Calcolo e allocazione in un array della successione di Fibonacci

Progetto d’esame di Architettura degli Elaboratori A.A. 2022/2023

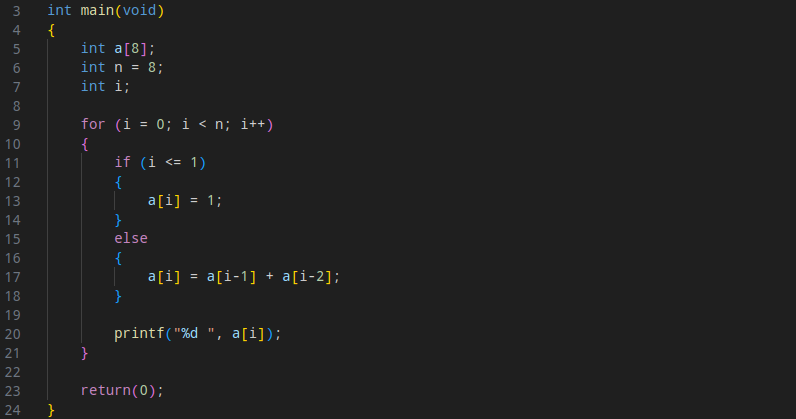
**Milena Balducci, 321791**

# 1 - Specifica

### 1.1 - Scopo del progetto

Lo scopo del progetto è, come da titolo, quello di calcolare e allocare in un array la successione di Fibonacci.  
Esistono due metodi per eseguire il calcolo: il metodo iterativo e quello ricorsivo. In questo progetto si è scelto di implementare il metodo iterativo.

### 1.2 - Specifica algoritmica



File: fibonacci.c

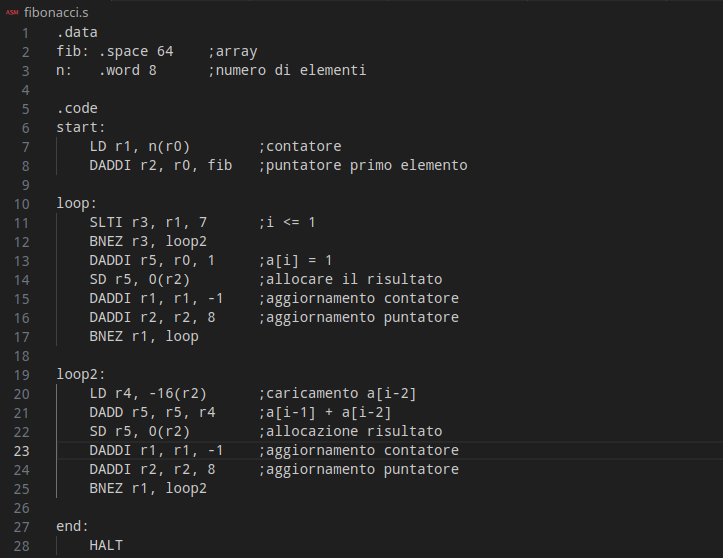
### 1.3 - Benchmark

L’algoritmo implementato utilizza dati di tipo int, allocati in un array di dimensione fissa pari a otto elementi. Tutti gli elementi della sequenza vengono allocati nell’array a runtime.

# 2 - Prima implementazione in Assembly

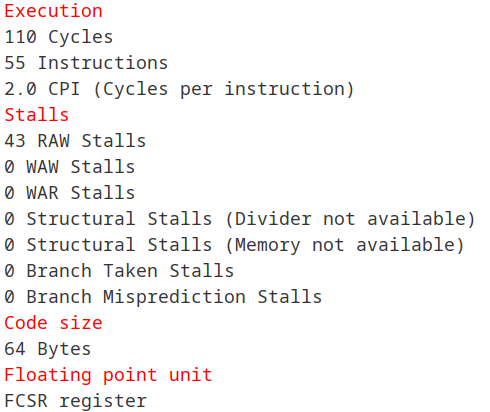
La prima implementazione del codice in Assembly si attiene strettamente alla versione del codice in linguaggio C. I dati utilizzati sono di tipo *double word* (.word o .word64) corrispondente a 8 byte. Per l’allocazione dell’array è stata utilizzata la direttiva **.space**, che alloca in memoria uno spazio vuoto pari al numero di byte indicato come parametro (in questo caso 64, dal momento che è necessario allocare otto elementi da 8 byte ciascuno).

