1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA APPLICATA
SCUOLA DI
SCIENZE TECNOLOGIE E FILOSOFIA DELL'INFORMAZIONE

RELAZIONE PROGETTO
Architettura degli Elaboratori

Anno Accademico 2019/2020

Sessione d'esame Invernale

Studente: Cossi Simone

Nr. Matricola 290796

s.cossi2@campus.uniurb.it

Calcolo dei numeri primi

Volendo dare una definizione per i numeri primi, diremo che:
un numero maggiore di 1 è primo se è divisibile solo per 1 e per sé stesso.
Al contrario, possiamo dare una definizione per i numeri composti scrivendo che: un numero maggiore di 1 è un numero composto se non è un numero primo.

In questo progetto andremo, in particolare, a calcolare i numeri primi dallo 0 al 51. Il programma funziona sia per un sottoinsieme qualsiasi dell'insieme da noi utilizzato sia per un insieme più grande. Ho scelto di utilizzare questo insieme poiché il linguaggio assembly non permette interazioni con l'utente (quindi il range in cui si vanno a calcolare i numeri primi deve essere stabilito all'interno del file da eseguire) e volendo analizzare il programma in fase di esecuzione ciclo per ciclo volevo pure giungere alla terminazione del programma, il quale richiede un grande numeri di cicli anche per pochi numeri da verificare.

Detto ciò:

1. Andremo ad analizzare il programma di partenza scritto in linguaggio C.
2. Andremo poi vedere la traduzione del programma dal linguaggio C all'assembly.
3. Vedremo tutte le varie ottimizzazioni da cosa è cambiato all'interno del programma a come è stato sostituito ai vari compromessi che ho dovuto adottare per ottimizzare il programma.
4. Concludendo con un paragone delle statistiche iniziali e finali e riassumendo tutti i compromessi adottati.

Codice in linguaggio C

```
#include <stdio.h>
void main ()
{
	int x, i;
	for (x = 1; x <= 100; x++)
	{
		for (i = 2; i <= x / 2; i++)
		{
			if (x % i == 0)
			{
				break;
			}
		}
		if (x % i != 0)
		{
			printf( "%d\n", x );
		}
	}
}
```

Traduzione del codice in C in linguaggio Assembly

```
        .data
x:      .word   2
i:      .word   2
lim:    .word   51              ; numero limite per calcolo dei numeri primi

        .text
start:  LW      r1, x(r0)       ; carico x
        Lw      r2, i(r0)       ; carico i
        LW      r3, lim(r0)     ; carico lim

loop0:  DADDI   r1, r1, 1       ; incremento di 1 x
        DADDI   r2, r0, 2       ; incremento di 2 r2
        BEQ     r1, r3, end     ; se r1 ed r3 sono uguali ho calcolato tutti i numeri primi interessati quindi termino il programma

loop1:  BEQ     r2, r1, loop2   ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2      ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2      ; moltiplico il risultato per il disore
        DSUB    r8, r1, r7      ; calcolo il resto
        DADDI   r2, r2, 1       ; incremento di 1 i
        BEQ     r8, r0, loop0   ; se il resto della divisione è 0 salta il ciclo
        BNE     r8, r0, loop1   ; se il resto è diverso da 0 divido per il numero successivo


loop2:  DADDI   r10, r1, 0      ; carico in r10 il numero primo calcolato dal loop 1
        BNE     r1, r3, loop0   ; se r1 ed r3 non sono uguali torno al loop 0 per continuare il calcolo


end:
        HALT
```

Osservazione delle prestazioni del codice

```
Execution
13142 Cycles
2609 Instructions
5.037 Cycles Per Instruction (CPI)

Stalls
17850 RAW Stalls
0 WAW Stalls
0 WAR Stalls
350 Structural Stalls
379 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
64 Bytes
```

Lo screenshot riportato mostra le prestazioni del codice in assembly, appena tradotto dal linguaggio c e verificatone il corretto funzionamento per mezzo della simulazione tramite WinMIPS64.

