

Confronto compilatori

1 Introduzione

Confronto i compilatori. I Compilatori confrontati sono *gcc RISC-V*, *gcc x86_64* e *gcc ARM*. I compilati vengono compilati con il livello di ottimizzazione di default (-O0).

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

```

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    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

```

(b) x86

```

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

```

(c) ARM

Figure 1: Funzione di somma

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.

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

2.2 Addizione

```

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

```

Inserisci sorgenti Addizione generale

2.3 Moltiplicazione

```

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.

Moltiplicazioni per potenze di 2

```

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    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

```

(b) x86

```

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) ARM

Figure 2: 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    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

```

(b) x86

```

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

```

(c) ARM

Figure 3: Moltiplicazione per 8

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

Moltiplicazioni per non potenze di 2

Nel caso più generale i sorgenti si comportano in maniera molto diversa. Partendo dal sorgente ARM (4.c) il corpo della funzione è tra le righe 6, 7 e 8 dove, dopo aver recuperato il valore di num, attraverso l'operazione di mul viene calcolato il valore del prodotto tra num e 11 (la costante). Nel sorgente x86 (4.b) invece viene calcolato attraverso shift e sommando dei registri con i risultati (riga 5-10). Infine nel sorgente RISC-V (4.a) la moltiplicazione

```

1 mult11:
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,a5,1
10    addw    a5,a5,a4
11    slliw   a5,a5,2
12    subw    a5,a5,a4
13    sext.w  a5,a5
14    mv      a0,a5
15    ld      s0,24(sp)
16    addi    sp,sp,32
17    jr      ra

```

(a) RISC-V

```

1 mult11:
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, 2
10    add     eax, edx
11    add     eax, eax
12    pop     rbp
13    ret

```

(b) x86

```

1 mult11:
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, #11
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

```

(c) ARM

Figure 4: Moltiplicazione per 2

avviene attraverso shift e addizioni e sottrazioni (riga 7-13).

*?Possibile? inserimento di sottrazione e divisione
Inserisci operazione di shift*