Di seguito è riportata tale implementazione:



File: fibonacci.s

Eseguendo questa prima implementazione si ottengono **43 stalli**, tutti di tipo *RAW*. Osservando il listato del codice assembly si nota che responsabili di questi stalli sono le istruzioni alle righe 7-11, 11-12, 13-14 e 20-21-22, che necessitano di accedere agli stessi registri in passaggi consecutivi dell’esecuzione.

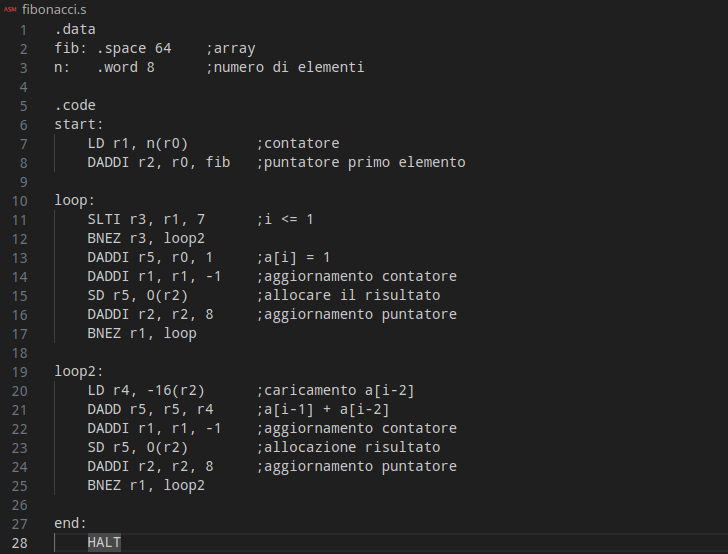


Di seguito si tenterà di applicare diverse tecniche di ottimizzazione per migliorare queste prestazioni.

# 3 - Ottimizzazione statica

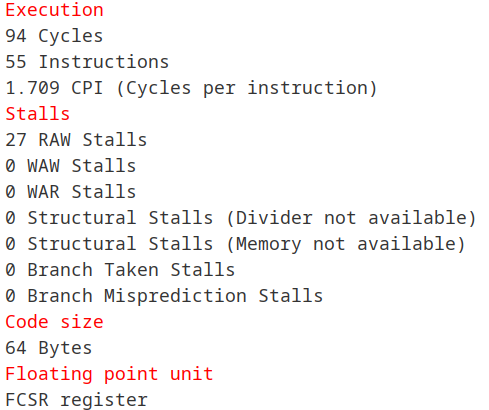
### 3.1 - Versione 2: Instruction Reordering

Il primo tentativo di ottimizzazione è stato fatto utilizzando la tecnica dell’***instruction reordering***, per tentare di allontanare istruzioni che creano conflitti.



