Clever Compilers: x86_64 assembly arithmetic analysis

junyu33

Introduction

This is a brief overview about how x86_64 C compilers (e.g. `gcc`) optimize arithmetic calculations. I'll try to give analysis for simple code blocks for different operations like:

- addition
- subtraction
- multiplication
- division
- modulo

In this way we can have some kind of intuition about how smart the compilers are.

Assuptions

Before we start, let's have some assumptions. For example, the time consumption of clock cycles in different instructions of x86_64[1]:

INST	ADD/SUB	MUL	IMUL	DIV	IDIV	SHR SHL SAR	ROR/ROL
r,r/i	1	2	1/2[2]	36	57	1	1
r,m	1	2	1/2				
m,r/i	2					3	4

and we don't consider other optimizations in CPU, like pipelining and out-of-order execution.

---

1. https://www.agner.org/optimize/instruction_tables.pdf, using `Skylake` architecture. ↩️
2. If we have 3 opreands, like `imul rcx, rsi, 10`, the result is 1. ↩️

Addition

```c
// gcc test.c -o test
#include <stdio.h>
int main() {
    int a = 1;
    int b = 2;
    int c = a + b;
    printf("%d", c);
    return 0;
}
```

```
0000000000001139 <main>:
    1139: 55                      push   rbp
    113a: 48 89 e5                mov    rbp,rsp
    113d: 48 83 ec 10             sub    rsp,0x10
    1141: c7 45 f4 01 00 00 00    mov    DWORD PTR [rbp-0xc],0x1
    1148: c7 45 f8 02 00 00 00    mov    DWORD PTR [rbp-0x8],0x2
    114f: 8b 55 f4                mov    edx,DWORD PTR [rbp-0xc]
    1152: 8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
    1155: 01 d0                   add    eax,edx
    1157: 89 45 fc                mov    DWORD PTR [rbp-0x4],eax
    115a: 8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
    115d: 89 c6                   mov    esi,eax
# other code about printf
```

What about O2 optimization?

```c
// gcc test.c -o test -O2
#include <stdio.h>
int main() {
    int a = 1;
    int b = 2;
    int c = a + b;
    printf("%d", c);
    return 0;
}
```

```
0000000000001040 <main>:
    1040: 48 83 ec 08             sub    rsp,0x8
    1044: be 03 00 00 00          mov    esi,0x3                 # 1 + 2 = 3
    1049: 48 8d 3d b4 0f 00 00    lea    rdi,[rip+0xfb4]         # 2004 <_IO_stdin_used+0x4>
    1050: 31 c0                   xor    eax,eax
    1052: e8 d9 ff ff ff          call   1030 <printf@plt>
    1057: 31 c0                   xor    eax,eax
    1059: 48 83 c4 08             add    rsp,0x8
    105d: c3                      ret
    105e: 66 90                   xchg   ax,ax
```

This techique is called `constant folding`.

Another example, let's try to encapsulate `add` into a function:

```c
// gcc test.c -o test
#include <stdio.h>
int add(int a, int b) {return a + b;}
int main() {
    int c = add(1, 2);
    printf("%d", c);
    return 0;
}
```

```
0000000000001139 <add>:
    1139: 55                    push   rbp
    113a: 48 89 e5              mov    rbp,rsp
    113d: 89 7d fc              mov    DWORD PTR [rbp-0x4],edi
    1140: 89 75 f8              mov    DWORD PTR [rbp-0x8],esi
    1143: 8b 55 fc              mov    edx,DWORD PTR [rbp-0x4]
    1146: 8b 45 f8              mov    eax,DWORD PTR [rbp-0x8]
    1149: 01 d0                 add    eax,edx
    114b: 5d                    pop    rbp
    114c: c3                    ret
```

A very classical `__cdecl` routine.

Let's try O2 again, here is the result:

```
0000000000001160 <add>:
    1160: 8d 04 37              lea    eax,[rdi+rsi*1]
    1163: c3                    ret
```

As you can see, compilers often use `lea` to optimize additions. Let's try to analyze the performance:

If we use classical `ADD` instruction, we need:

```
add rdi, rsi
mov eax, rdi
```

which costs 2 clock cycles, however:

```
lea    eax,[rdi+rsi*1]
```

only costs 1 clock cycle, so `LEA` way is better.

This optimization is also very common in mutiplication.

Subtraction

We all know that subtracting a number is the same as adding its 2’s complement. So it is almost the same:

```c
// gcc test.c -o test -O2
#include <stdio.h>
int sub(int a, int b) {return a - b;}
int main() {
    int c = sub(1, 2);
    printf("%d", c);
    return 0;
}
```

```
0000000000001160 <sub>:
    1160: 89 f8                 mov    eax,edi
    1162: 29 f0                 sub    eax,esi
    1164: c3
```

As you can see, we don’t have such instructions like `lea eax,[rdi-rsi*1]`, because it is inherently not supported in x86 assembly. I believe this is the only difference.

