

Compilers are smart

Matt Godbolt - Advent of Compiler Optimization 2025

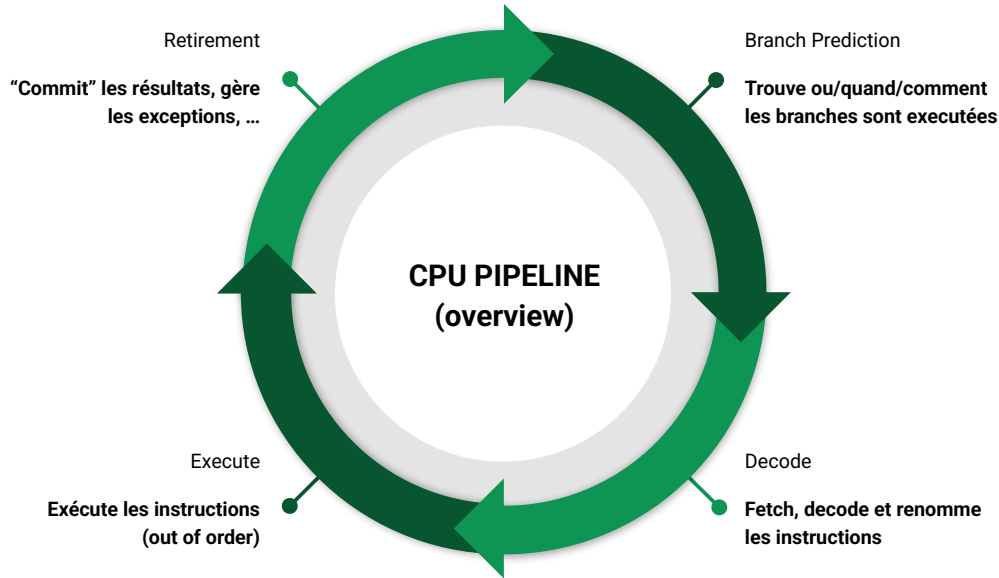
Bastien DOIGNIES - GDL LIRIS

Compilers

Un compilateur est un outils qui traduit du code écrit par des humains de façon à être exécuté par l'ordinateur.



CPU models



Memory latency (3Ghz)		
register:	0.25 ns	(1/4 cycle)
L1 cache:	1 ns	(3 cycles)
1m at ls:	3.3 ns	
L2 cache:	4 ns	(12 cycles)
L3 cache:	12 ns	(38 cycles)
DRAM :	65 ns	(200 cycles)
NVME :	100 us	(50000 cycles)
SATA SSD:	500 us	(500000 cycles)
HDD :	15000 us	(50000000 cycles)

Assembly

L'assembleur est un langage très bas niveau dans lequel on écrit le code très proche de ce que perçoit le CPU. On est proche du “une ligne = une instruction”. Il est facile de traduire ce langage en pure binaire directement exécuté par le CPU.

L'assembleur dépend de l'architecture du processeur.



```
mov eax, 5 ; eax = 5
add eax, 3 ; eax = eax + 3
mov eax, [ebx] ; eax = value at ebx
label:
    dec eax ; eax = eax - 1
    cmp eax, 0 ; compare
    jne label ; jump if not equal
```



```
MOV r0, #5 ; r0 = 5
ADD r0, r0, #3 ; r0 = r0 + 3
LDR r0, [r1] ; r0 = value at r1
loop:
    SUBS r0, r0, #1 ; r0 = r0 - 1
    CMP r0, #0 ; compare to 0
    BNE loop ; branch if not equal
```

xor eax, eax

c++



```
1  int test()  
2  {  
3      return 0;  
4  }
```

01



```
1  test():  
2      mov  eax, 0  
3      ret
```

xor eax, eax

c++



```
1  int test()  
2  {  
3      return 0;  
4  }
```

01



```
1  test():  
2      mov eax, 0  
3      ret
```

02



```
1  test():  
2      xor eax, eax  
3      ret
```

Xor eax, eax produit un programme plus court, car il est attendu que le nombre de droite soit sur 4 octets, alors qu'un registre est encodé avec un seul. Aussi, le CPU sait que le résultat est 0 et n'exécute pas vraiment l'opération.

lea (load effective address)

C++



```
1  int add(int x, int y)
2  {
3      return x + y;
4  }
```

lea (load effective address)

C++

```
1  int add(int x, int y)
2  {
3      return x + y;
4  }
```

O2

```
1  add(int,int):
2      lea eax, [rdi+rsi]
3      ret
```

Ici, le compilateur choisit l'instruction "lea" plutôt qu'une addition ; qui est normalement utilisée pour l'adressage dans les tableaux.