File: fibonacci.s

Il listato originale non presentava particolari opportunità di riordinamento, quindi l’unica modifica effettuata è stata quella di spostare le istruzioni di decremento del contatore dalla riga 15 alla riga 14 nel primo loop e dalla riga 23 alla riga 22 nel secondo loop, in modo da allontanare il calcolo di un risultato dall’allocazione dello stesso in memoria. Questi due cambiamenti, per quanto piccoli, hanno prodotto un notevole miglioramento in termini di prestazioni:

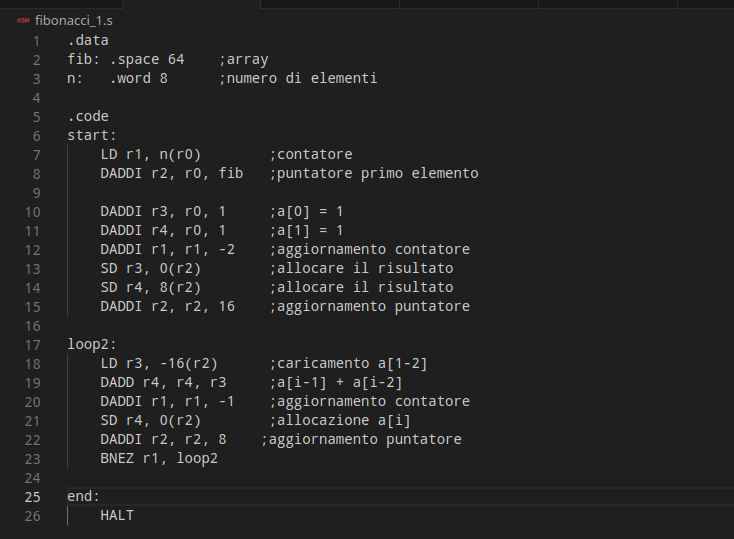


Come è possibile osservare, il numero di stalli è stato quasi dimezzato, e sia il numero di cicli che il *CPI* sono migliorati.

Gran parte degli stalli rimanenti è causata dalle due istruzioni alle righe 11 e 12, che necessitano entrambe dell’accesso al registro r3, e che non è possibile allontanare senza intaccare la funzionalità del codice o senza creare ulteriori stalli. La prossima versione del codice è mirata a risolvere questo problema.

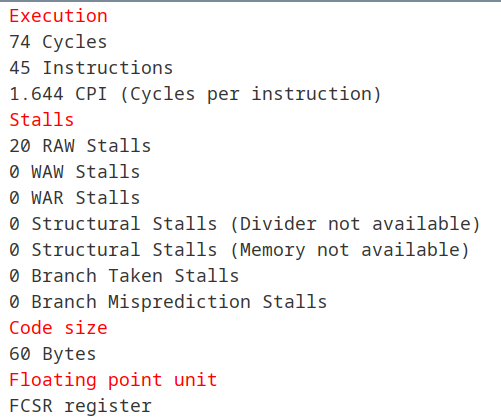
### 3.2 - Versione 3: Loop Unrolling

In questa versione del codice è stata utilizzata la tecnica del ***loop unrolling*** per eliminare il conflitto generato dalle istruzioni atte a decidere se continuare con il primo loop o passare al secondo. Per fare questo si è srotolato totalmente il primo dei due: in questo modo queste istruzioni non sono più necessarie.



File: fibonacci\_1.s

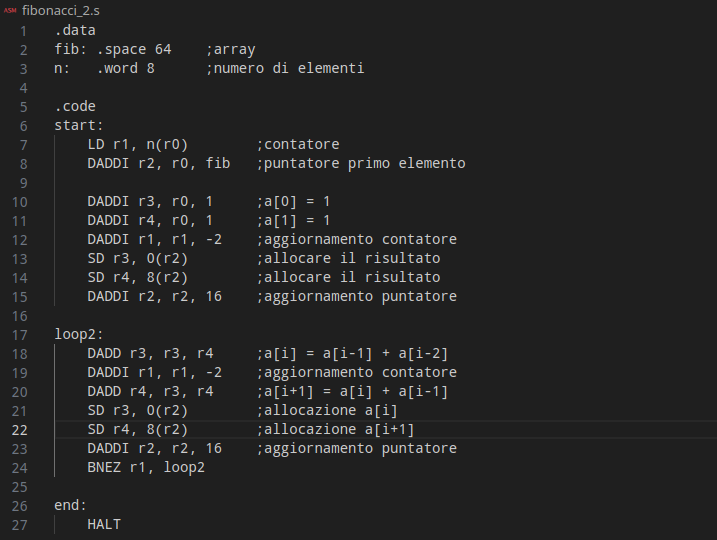
Questa tecnica ha prodotto i risultati desiderati, migliorando il numero di stalli, il *CPI* e il numero di cicli.