Multiplication

```c
// gcc test.c -o test -O2
#include <stdio.h>
int mul0() {return 3 * 5;}
int mul1(int x) {return 7 * x;}
int mul2(int x) {return x * 8 + 5;}
int mul3(int x) {return x * 113 + 5;}
int mul4(int x) {return x * 1024 + 5;}
int mul5(int x, int y) {return x * y;}
int main() {
    int a = mul0(), b = mul1(3), c = mul2(3), d = mul3(3), e = mul4(3), f = mul5(3, 5);
    printf("%d %d %d %d %d %d\n", a, b, c, d, e, f);
    return 0;
}
```

And here is the disassembly (ignored some useless code):

```
0000000000001180 <mul0>:
    1180: b8 0f 00 00 00          mov    eax,0xf
    1185: c3                      ret
```

```
0000000000001190 <mul1>:
    1190: 8d 04 fd 00 00 00 00  lea    eax,[rdi*8+0x0]
    1197: 29 f8                 sub    eax,edi
    1199: c3                    ret

00000000000011a0 <mul2>:
    11a0: 8d 04 fd 05 00 00 00  lea    eax,[rdi*8+0x5]
    11a7: c3                    ret

00000000000011b0 <mul3>:
    11b0: 6b c7 71              imul   eax,edi,0x71
    11b3: 83 c0 05              add    eax,0x5
    11b6: c3                    ret

00000000000011c0 <mul4>:
    11c0: c1 e7 0a              shl    edi,0xa
    11c3: 8d 47 05              lea    eax,[rdi+0x5]
    11c6: c3                    ret

00000000000011d0 <mul5>:
    11d0: 89 f8                 mov    eax,edi
    11d2: 0f af c6              imul   eax,esi
    11d5: c3                    ret
```

We can conclude some features below:

- `constant folding` still exists.
- the sequence of two multipliers doesn't matter.
- for small numbers (especially closes to the power of 2), `LEA` instruction is mostly used.
- If `LEA` is used and we don't need to subtract something, we can also **add** a constant in the same `LEA` instruction.
- If the number is large and closes to the power of 2, `SHL` instrcution will be used, otherwise uses `IMUL` finally.

Wait! But `IMUL` only costs 1 clock cycle if we use something like `imul eax,edi,<arg>`, compilers don't need to do such conversion!

In modern CPU architectures like `Skylake`, this is correct. But there are also older CPUs like `Pentium`, `MUL/IMUL` instruction can cost at most **9** clock cycles. However `SHL/LEA` always costs **1** clock cycle. So it is more "secure" for compilers to have such conversion in order to reach better efficiency.

Signed Division

From the previous content, we know signed division costs 57 clock cycles, so compilers will avoid using `IDIV` command, especially if the divisor is a constant. Here are 4 possible cases:

- divisor is the power of 2
- divisor is the power of 2, but negative
- divisor is not the power of 2
- divisor is not the power of 2, and negative

The unsigned division is similar, and I won't expand it again.

divisor is the power of 2

```c
// gcc test.c -o test -O2
#include <stdio.h>
int div32(int x) {
    return x / 32;
}
int main() {
    int a = 0x12345678;
    int b = div32(a);
    printf("0x%x\n", b);
    return 0;
}
```

```
0000000000001160 <div32>:
    1160: 85 ff                test   edi,edi          # set SF flag
    1162: 8d 47 1f             lea    eax,[rdi+0x1f]   # if it is negative, add 31
    1165: 0f 49 c7             cmovns eax,edi          # conditional move (if SF=0, then eax=edi)
    1168: c1 f8 05             sar    eax,0x5          # eax >>= 5
    116b: c3                   ret
```

If the dividend is positive and 0, obviously it is correct. Otherwise we have,

$$\frac{x}{2^n} = (x + (2^n - 1)) \gg n$$

divisor is the power of 2, but negative

```
// gcc test.c -o test -O2
#include <stdio.h>
int div32(int x) {
    return x / -32;
}
int main() {
    int a = 0x12345678;
    int b = div32(a);
    printf("0x%x\n", b);
    return 0;
}
```

```
0000000000001160 <div32>:
    1160: 85 ff                     test   edi,edi
    1162: 8d 47 1f                  lea    eax,[rdi+0x1f]
    1165: 0f 49 c7                  cmovns eax,edi
    1168: c1 f8 05                  sar    eax,0x5
    116b: f7 d8                     neg    eax
    116d: c3                        ret
```

The only difference is negative the result.

divisor is not the power of 2

```c
// gcc test.c -o test -O2
#include <stdio.h>
int div32(int x) {
    return x / 53;
}
int main() {
    int a = 0x12345678;
    int b = div32(a);
    printf("0x%x\n", b);
    return 0;
}
```

```
0000000000001160 <div32>:
    1160: 48 63 c7              movsxd rax,edi                      # sign-extension
    1163: c1 ff 1f              sar    edi,0x1f                     # check the sign-bit
    1166: 48 69 c0 ed 73 48 4d  imul   rax,rax,0x4d4873ed
    116d: 48 c1 f8 24           sar    rax,0x24                     # signed right-shift 36 bit
    1171: 29 f8                 sub    eax,edi                      # if negative, minus -1
    1173: c3                    ret
```

So, what the HELL is `0x4d4873ed` ???