Le problème de l'addition en x86, est que c'est un "+=", et donc on ne choisit pas la destination. Ce problème n'existe pas pour ARM.

Hidden additions

```
1 unsigned add_v3(unsigned x, unsigned y)
2 {
3     while(x--) y++;
4     return y;
5 }
6
7 unsigned add_v4(unsigned x, unsigned y)
8 {
9     if (y == 0) return x;
10    return add_v4(x + 1, y - 1);
11 }
```

```
add_v3(unsigned, unsigned):
    ADD w0, w1, w0
    RET

add_v4(unsigned, unsigned):
    ADD w0, w1, w0
    RET
```

(Note: en O1, le compilateur génère une boucle pour add_v4)

Multiplications par une constante



```
1  int mult(int x)
2  {
3      return x * 2;
4  }
```

Multiplications par une constante



```
1  int mult(int x)
2  {
3      return x * 2;
4  }
```



```
1  int mult(int x)
2  {
3      return (x << 1);
4  }
```

Multiplications par une constante

```
1  int mult(int x)
2  {
3      return x * 2;
4  }
```

```
1  int mult(int x)
2  {
3      return (x << 1);
4  }
```

```
1  mult(int):
2      lea eax, [rdi+rdi]
3      ret
```

Pas d'explications totale sur le pourquoi lea plutôt qu'un shift. Potentiellement plus rapide pour les architectures modernes.

Cf: https://www.agner.org/optimize/instruction_tables.pdf

Multiplications par une constante



```
1  int mult(int x)
2  {
3      return x * 9;
4  }
```



```
1  int mult(int x)
2  {
3      return x * 25;
4  }
```



```
1  mult(int):
2      lea eax, [rdi+rdi*8]
3      ret
```



```
1  mult(int):
2      lea eax, [rdi+rdi*4]
3      lea eax, [rax+rax*4]
4      ret
```

Multiplications par une constante



```
1  int mult(int x)
2  {
3      // return x * 522;
4      return
5          (x << 9) +
6          (x << 3) +
7          (x << 1);
8  }
```



```
1  mult(int):
2      imul eax, edi, 522
3      ret
```

Division par une constante



```
1  int div(int x)
2  {
3      return div / 512;
4  }
```

Division par une constante



```
1  int div(int x)
2  {
3      return div / 512;
4  }
```

On doit arrondir vers 0, alors que le shift arrondit les négatifs vers -oo.



```
1  div(int):                ; edi = x
2      test edi, edi
3      lea eax, [rdi+511]    ; eax = x + 511
4      cmovns eax, edi       ; if (x >= 0) eax = x
5      sar eax, 9            ; eax = (eax << 9)
6      ret
```

Division par une constante

```
1 unsigned div(unsigned x)
2 {
3     return x / 10;
4 }
```

GCC a une table pour les division jusqu'à 12345 environ.

$$x = x \frac{2^{35}}{10} \frac{1}{2^{35}}$$

```
1 div(int):                ; edi = x
2     mov eax, edi
3     mov edi, 3435973837 ; ~= 2^35 / 10
4     imul rax, rdi        ; eax *= 2^35 / 10
5     shr rax, 35          ; eax /= 2^35
6     ret
```

“sWitchCraft”



```
int func(int x)
{
    switch (x)
    {
        case 1: return 100;
        case 2: return 200;
        case 3: return 300;
        case 4: return 400;
        default:
            return 0;
    }
}
```



```
func(int):
    imul ecx, edi, 100
    dec edi
    xor eax, eax
    cmp edi, 4
    cmovb eac, ecx
    ret
```