Il primo loop, ora srotolato totalmente, non presenta ulteriori stalli. Per migliorare ulteriormente le prestazioni del codice, dunque, è necessario concentrarsi sul secondo loop, che è quello che faremo nelle versioni seguenti.

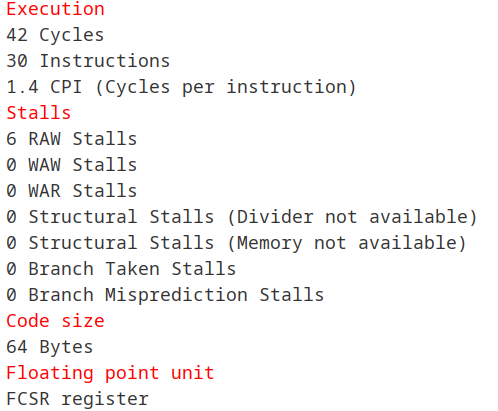
### 3.3 - Versione 4: Loop Unrolling e Instruction Reordering

In questa versione si è tentato di migliorare le prestazioni srotolando il secondo loop parzialmente, ovvero calcolando due numeri della sequenza all’interno di ogni iterazione, anzichè uno solo.



File: fibonacci\_2.s

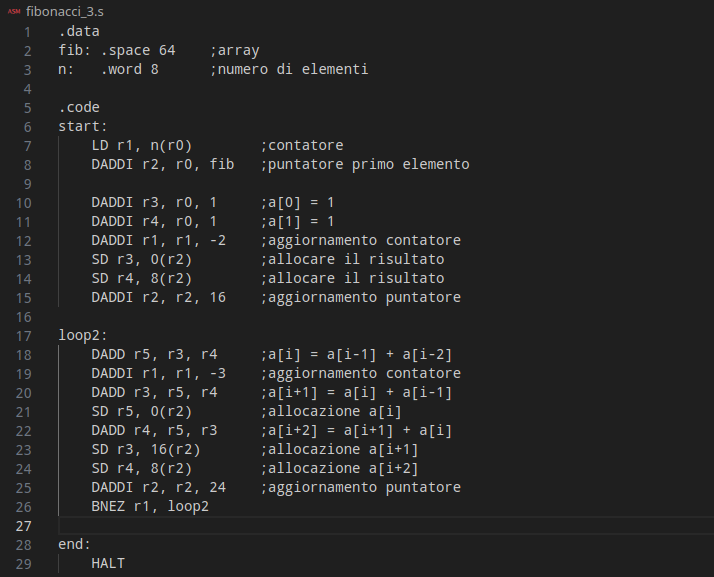
Calcolare due numeri alla volta ha permesso di rimuovere l’istruzione di load del penultimo numero calcolato, in quanto sia l’ultimo che il penultimo numero sono gia presenti all’interno dei registri. Oltre al *loop unrolling* è stata utilizzata anche la tenica dell’***instruction reordering***, per tentare di migliorare ulteriormente le prestazioni. L’applicazione di queste due tecniche ha prodotto un notevole miglioramento in termini di prestazioni:



Questa versione del codice ha permesso di rmuovere ben 14 stalli e di abbassare ulteriormente il *CPI* e il numero di cicli.