Detto ciò c'è da aggiungere che già in questo codice, per garantirne il corretto funzionamento, ho dovuto "sacrificare" il calcolo del numero primo '2'.

Problemi di sorting e scelte di ottimizzazione

Prima di andare alle ottimizzazioni mi sono soffermato sull'analizzare quelli che sono i problemi di sorting nel linguaggio assembly.

I principali problemi sono:

- L'ottimizzazione del CPI (Cicli per istruzione)
  Ovvero cercare di diminuire il numero di cicli di clock per istruzione anche se a discapito del numero di istruzioni.
- L'ottimizzazione delle istruzioni
  Ovvero cercare di diminuire il numero di istruzioni anche se a discapito del CPI

Detto ciò ho scelto di intraprendere la strada per l'ottimizzazione del CPI, dato l'elevato CPI del programma.

Prima ottimizzazione del codice

Per quanto riguarda la prima ottimizzazione sono andato semplicemente a modificare due punti che mi permettono di saltare diverse istruzioni.

Il primo codice analizzava tutti i numeri tra il 2 e il prestabilito 51 e allocava in r10 i numeri primi, mentre con la prima ottimizzazione vado ad analizzare tutti i numeri DISPARI tra il 3 e il 51, dato che i numeri pari non possono ovviamente essere numeri primi (2 escluso).

Con questa scelta però devo assicurarmi di settare il valore di 'lim' con un numero dispari, altrimenti il programma non riuscirà mai a terminare. Potrei aggirare il problema aggiungendo un'istruzione ma dato che se si vuole modificare il numero di numeri primi da calcolare bisogna modificare il valore di 'lim' prima dell'esecuzione del codice non l'ho visto come un problema.

Codice ottimizzato

```
        .data
x:      .word   1
i:      .word   2
lim:    .word   51              ; il valore deve essere dispari

        .text
start:  LW      r1, x(r0)       ; carico x
        Lw      r2, i(r0)       ; carico i
        LW      r3, lim(r0)     ; carico lim

loop0:  DADDI   r1, r1, 2       ; incremento di 2 x
        DADDI   r2, r0, 2       ; riporto r2 a 2
        BEQ     r1, r3, end     ; se r1 ed r3 sono uguali termino il programmma

loop1:  BEQ     r2, r1, loop2   ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2      ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2      ; moltiplico il risultato per il disore
        DSUB    r8, r1, r7      ; calcolo il resto
        DADDI   r2, r2, 1       ; incremento di 1 i
        BEQ     r8, r0, loop0   ; se il resto della divisione è 0 salta il ciclo
        BNE     r8, r0, loop1   ; se il resto è diverso da 0 divido per il numero successivo


loop2:  DADDI   r10, r1, 0      ; alloco il numero primo calcolato in r10
        BNE     r1, r3, loop0   ; se r1 non è uguale a r3 torno al loop 0


end:
        HALT
```

Osservazioni sulle prestazioni del codice

Codice originale:

```
Execution
13142 Cycles
2609 Instructions
5.037 Cycles Per Instruction (CPI)

Stalls
17850 RAW Stalls
0 WAW Stalls
0 WAR Stalls
350 Structural Stalls
379 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
64 Bytes
```

Codice ottimizzato:

```
Execution
12206 Cycles
2393 Instructions
5.101 Cycles Per Instruction (CPI)

Stalls
16626 RAW Stalls
0 WAW Stalls
0 WAR Stalls
326 Structural Stalls
355 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
64 Bytes
```

Con questa semplice modifica l'esecuzione ha circa:

- mille cicli di meno
- duecento istruzioni in meno
- mille stalli di meno

Tuttavia, il CPI è aumentato di poco poiché le istruzioni che ho eliminato erano i cicli più corti che il programma aveva, quindi togliendo quelle il CPI è aumentato leggermente.

Seconda e ultima ottimizzazione del codice

Nella seconda ottimizzazione sono andato a modificare il codice come nella prima ottimizzazione, ovvero a ragionamento e intuito. In questo codice smetto anche di dividere per tutti i numeri pari, dato che analizzando solamente i numeri dispari le divisioni per i numeri pari diventano inutili.