“sWitchCraft”



```
int func(int x)
{
    switch (x)
    {
        case 1: return 1;
        case 2: return 5284;
        case 3: return 2511;
        case 4: return 483;
    }
    return 0;
}
```



```
func(unsigned int):
    dec edi
    xor eax, eax
    cmp edi, 3
    ja .LBB0_2
    mov eax, edi
    lea rcx, [rip + .Lswitch.table.func(unsigned)]
    mov eax, dword ptr [rcx + 4 * rax]
.LBB0_2:
    ret

.Lswitch.table.func(unsigned):
    .long 1
    .long 5284
    .long 2511
    .long 483
```

“sWitchCraft”



```
bool isspace(char x)
{
    switch (x)
    {
        case ' ':
        case '\n':
        case '\t':
        case '\r':
        case '\v':
            return true;
    }
    return false;
}
```



```
isspace(char):
    sub edi, 9 ; ' ' = 32, '\t' = 9
    mov eax, 8388631 ; 10000000000000000000000010111b
    bt rax, rdi ; Test if 'x-9'th bit is set
    setc al ; Cleanup and check bounds
    xor edx, edx
    cmp dil, 24
    cmovnb eax, edx
    ret
```

“sWitchCraft”



```
unsigned func(unsigned x)
{
    switch (x)
    {
        case 1 : return 1;
        case 5284: return 2;
        case 2511: return 3;
        case 483 : return 4;
        case 999 : return 5;
    }
    return 0;
}
```



```
func(unsigned):
    cmp edi, 2511
    je .L4 ; returns 3
    jbe .L11
    mov eax, 2
    cmp edi, 5284
    je .L1
    xor eax, eax
    cmp edi, 9999
    sete al ; al = 0 or 1 (cmp)
    lea eax, [rax + rax * 4] ; eax = al + al * 5
    ret ; = 0 or 5 depending on cmp

.L11:
    mov eax, 1
    cmp edi, 1
    je .L1 ; returns 1
    xor eax, eax
    cmp edi, 483
    sete al ; al = 0 or 1
    sal eax, 2 ; shift left => al = 0 or 4
    ret ;
```

Partial inlining



```
unsigned calculate(unsigned x) {  
    if (x < 100) return x * 2;  
    // Beaucoup de calculs compliqué plus tard  
    return result;  
}
```

```
unsigned calculate_more(unsigned x) {  
    return calculate(x) + calculate(x + 1);  
}
```



```
calculate_more(unsigned):  
    mov esi, edi  
    cmp edi, 99  
    jbe .L13  
    call calculate(unsigned)(.part.0)  
    lea edi, [rsi+1]  
    mov r8d, eax  
    cmp edi, 99  
    jbe .L14  
.L11:  
    call calculate(unsigned)(.part.0)  
    add eax, r8d  
    ret  
.L13:  
    lea r8d, [rdi+rdi]  
    lea edi, [rsi+1]  
    cmp edi, 99  
    ja .L11  
.L14:  
    lea eax, [rdi+rdi]  
    add eax, r8d  
    ret
```

Branch optimization



```
for (auto integer : lists_of_ints)
{
    total_sum += integer;
    if (integer < 128) sum_128 += integer;
}
```



```
.L4:
    mov ecx, [rdx] ; integer
    add r8d, ecx   ; total_sum += integer
    lea r9d, [rsi + rcx] ; tmp = sum_128 + integer
    cmp ecx, 128   ; integer < 128 ?
    cmovb esi, r9d ; sum_128 = tmp if yes
    add rdx, 4     ; Advance and loop
    cmp r10, rdx
    jne .L4
```

Trucs divers

Les compilateurs préfèrent:

- Les '!=' que les '<' dans les boucles
- Les lambdas
- Les 'for range loops'
- Ternaries aux ifs

Ils font aussi:

- De la vectorisation
- Des "tail call optimizations" (parfois il faut un attribut)
- Du réordonnement d'instructions (modulos & div !!!)

clang ScalarEvolution <3

<https://github.com/llvm/llvm-project/blob/llvmorg-21.1.8/llvm/lib/Analysis/ScalarEvolution.cpp>

```
long sum(long n)
{
    long total = 0;
    for (long i = 0; i < n; ++i)
        total += i;
    return total;
}
```

```
sum(long):
    test edi, edi
    jle .LBB0_1
    lea eax, [rdi - 1] ; eax = (n - 1)
    lea ecx, [rdi - 2] ; ecx = (n - 2)
    imul rcx, rax      ; rcx = (n - 1)(n - 2)
    shr rcx ; rcx = (n - 1)(n - 2) / 2
    ; eax = n + (n-1)(n-2)/2 - 1 = n(n-1)/2
    lea eax, [rdi + rcx]
    dec eax
    ret
.LBB0_1:
    xor eax, eax
    ret
```

