мало, поэтому компиляторы исторически уводили всё в стек. (refspecs.linuxbase.org)

2. Порядок и способ очистки стека

ABI	Кто убирает аргументы	Инструкция «RET n»
System V AMD64	аргументы лежат в регистрах → чистить нечего	используется <code>ret</code> без смещения
cdecl	Caller (<code>add esp, size</code>)	<code>ret</code> (без n)
stdcall	Callee (<code>ret size</code>)	<code>ret n</code>

Практический эффект: функция `foo()` в `stdcall` почти всегда заканчивается `ret 12`, а та же в `cdecl` — `ret`. На ассемблерном уровне это первый визуальный маркер. ([Википедия](#), [Stack Overflow](#))

3. Выравнивание и *red zone*

- **AMD64:** до `call` стек **уже** должен быть выровнен на 16 B. Под macOS/Linux разрешено использовать невидимую «red zone» (-128...-1 от RSP) для временных буферов — прерывания её не трогают.
- **IA-32:** гарантируется лишь кратность 4 B; red zone отсутствует — прерывание может затирать всё ниже ESP.

Поэтому портируя 32-битный «ручной» код в 64 бита, нельзя просто заменить `push / pop` — придётся обеспечить выравнивание перед каждым `call`. ([OSDev Вики](#))

4. Сохраняемые (callee-saved) регистры

ABI	Callee обязан сохранить
System V AMD64	RBX, RBP, R12–R15, XMM6–XMM15
cdecl/stdcall	EBX, EBP, ESI, EDI (+ для stdcall — все SSE за исключением стандарта)

Следствие: в 64-битном коде у нас больше «caller-save» регистров (например RAX, RCX, RDX, XMM0–5) — это ускоряет короткие leaf-функции, потому что им не нужен пролог/эпилог для сохранения. (refspecs.linuxbase.org)

5. Возврат значений

- **Скалярные:**
 - AMD64 — RAX (int) / XMM0 (double/float)
 - IA-32 — EAX (int) / ST0 (float)
- **Структуры > 16 В:**
 - AMD64 — скрытый первый аргумент «sret» (адрес приёмника) в RDI
 - IA-32 — тот же приём, но адрес лежит по [esp+4]

6. Имена символов (mangling)

Платформа	cdecl	stdcall	System V AMD64
Win x86	_foo	_foo@N	— (не применимо)
Linux/mac x86	foo	не используется	foo

Это важно при связывании asm ↔ C: под Windows вам придётся писать `global _foo@8`, а под Linux — просто `global foo`. ([Википедия](#))

7. Var-args и va_list

- В **cdecl** аргументы уже на стеке — `va_start` просто берёт следующий адрес после последнего именованного.
- В **System V AMD64** компилятор обязан скопировать все регистровые аргументы в «overflow area» (тень на стеке) — так `va_arg` может доставать их, не зная о регистрах. Это причина, почему даже leaf-функция `printf` получает 48 байт «home-area» под каждым вызовом.

8. Исключения и unwinding

- В 64-битных UNIX-ах используется *DWARF CFI* — стековое раскрутчик считывает CIE/FDE-записи, написанные компилятором.
- IA-32-cdecl к этому равнодушен; stdcall под Windows хранит unwind-информацию в *SEH* таблицах, но это часть PE/SEH, а не ABI.

9. Shadow / home area

Только Microsoft-x64 ABI (не System V) резервирует 32 байта «shadow space» под каждый `call`. System V его **не** имеет; путаница возникает потому, что stdcall «переехал» в Windows-x64 именно в этом виде, но вопрос был про System V, так что помнить: *home-area в SysV есть у var-args, shadow-area — нет.*

10. Практические советы ассемблеру

1. **Сделайте макрос «ALIGN_STACK»** — перед любым `call` в 64-битном коде.
2. **Не забудьте про red zone:** `sub rsp, n` внутри сигнально-чувствительного кода.
3. **Var-args на AMD64:** сохраните (`mov`) XMM / GPR аргументы в отведённое компилятором место, если пишете «ручной» пролог.
4. **stdcall vs cdecl:** решающим остаётся `ret n` и порядок очистки — легко диагностировать дизассемблером.

11. Сравнение кода

```
; ----- functions32.asm -----  
; NASM, 32-битный режим, объектный формат elf32  
;  
; Аргументы-double передаются стеком:  
;   esp+4   : первый (x)  
;   esp+12  : второй (a)   – для df_poly2  
;   esp+20  : третий  (b)   – для df_poly2  
; Результат-double возвращается в ST(0) (x87 FPU).
```