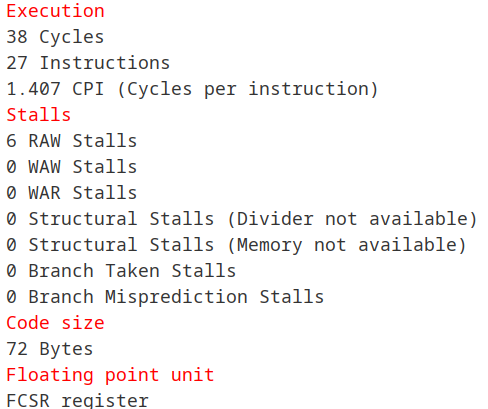
Dal momento che questa tecnica ha sortito l’effetto desiderato, si è tentata un’ulteriore ottimizzazione calcolando 3 numeri per ogni iterazione del loop.

### 3.4 - Versione 5: Loop Unrolling



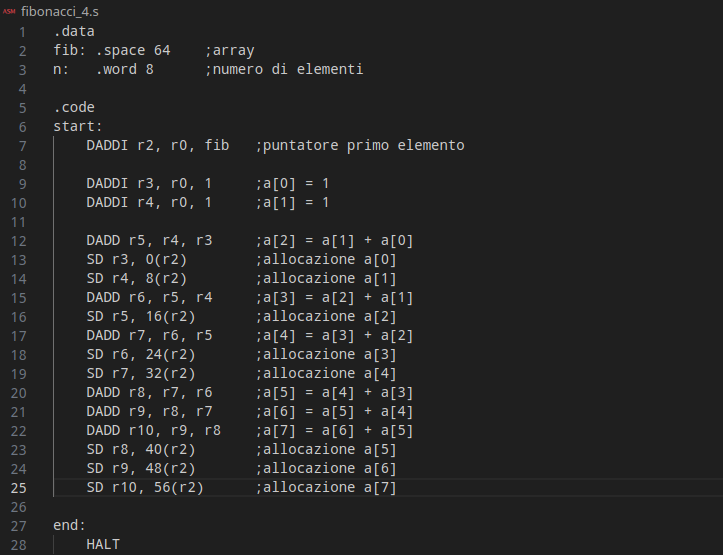
File: fibonacci\_3.s

In questa implementazione il numero di stalli è rimasto invariato, mentre sono migliorati il numero di cicli e di istruzioni, ma non il *CPI,* che è leggermente aumentato.



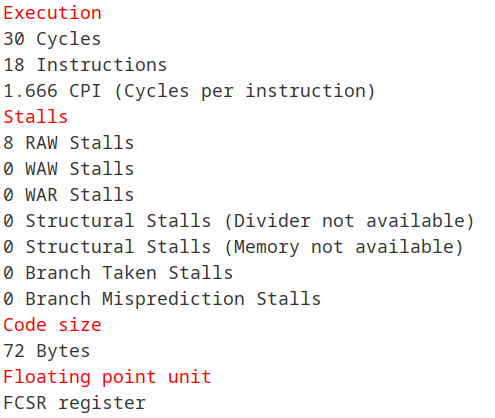
### 3.5 - Versione 6: Loop Unrolling totale e Instruction Reordering

Un ultimo tentativo di migliorare le prestazioni è stato fatto srotolando totalmente il loop, e dunque calcolando tutti i numeri della sequenza direttamente.



File: fibonacci\_4.s

Anche in questo caso è stata applicata la tecnica dell’*instruction reordering,* in modo da eliminare il maggior numero possibile di conflitti.



Come è possibile osservare, le prestazioni non sono totalmente migliorate. Solo il numero di cicli eseguiti ha avuto un miglioramento, assieme al numero di istruzioni, mentre il numero di stalli e il *CPI* sono aumentati.

# 4 - Conclusioni

Osservando le varie simulazioni riportate, risulta evidente che le versioni migliori in termini di prestazioni sono le **versioni 4 e 5**, che presentano il minor numero di stalli in assoluto, con la versione 4 che risulta la migliore in termini di *CPI* e la versione 5 che invece è più efficiente per numero di cicli e di istruzioni.