Архитектура ЭВМ и Язык Ассемблера

Семинар #3:

1. Отладка i386-кода на других архитектурах.
2. Знаковое/беззнаковое переполнение, регистр EFLAGS
3. Сложение 64-битных чисел.
4. Деление/умножение 64-битных чисел.
5. Битовые операции и сдвиги.

09.04.2024

HSE University

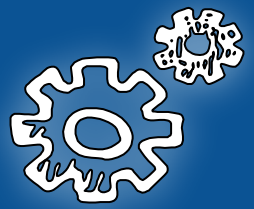
Отладка i386-кода на других архитектурах

Отладка i386 на других архитектурах

1. Предложение от семинариста
   Виртуальная машина с Debian x86_64.
   Гипервизор – UTMv4.4.5 (только для Mac OS).
   Передача файлов – через общую сетевую папку.
2. Предложения от лектора:
   - Локальная виртуальная машина только с SASM.
   - Удалённая виртуальная машина с сетевым доступом.

HSE University

Знаковое/беззнаковое переполнение, регистр EFLAGS



Знаковое/беззнаковое переполнение

Приведите пример 8-битных значений, при сложении которых:

1. Происходит **только беззнаковое** переполнение:

2. Происходит **только знаковое** переполнение.

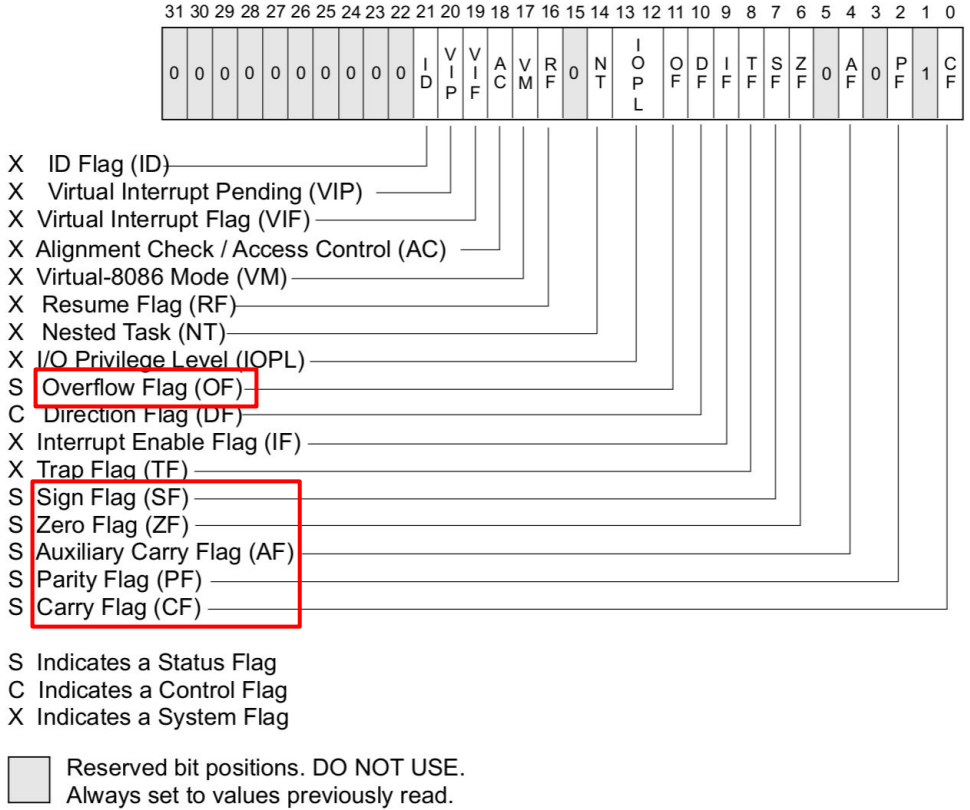
3. Происходит **как знаковое, так и беззнаковое** переполнение.

Знаковое/беззнаковое переполнение

Приведите пример 8-битных значений, при сложении которых:

1. Происходит **только беззнаковое** переполнение:
   `255 + 1 = 0, -1 + 1 = 0`
   `11111111 + 00000001 = 00000000`
2. Происходит **только знаковое** переполнение.
   `127 + 127 = 254, 127 + 127 = -2`
   `01111111 + 01111111 = 11111110`
3. Происходит **как знаковое, так и беззнаковое** переполнение.
   `128 + 128 = 0, -128 + -128 = 0`
   `10000000 + 10000000 = 00000000`

Регистр EFLAGS



Регистр EFLAGS

3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

CF (bit 0)	**Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.
PF (bit 2)	**Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.
AF (bit 4)	**Auxiliary Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
ZF (bit 6)	**Zero flag** — Set if the result is zero; cleared otherwise.
SF (bit 7)	**Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
OF (bit 11)	**Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

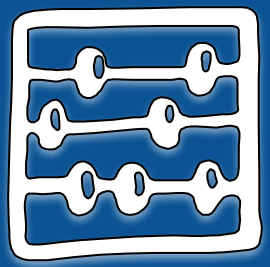
OF = "знаковое переполнение"

CF = "беззнаковое переполнение"