Codice ottimizzato

```
        .data
x:      .word   1
lim:    .word   51              ; deve essere un numero dispari

        .text
start:  LW      r1, x(r0)       ; carico x
        LW      r3, lim(r0)     ; carico lim

loop0:  DADDI   r1, r1, 2       ; incremento di 1 x
        DADDI   r2, r0, 3       ; do a r2 il valore 3
        BEQ     r1, r3, end     ; se r1 ed r3 sono uguale termino il programma

loop1:  BEQ     r2, r1, loop2   ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2      ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2      ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7      ; calcolo il resto
        DADDI   r2, r2, 2       ; incremento di 2 il divisore
        BEQ     r8, r0, loop0   ; se il resto della divisione è 0 salta il ciclo
        BNE     r8, r0, loop1   ; se il resto è diverso da 0 divido per il numero successivo


loop2:  DADDI   r10, r1, 0      ; alloco in r10 il numero primo calcolato
        BNE     r1, r3, loop0   ; se r1 ed r3 sono diversi torno al loop0


end:
        HALT
```

Osservazioni sulle prestazioni del codice

Codice precedentemente ottimizzato:

```
Execution

12206 Cycles
2393 Instructions
5.101 Cycles Per Instruction (CPI)

Stalls

16626 RAW Stalls
0 WAW Stalls
0 WAR Stalls
326 Structural Stalls
355 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

64 Bytes
```

Codice appena ottimizzato:

```
Execution

5915 Cycles
1202 Instructions
4.921 Cycles Per Instruction (CPI)

Stalls

7956 RAW Stalls
0 WAW Stalls
0 WAR Stalls
156 Structural Stalls
185 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

60 Bytes
```

Con questa semplice modifica l'esecuzione ha circa:

- metà dei cicli
- metà delle istruzioni
- metà degli stalli
- CPI leggermente più basso

Con queste modifiche sono finalmente riuscito ad abbassare il leggermente il CPI.

Nonostante ciò non ero ancora riuscito ad abbassare quanto desideravo il CPI allora ho provato altri metodi.

Metodi di ottimizzazione del codice

Mettendo da parte le prime ottimizzazione, che sono frutto di semplice intuito, ci sono due metodi principali per cercare di ottimizzare il codice:

- Loop-unrolling (srotolamento del codice)
  Consiste nella sostituzione di un ciclo come ad esempio quello che ho chiamato ‘loop1’ con lo scrivere tutte le istruzioni senza che una stessa istruzione non venga ripetuta.
- Instruction reordering (riordinamento delle istruzioni)
  Consiste nel riordinare le istruzioni in modo che il risultato non cambi. Questo viene effettuato poiché le istruzioni hanno un tempo di esecuzione e diverse istruzioni hanno diversi tempi di esecuzione e facendo ciò si potrebbe diminuire il numero di stalli, il CPI e il numero di cicli.

Loop-unrolling

```
.data
x:      .word   1
lim:    .word   51                      ; deve essere un numero dispari

        .text
start:  LW      r1, x(r0)     ; carico x
        LW      r3, lim(r0)   ; carico lim

loop0:  DADDI   r1, r1, 2               ; incremento di 2 x
        DADDI   r2, r0, 3               ; do a r2 il valore 3
        BEQ     r1, r3, end             ; se r1 ed r3 sono uguale termino il programma

loop1:  BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2              ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2              ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7              ; calcolo il resto
        DADDI   r2, r2, 2               ; incremento di 2 il divisore
        BEQ     r8, r0, loop0           ; se il resto della divisione è 0 salta il ciclo

        BEQ     r2, r1, loop2           ; se divisore e dividendo sono uguali salta
```

```
        DDIV    r6, r1, r2          ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2          ; moltiplico il risultato per il divisore
        DSUB    r8, r1, r7          ; calcolo il resto
        DADDI   r2, r2, 2           ; incremento di 2 il divisore
        BEQ     r8, r0, loop0       ; se il resto della divisione è 0 salta il ciclo

loop2:  DADDI   r10, r1, 0          ; alloco in r10 il numero primo calcolato
        BNE     r1, r3, loop0       ; se r1 ed r3 sono diversi torno al loop0

end:
        HALT
```