- それは~~魔法の数字~~「マジックナンバー」です (It's a magic number)

Let's do a little calculation (assume c is the magic number):

$$\frac{x}{53} = x \times c \gg 36$$

so we have:

$$\frac{2^{36}}{53} = c$$

After a little calculation, we know the answer:

```
Python 3.11.8 (main, Feb 12 2024, 14:50:05) [GCC 13.2.1 20230801] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> pow(2,36) / 53
1296593900.6792452
>>> hex(int(pow(2,36) / 53))
'0x4d4873ec'
>>> hex(round(pow(2,36) / 53))
'0x4d4873ed'
>>>
```

That's how the magic number originates.

Let's summarize a little, suppose the dividend is o, we have:

$$\frac{x}{o} = (x \times c \gg N) + x[31]$$

where $x[31]$ is the sign-bit of x, and $o = \frac{2^N}{c}$.

The reason why adding $x[31]$ is the same as the previous $2^n - 1$: Rounding towards zero.

divisor is not the power of 2, and negative

```
// gcc test.c -o test -O2
#include <stdio.h>
int div32(int x) {
    return x / -53;
}
int main() {
    int a = 0x12345678;
    int b = div32(a);
    printf("0x%x\n", b);
    return 0;
}
```

```
0000000000001160 <div32>:
    1160: 48 63 d7              movsxd rdx,edi
    1163: c1 ff 1f              sar    edi,0x1f
    1166: 48 69 d2 ed 73 48 4d  imul   rdx,rdx,0x4d4873ed
    116d: 89 f8                 mov    eax,edi
    116f: 48 c1 fa 24           sar    rdx,0x24
    1173: 29 d0                 sub    eax,edx
    1175: c3                    ret
```

Just exchanged the minuend and subtrahend to omit `NEG` instruction.

Modulo

When the dividend is the power of 2:

```c
// gcc test.c -o test -O2
#include <stdio.h>
int mod32(int x) {
    return x % 32;
}
int main() {
    int a = 0x12345678;
    int b = mod32(a);
    printf("0x%x\n", b);
    return 0;
}
```

```
0000000000001160 <mod32>:
    1160: 89 fa                 mov    edx,edi
    1162: c1 fa 1f              sar    edx,0x1f
    1165: c1 ea 1b              shr    edx,0x1b              # if edi is negative, edx will equal to 31
    1168: 8d 04 17              lea    eax,[rdi+rdx*1]       # add 31 or 0
    116b: 83 e0 1f              and    eax,0x1f
    116e: 29 d0                 sub    eax,edx               # cancel the add operation
    1170: c3                    ret
```

From the assembly above, we can have this formula:

$$x\%2^n = \begin{cases} x\&(2^n-1) & (x \geq 0) \\ ((x+(2^n-1))\&(2^n-1))-(2^n-1) & (x < 0) \end{cases}$$

The proof will be left for exercise after this talk.

Another situation is that the dividend is not the power of 2:

```
// gcc test.c -o test -O2
#include <stdio.h>
int mod32(int x) {
    return x % 53;
}
int main() {
    int a = 0x12345678;
    int b = mod32(a);
    printf("0x%x\n", b);
    return 0;
}
```

```
0000000000001160 <mod32>:
    1160: 48 63 c7              movsxd rax,edi
    1163: 89 fa                 mov    edx,edi
    1165: 48 69 c0 ed 73 48 4d  imul   rax,rax,0x4d4873ed
    116c: c1 fa 1f              sar    edx,0x1f
    116f: 48 c1 f8 24           sar    rax,0x24
    1173: 29 d0                 sub    eax,edx
    1175: 6b d0 35              imul   edx,eax,0x35
    1178: 89 f8                 mov    eax,edi
    117a: 29 d0                 sub    eax,edx
    117c: c3                    ret
```

```
0000000000001160 <mod32>:
    1160: 48 63 c7              movsxd rax,edi
    1163: 89 fa                 mov    edx,edi
    1165: 48 69 c0 ed 73 48 4d  imul   rax,rax,0x4d4873ed
    116c: c1 fa 1f              sar    edx,0x1f
    116f: 48 c1 f8 24           sar    rax,0x24
    1173: 29 d0                 sub    eax,edx
    1175: 6b d0 35              imul   edx,eax,0x35
    1178: 89 f8                 mov    eax,edi
    117a: 29 d0                 sub    eax,edx
    117c: c3                    ret
```

Guess what does it mean?

Answer:

$$a\%b = a - (a/b) * b$$

And please mention that, because the division is rounding towards zero, the definition is not the same as that in maths. For example:

$-5\%3 = -2$ in this code, while 1 in maths.

Summary

In this talk, we have learned:

- constant folding
- `LEA` to substitude addition and multiplication
- `SHL` to substitude large number multiplication with the power of 2
- magic numbers
- use division or `AND` to substitute modulo

Hope all of listeners can have a rough understanding of the compilers' intelligence.

References

- https://www.agner.org/optimize/instruction_tables.pdf
- 《加密与解密》（第四版）
- https://ja.wikipedia.org/wiki/マジックナンバー_(プログラム)
- https://tieba.baidu.com/p/3786445337