

Confronto compilatori

1 Introduzione

In questo documento vengono confrontati dei compilatori. I Compilatori confrontati sono *gcc RISC-V*, *gcc ARM* e *gcc x86_64*. I compilatori RISC-V e ARM vengono confrontati per il loro focus sul ambiente embedded ed entrambi sono architetture RISC. Tutti i codici presentati in questa sezione vengono compilati e presentati con il livello di ottimizzazione di default (-O0).

– Add more Intro info

2 Operazioni

2.1 Addizione con costante

```
1 | int get_num(int num) {  
2 |     return 23 + num;  
3 | }
```

La funzione è una semplice funzione scritta in C che dato un numero di tipo intero restituisce il numero sommato a 23.

RISC-V

Il sorgente compilato con il compilatore RISC-V(a) da riga 2 fino a riga 6 predispone la chiamata della procedura posizionando sullo stack il necessario, da riga 7 inizia la funzione. Su quella riga viene recuperato il valore di num che alla riga 8, tramite l'operazione di add immediate, viene sommato a num. Il risultato dell'operazione addiw è la somma del valore di num sommato alla costante 23, il risultato è esteso su 64 bit, vengono ignorati gli errori di overflow. Successivamente tramite la pseudo istruzione sext.w che prende i 32 bit inferiori e li memorizza nel registro rd. Questa istruzione corrisponde a addiw rd, rs, 1 0. Il risultato viene spostato nel registro a0 che, nei processori

```

1 get_num:
2     addi    sp,sp,-32
3     sd      s0,24(sp)
4     addi    s0,sp,32
5     mv      a5,a0
6     sw      a5,-20(s0)
7     lw      a5,-20(s0)
8     addiw   a5,a5,23
9     sext.w  a5,a5
10    mv      a0,a5
11    ld      s0,24(sp)
12    addi    sp,sp,32
13    jr      ra

```

(a) RISC-V

```

1 get_num:
2     push    {r7}
3     sub     sp, sp, #12
4     add     r7, sp, #0
5     str     r0, [r7, #4]
6     ldr     r3, [r7, #4]
7     adds   r3, r3, #23
8     mov     r0, r3
9     adds   r7, r7, #12
10    mov     sp, r7
11    ldr     r7, [sp], #4
12    bx      lr

```

(b) ARM

```

1 get_num:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp
5     -4], edi
6     mov     eax, DWORD PTR
7     [rbp-4]
8     add     eax, 23
9     pop     rbp
10    ret

```

(c) x86

Figure 1: Funzione di somma

RISC-V, viene utilizzato come restituzione di risultato. Le righe successive ripristinano lo stack e restituisce il controllo al chiamante.

ARM

Il sorgente (c), compilato con gcc ARM, mostra che la preparazione della procedura si esegue da riga 2 a riga 5, le con le due righe successive si esegue la funzione. La riga 6 recupera il valore di num la riga successiva calcola il valore del risultato e, infine, alla riga 8 si sposta il risultato nel registro di restituzione

x86

Il sorgente (b) è compilato con gcc di x86. Da riga 2 fino a riga 4 viene preparato lo stack, a riga 5 viene posizionato num nel registro eax che a riga 6 viene sommato a 23 che viene memorizzato nel registro eax. Infine viene ridato il controllo al chiamante.

2.2 Addizione

Nel caso generale viene calcolata la somma di 3 numeri.

```

1 int sumGen(int num, int num2, int num3) {
2     return num + num2 + num3 ;
3 }

```

Inserisci sorgenti Addizione generale

RISC-V

Nel caso del compilatore RISC-V la somma avviene tra 13 e 19 dove gli operandi vengono caricati nei registri a4 e a5, successivamente calcolato il risultato e memorizzato in a4 che poi verrà sommato con l'ultimo operando,