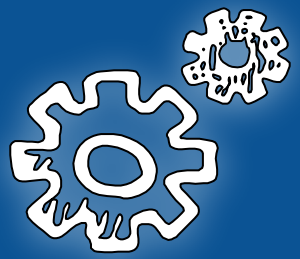
ZF = "результат равен 0"

SF = "результат отрицателен"

Сложение 64-битных чисел



Деление/умножение 64-битных чисел



Операции с 32-битными числами

```
// Unsigned 32-bit multiplication:
static uint32_t u32_a;
static uint32_t u32_b;
static uint32_t u32_c;
static uint32_t u32_d;

u32_c = u32_a * u32_b;
u32_d = u32_a / u32_b;

// Signed 32-bit multiplication:
static int32_t s32_a;
static int32_t s32_b;
static int32_t s32_c;
static int32_t s32_d;

s32_c = s32_a * s32_b;
s32_d = s32_a / s32_b;
```

Операции с 32-битными числами

```
call    __x86.get_pc_thunk.bx ; sym.__x86.get_pc_thunk.bx
add     ebx, 0x2e08
mov     edx, dword [ebx + 0x70] ; benchmark.c:27 u32_c = u32_a * u32_b;
mov     eax, dword [ebx + 0x74]
imul    eax, edx
mov     dword [ebx + 0x78], eax
mov     eax, dword [ebx + 0x70] ; benchmark.c:28 u32_d = u32_a / u32_b;
mov     edi, dword [ebx + 0x74]
mov     edx, 0
div     edi
mov     dword [ebx + 0x7c], eax
mov     edx, dword [ebx + 0x80] ; benchmark.c:36 s32_c = s32_a * s32_b;
mov     eax, dword [ebx + 0x84]
imul    eax, edx
mov     dword [ebx + 0x88], eax
mov     eax, dword [ebx + 0x80] ; benchmark.c:37 s32_d = s32_a / s32_b;
mov     edi, dword [ebx + 0x84]
cdq
idiv    edi
mov     dword [ebx + 0x8c], eax
```

Знаковое расширение

```
mov     eax, dword [ebx + 0x80] ; benchmark.c:37 s32_d = s32_a / s32_b;
mov     edi, dword [ebx + 0x84]
cdq
idiv    edi
mov     dword [ebx + 0x8c], eax
```

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
99	CWD	ZO	Valid	Valid	DX:AX := sign-extend of AX.
99	CDQ	ZO	Valid	Valid	EDX:EAX := sign-extend of EAX.
REX.W + 99	CQO	ZO	Valid	N.E.	RDX:RAX:= sign-extend of RAX.

Операции с 64-битными числами

```
// Unsigned 64-bit multiplication:
static uint64_t u64_a;
static uint64_t u64_b;
static uint64_t u64_c;
static uint64_t u64_d;

u64_c = u64_a * u64_b;
u64_d = u64_a / u64_b;

// Signed 64-bit multiplication:
static int64_t s64_a;
static int64_t s64_b;
static int64_t s64_c;
static int64_t s64_d;

s64_c = s64_a * s64_b;
s64_d = s64_a / s64_b;
```

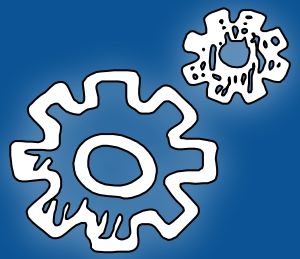
64-битное умножение

```
mov     esi, dword [ebx + 0x90] ; benchmark.c:45 u64_c = u64_a * u64_b;
mov     edi, dword [ebx + 0x94]
mov     eax, dword [ebx + 0x98]
mov     edx, dword [ebx + 0x9c]
mov     ecx, edi
imul    ecx, eax
mov     dword [var_1ch], ecx
mov     ecx, edx
imul    ecx, esi
add     ecx, dword [var_1ch]
mul     esi
add     ecx, edx
mov     edx, ecx
mov     dword [ebx + 0xa0], eax
mov     dword [ebx + 0xa4], edx
```

64-битное деление

```
mov     eax, dword [ebx + 0x90] ; benchmark.c:46 u64_d = u64_a / u64_b;
mov     edx, dword [ebx + 0x94]
mov     esi, dword [ebx + 0x98]
mov     edi, dword [ebx + 0x9c]
push    edi
push    esi             ; int32_t arg_34h
push    edx             ; int32_t arg_30h
push    eax             ; int32_t arg_3ch
call    __udivdi3       ; sym.__udivdi3
add     esp, 0x10
mov     dword [ebx + 0xa8], eax
mov     dword [ebx + 0xac], edx
```

Битовые операции и сдвиги



Битовые операции и сдвиги

```
int a [4];     unsigned int c [4];     unsigned char q [4];
```

Реализовать на ассемблере:

1. `a[0] = q[0] | (q[1] << 8) | (q[2] << 16)`
2. `c[0] = (c[0] << q[0]) | (c[0] >> (32 - q[0]))`
3. `a[0] = a[0] < 0 ? 0xffffffff : 0`

Вопросы?

Красивые иконки взяты с сайта handdrawngoods.com