BUT ... ALIASING

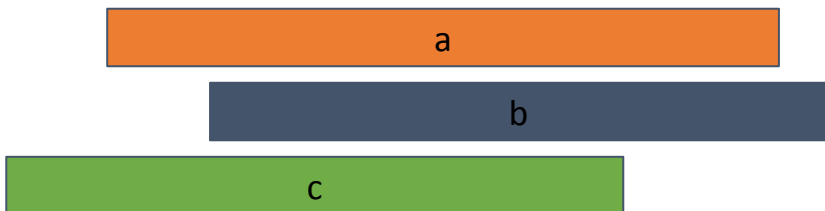
A.K.A

**Code is not always as simple as
you may think.**

Aliasing

```
void add(int* a, int* b, int* c, int n) {  
    for (int i = 0; i < n; ++i) {  
        a[i] = b[i] + c[i];  
    }  
}
```

a, b et c peuvent se “couvrir” dans la mémoire



```
add(int*, int*, int*, int):  
    mov     r8, rdx  
    test    ecx, ecx  
    jle     .L1  
    mov     ecx, ecx  
    xor     eax, eax  
    sal     rcx, 2  
  
.L3:  
    mov     edx, DWORD PTR [r8+rax]  
    add     edx, DWORD PTR [rsi+rax]  
    mov     DWORD PTR [rdi+rax], edx  
    add     rax, 4  
    cmp     rcx, rax  
    jne     .L3  
  
.L1:  
    ret
```

Aliasing

```
void add(int* __restrict a, int* __restrict b, int* __restrict c, int n) {  
    for (int i = 0; i < n; ++i) {  
        a[i] = b[i] + c[i];  
    }  
}
```

```
"add(int*, int*, int*, int)":  
    mov     r8, rdx  
    test    ecx, ecx  
    jle     .L1  
    lea     eax, [rcx-1]  
    cmp     eax, 2  
    jbe     .L6  
    mov     r9d, ecx  
    xor     eax, eax  
    shr     r9d, 2  
    mov     edx, r9d  
    sal     rdx, 4  
  
.L4:  
    movdqu  xmm0, XMMWORD PTR [rsi+rax]  
    movdqu  xmm2, XMMWORD PTR [r8+rax]  
    paddb   xmm0, xmm2  
    movups  XMMWORD PTR [rdi+rax], xmm0  
    add     rax, 16  
    cmp     rax, rdx  
    jne     .L4  
    lea     eax, [0+r9*4]  
    mov     edx, eax  
    cmp     ecx, eax  
    je      .L1  
  
.L3:  
    sub     ecx, eax  
    cmp     ecx, 1  
    je      .L5  
    movq    xmm0, QWORD PTR [rsi+rax*4]  
    movq    xmm1, QWORD PTR [r8+rax*4]  
    paddq   xmm0, xmm1  
    movq    QWORD PTR [rdi+rax*4], xmm0  
    mov     eax, ecx  
    and     eax, -2  
    and     ecx, 1  
    je      .L1  
    add     edx, eax  
  
.L5:  
    mov     eax, DWORD PTR [r8+rdx*4]  
    add     eax, DWORD PTR [rsi+rdx*4]  
    mov     DWORD PTR [rdi+rdx*4], eax  
  
.L1:  
    ret
```

Aliasing is ... surprising

```
template<typename Total>
struct Counter
{
    Total total{};
    void count(std::span<const int> span) {
        for (auto elem : span)
            total += elem;
    }
};

template class Counter<int>;
template class Counter<long>;
```

Dans le cas 'int', il peut y avoir aliasing. Il est possible, que le span pointe (partiellement) vers la variable total. Ainsi, le compilateur est obligé de recharger le compteur dans la mémoire.

Dans le cas long, il n'a qu'un seul registre pour compter.

FIN