<pre> 1 sumGen: 2 addi sp,sp,-32 3 sd s0,24(sp) 4 addi s0,sp,32 5 mv a5,a0 6 mv a3,a1 7 mv a4,a2 8 sw a5,-20(s0) 9 mv a5,a3 10 sw a5,-24(s0) 11 mv a5,a4 12 sw a5,-28(s0) 13 lw a4,-20(s0) 14 lw a5,-24(s0) 15 addw a5,a4,a5 16 sext.w a5,a5 17 lw a4,-28(s0) 18 addw a5,a4,a5 19 sext.w a5,a5 20 mv a0,a5 21 ld s0,24(sp) 22 addi sp,sp,32 23 jr ra </pre>	<pre> 1 sumGen: 2 push {r7} 3 sub sp, sp, #20 4 add r7, sp, #0 5 str r0, [r7, #12] 6 str r1, [r7, #8] 7 str r2, [r7, #4] 8 ldr r2, [r7, #12] 9 ldr r3, [r7, #8] 10 add r2, r2, r3 11 ldr r3, [r7, #4] 12 add r3, r3, r2 13 mov r0, r3 14 adds r7, r7, #20 15 mov sp, r7 16 ldr r7, [sp], #4 17 bx lr </pre>	<pre> 1 sumGen: 2 push rbp 3 mov rbp, rsp 4 mov DWORD PTR [rbp 5 -4], edi 6 mov DWORD PTR [rbp 7 -8], esi 8 mov DWORD PTR [rbp 9 -12], edx 10 mov edx, DWORD PTR 11 [rbp-4] 12 mov eax, DWORD PTR 13 [rbp-8] 14 add edx, eax 15 mov eax, DWORD PTR 16 [rbp-12] 17 add eax, edx 18 pop rbp 19 ret </pre>
---	--	--

(a) RISC-V
(b) ARM
(c) x86

Figure 2: Funzione di somma

caricato a riga 17.

ARM

Nel caso ARM avviene lo stesso meccanismo. Tra le righe 8 e 12 avviene il caricamento dei primi due operandi la somma parziale e infine la somma totale.

x86

Infine per x86 il calcolo avviene tra le righe 7 e 11 nello stesso modo con cui viene eseguito in ARM.

2.3 Moltiplicazione

Moltiplicazioni per potenze di 2

```

1 int mult2(int num){
2     return 2 * num;
3 }

```

La funzione dato un numero di tipo intero restituisce il numero moltiplicato per 2. Per i sorgenti le parti di preparazione sono simili per le rispettive preparazioni precedenti.

Nel sorgente RISC-V (2.a) l'operazione di moltiplicazione per 2 avviene tramite uno shift logical left di 1 bit (SLLIW). Stesso concetto avviene nel sor-

```

1 mult2:
2     addi    sp,sp,-32
3     sd      s0,24(sp)
4     addi    s0,sp,32
5     mv      a5,a0
6     sw      a5,-20(s0)
7     lw      a5,-20(s0)
8     slliw   a5,a5,1
9     sext.w  a5,a5
10    mv      a0,a5
11    ld      s0,24(sp)
12    addi    sp,sp,32
13    jr      ra

```

(a) RISC-V

```

1 mult2:
2     push    {r7}
3     sub     sp, sp, #12
4     add     r7, sp, #0
5     str     r0, [r7, #4]
6     ldr     r3, [r7, #4]
7     lsls    r3, r3, #1
8     mov     r0, r3
9     adds    r7, r7, #12
10    mov     sp, r7
11    ldr     r7, [sp], #4
12    bx      lr

```

(b) ARM

```

1 mult2:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp
5     -4], edi
6     mov     eax, DWORD PTR
7     [rbp-4]
8     add     eax, eax
9     pop     rbp
10    ret

```

(c) x86

Figure 3: Moltiplicazione per 2

```

1 mult8:
2     addi    sp,sp,-32
3     sd      s0,24(sp)
4     addi    s0,sp,32
5     mv      a5,a0
6     sw      a5,-20(s0)
7     lw      a5,-20(s0)
8     slliw   a5,a5,3
9     sext.w  a5,a5
10    mv      a0,a5
11    ld      s0,24(sp)
12    addi    sp,sp,32
13    jr      ra

```

(a) RISC-V

```

1 mult8:
2     push    {r7}
3     sub     sp, sp, #12
4     add     r7, sp, #0
5     str     r0, [r7, #4]
6     ldr     r3, [r7, #4]
7     lsls    r3, r3, #3
8     mov     r0, r3
9     adds    r7, r7, #12
10    mov     sp, r7
11    ldr     r7, [sp], #4
12    bx      lr

```

(b) ARM

```

1 mult8:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp
5     -4], edi
6     mov     eax, DWORD PTR
7     [rbp-4]
8     sal     eax, 3
9     pop     rbp
10    ret

```

(c) x86

Figure 4: Moltiplicazione per 8

gente ARM (2.c) dove l'operazione di moltiplicazione per 2 avviene tramite lo shift left di 1 bit. Invece nel sorgente x86(2.b) la moltiplicazione avviene tramite una somma. Questa somma è un caso particolare, infatti se volessimo moltiplicare per una qualsiasi potenza di 2 (ad esempio in figura 3) le operazioni avvengono tutte tramite shift left di un opportuno valore. Con la figura 4 viene mostrato il calcolo di una moltiplicazione per 8. Nei casi RISC-V e ARM il calcolo avviene attraverso Shift e nel caso x86 avviene anche qui uno shift.