BITS 32

```
section .data
```

```
    a1    dq 0.35  
    b1    dq -0.95  
    c1    dq 2.7  
    b2    dq 3.0  
    c2    dq 1.0  
    two   dq 2.0  
    one   dq 1.0
```

```
section .text
```

```
global f1, f2, f3, df_poly2, df_f3
```

```
; -----  
;    $f1(x) = 0.35 \cdot x^2 - 0.95 \cdot x + 2.7$   
; -----
```

```
f1:
```

```
    push    ebp  
    mov     ebp, esp  
  
    fld     qword [ebp+8]      ; x  
    fld     st0                ; x, x  
    fmulp   st1, st0           ;  $x^2$   
    fmul    qword [a1]         ;  $0.35 \cdot x^2$   
  
    fld     qword [ebp+8]      ; x  
    fmul    qword [b1]         ;  $-0.95 \cdot x$   
  
    faddp   st1, st0           ;  $0.35 \cdot x^2 - 0.95 \cdot x$   
    fadd    qword [c1]         ; ... + 2.7
```

```
    pop    ebp
    ret
```

```
; -----
;  f2(x) = 3·x + 1
; -----
```

f2:

```
    push   ebp
    mov     ebp, esp

    fld     qword [ebp+8]      ; x
    fmul    qword [b2]        ; 3·x
    fadd     qword [c2]        ; +1

    pop     ebp
    ret
```

```
; -----
;  f3(x) = 1 / (x + 2)
; -----
```

f3:

```
    push   ebp
    mov     ebp, esp

    fld     qword [ebp+8]      ; x
    fadd     qword [two]       ; x + 2
    fld1                    ; 1.0 (из x87 короткая команда)
    fdivrp   st1, st0          ; 1 / (x+2)

    pop     ebp
    ret
```

```
; -----
;  df_poly2(x,a,b) = 2·a·x + b
;  Параметры: x – [ebp+8], a – [ebp+16], b – [ebp+24]
; -----
```

df_poly2:

```
    push   ebp
    mov     ebp, esp

    fld     qword [two]        ; 2
    fld     qword [ebp+16]     ; a
```

```

fmulp    st1, st0          ; 2·a
fld      qword [ebp+8]     ; x
fmulp    st1, st0          ; 2·a·x
fadd     qword [ebp+24]    ; +b

pop      ebp
ret

```

```

; -----
;  df_f3(x) = -1 / (x + 2)2
; -----

```

df_f3:

```

push     ebp
mov      ebp, esp

fld      qword [ebp+8]     ; x
fadd     qword [two]       ; x + 2
fld      st0               ; дублирование
fmulp    st1, st0          ; (x+2)2
fld1
fdivrp   st1, st0          ; 1 / (x+2)2
fchs
; отрицание

pop      ebp
ret

```

```

; -----

```



```

; -----
; functions_x86_64.asm (NASM ≥ 2.16, объектный формат ELF-64)
; Аргументы-double → в регистрах xmm0...xmm2, результат-double → xmm0
; -----

        BITS 64
        default rel

; ----- Константы -----
section .rodata
        align 8
a1      dq  0.35
b1      dq -0.95
c1      dq  2.7
b2      dq  3.0
c2      dq  1.0
two     dq  2.0
one     dq  1.0

; ----- Код -----
section .text
        align 16
        global f1, f2, f3, df_poly2, df_f3

; -----
;  $f1(x) = 0.35 \cdot x^2 - 0.95 \cdot x + 2.7$ 
; -----
f1:      ; xmm0 = x
        movapd  xmm1, xmm0      ; xmm1 = x
        mulsd   xmm1, xmm1      ;  $x^2$ 
        mulsd   xmm1, [rel a1]  ;  $0.35 \cdot x^2$ 

        movapd  xmm2, xmm0
        mulsd   xmm2, [rel b1]  ;  $-0.95 \cdot x$ 

        addsd   xmm1, xmm2      ;  $0.35 \cdot x^2 - 0.95 \cdot x$ 
        addsd   xmm1, [rel c1]  ; ... + 2.7
        movapd  xmm0, xmm1      ; → xmm0
        ret

; -----
;  $f2(x) = 3 \cdot x + 1$ 
; -----
f2:

```

```

    mulsd    xmm0, [rel b2]
    addsd    xmm0, [rel c2]
    ret

```

```

; -----
;  f3(x) = 1 / (x + 2)
; -----

```

```

f3:
    addsd    xmm0, [rel two]      ; x+2
    movapd   xmm1, [rel one]
    divsd    xmm1, xmm0
    movapd   xmm0, xmm1
    ret

```

```

; -----
;  df_poly2(x,a,b) = 2·a·x + b
;  (xmm0 = x, xmm1 = a, xmm2 = b)
; -----

```

```

df_poly2:
    mulsd    xmm1, [rel two]      ; 2·a
    mulsd    xmm1, xmm0           ; 2·a·x
    addsd    xmm1, xmm2           ; +b
    movapd   xmm0, xmm1
    ret

```

```

; -----
;  df_f3(x) = -1 / (x + 2)2
; -----

```

```

df_f3:
    addsd    xmm0, [rel two]      ; x+2
    movapd   xmm1, xmm0
    mulsd    xmm1, xmm1           ; (x+2)2
    movapd   xmm2, [rel one]
    divsd    xmm2, xmm1           ; 1 / (x+2)2
    movapd   xmm0, xmm2
    xorpd    xmm1, xmm1
    subsd    xmm1, xmm0           ; -...
    movapd   xmm0, xmm1
    ret

```

Итого

System V AMD64 опирается на богатый регистровый файл и строгие правила выравнивания — ради скорости вызовов и поддержки SIMD.

cdecl/stdcall исторически стек-ориентированы, потому что IA-32 имела мало регистров и не требовала 16-байтовых границ. Разница между ними сводится к тому, **кто** после вызова «заметает» стек. Всё остальное — (сохранение регистров, манглинг, x87-или-SSE) — определяется архитектурой и инструментальной цепочкой времён, когда соглашение появилось.