Moltiplicazione per una costante

Vengono presentati due codici molto simili, il primo moltiplica il numero per 31 che rappresenta più in generale un numero che dista da una potenza di 2 di 1. Il secondo è un caso più generale dove avviene la moltiplicazione di un numero non potenza di due e che dista da una potenza almeno di 2, nel nostro caso il numero è 30.

```

1 int mul31(int a) {
2     return a * 31;
3 }

```

```

1 int mul31(int a) {
2     return a * 30;
3 }

```

```

1 mul31:
2     addi    sp,sp,-32
3     sd      s0,24(sp)
4     addi    s0,sp,32
5     mv      a5,a0
6     sw      a5,-20(s0)
7     lw      a4,-20(s0)
8     mv      a5,a4
9     slliw   a5,a5,5
10    subw    a5,a5,a4
11    sext.w  a5,a5
12    mv      a0,a5
13    ld      s0,24(sp)
14    addi    sp,sp,32
15    jr      ra

```

```

1 mul31:
2     push    {r7}
3     sub     sp, sp, #12
4     add     r7, sp, #0
5     str     r0, [r7, #4]
6     ldr     r2, [r7, #4]
7     mov     r3, r2
8     lsls    r3, r3, #5
9     subs    r3, r3, r2
10    mov     r0, r3
11    adds    r7, r7, #12
12    mov     sp, r7
13    ldr     r7, [sp], #4
14    bx      lr

```

```

1 mul31:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp
5     -4], edi
6     mov     edx, DWORD PTR
7     [rbp-4]
8     mov     eax, edx
9     sal     eax, 5
10    sub     eax, edx
11    pop     rbp
12    ret

```

(a) RISC-V

(b) ARM

(c) x86

Figure 6: Moltiplicazione per 31

Nel caso della moltiplicazione per 31 l'approccio dei 3 sorgenti è identico. Viene calcolata la moltiplicazione per 32 attraverso shift logici e poi viene sottratto una volta il valore per ottenere la moltiplicazione per 31. Se il valore costante fosse 33, il numero successivo alla potenza di 2, l'operazione di sottrazione viene sostituita con una di addizione. Nel caso più generale invece abbiamo un approccio differente.

Partendo dall'x86 la moltiplicazione avviene semplicemente con l'istruzione `imul` a riga 6. Nel caso ARM l'operazione di moltiplicazione viene comunque eseguita da una singola istruzione (riga 8) ma vengono utilizzati i registri `r2` e `r3` che precedentemente (riga 6 e 7) vengono riempiti con gli operandi. Infine l'implementazione di RISC-V utilizza ancora shift. Nel caso della moltiplicazione per 30 avviene prima uno shift di 4 (moltiplicazione per 16) successivamente sottratto una volta il numero e infine al risultato avviene applicato uno shift di 1 (moltiplicazione per 2).

Quindi:

$$\begin{aligned}
 & ((num * 2^4) - num) * 2^1 = \\
 & = ((num * 16) - num) * 2 = \\
 & = 15 * num * 2 = num * 30
 \end{aligned}$$

In generale RISC-V utilizza opportuni shift combinate con addizioni e sottrazioni per ottenere il valore della costante.

```

1 mul30:
2     addi    sp,sp,-32
3     sd      s0,24(sp)
4     addi    s0,sp,32
5     mv      a5,a0
6     sw      a5,-20(s0)
7     lw      a4,-20(s0)
8     mv      a5,a4
9     slliw   a5,a5,4
10    subw    a5,a5,a4
11    slliw   a5,a5,1
12    sext.w  a5,a5
13    mv      a0,a5
14    ld      s0,24(sp)
15    addi    sp,sp,32
16    jr      ra

```

(a) RISC-V

```

1 mul30:
2     push    {r7}
3     sub     sp, sp, #12
4     add     r7, sp, #0
5     str     r0, [r7, #4]
6     ldr     r3, [r7, #4]
7     movs    r2, #30
8     mul     r3, r2, r3
9     mov     r0, r3
10    adds    r7, r7, #12
11    mov     sp, r7
12    ldr     r7, [sp], #4
13    bx      lr

```

(b) ARM

```

1 mul30:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp
5     -4], edi
6     mov     eax, DWORD PTR
7     [rbp-4]
8     imul    eax, eax, 30
9     pop     rbp
10    ret

```

(c) x86

Figure 7: Moltiplicazione per 30

2.4 Divisione

Per quanto riguarda la divisione viene utilizzato la stessa metodologia della moltiplicazione.