Codice precedentemente ottimizzato:

```
Execution

5915 Cycles
1202 Instructions
4.921 Cycles Per Instruction (CPI)

Stalls

7956 RAW Stalls
0 WAW Stalls
0 WAR Stalls
156 Structural Stalls
185 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

60 Bytes
```

Codice con il loop-unrolling:

```
Execution

4456 Cycles
852 Instructions
5.230 Cycles Per Instruction (CPI)

Stalls

6273 RAW Stalls
0 WAW Stalls
0 WAR Stalls
123 Structural Stalls
33 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

320 Bytes
```

Una volta eseguito il loop-unrolling perdo la possibilità di andare a calcolare un numero maggiore di numeri primi modificando il valore di ‘lim’, nonostante ciò se il compito del programma fosse strettamente limitato al calcolo di soli questi numeri otterrei:

- una diminuzione di circa 1500 cicli
- circa 350 istruzioni in meno
- circa 1700 stalli in meno
- una diminuzione drastica dei branch taken stalls.

Detto ciò ci sono ovviamente delle controparti:

- il CPI si alza di quasi 0.3
- il peso equivale a oltre 5 volte il peso del codice “compresso”.

Dato che il mio obbiettivo è riuscire a far calare il CPI trascurerò l’ottimizzazione appena provata per andare a provare l’instruction reordering.

Instruction reordering

```
        .data
x:      .word   1
lim:    .word   51              ; deve essere dispari

        .text
start:  LW      r1, x(r0)       ; carico x
        LW      r3, lim(r0)     ; carico lim

loop0:  DADDI   r1, r1, 2       ; incremento di 1 x
        DADDI   r2, r0, 3       ; do a r2 il valore 3
        BEQ     r1, r3, end     ; se r1 ed r3 sono uguali termino il programma

loop1:  BEQ     r2, r1, loop2   ; se divisore e dividendo sono uguali salta
        DDIV    r6, r1, r2      ; eseguo la divisione e la alloco in r6
        DMUL    r7, r6, r2      ; moltiplico il risultato per il disore
        DSUB    r8, r1, r7      ; calcolo il resto
        DADDI   r2, r2, 2       ; incremento di 2 il divisore
        BNE     r8, r0, loop1   ; se il resto è diverso da 0 divido per il numero successivo
        BEQ     r8, r0, loop0   ; se il resto della divisione è 0 salta il ciclo


loop2:  DADDI   r10, r1, 0      ; alloco in r10 il numero primo calcolato
        BNE     r1, r3, loop0   ; se r1 ed r3 sono diversi torno al loop0


end:
        HALT
```

Purtroppo, non ho potuto eseguire molto l'instruction reordering poiché quasi tutte le istruzioni hanno bisogno di attendere il risultato dell'istruzione precedente.

Osservazioni sulle prestazioni del codice

Codice precedentemente ottimizzato:

```
Execution

5915 Cycles
1202 Instructions
4.921 Cycles Per Instruction (CPI)

Stalls

7956 RAW Stalls
0 WAW Stalls
0 WAR Stalls
156 Structural Stalls
185 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

60 Bytes
```

Codice con l'instruction reordering:

```
Execution

5779 Cycles
1066 Instructions
5.421 Cycles Per Instruction (CPI)

Stalls

7956 RAW Stalls
0 WAW Stalls
0 WAR Stalls
156 Structural Stalls
185 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

60 Bytes
```

Come possiamo ben notare i cicli diminuiscono come diminuiscono le istruzioni, tuttavia il CPI aumenta e non di poco. Ciò significa che se andassi ad aumentare il numero di numeri primi che voglio calcolare il codice sarà meno efficiente, quindi ho scelto di tenere come codice finale il codice prima dell'instruction reordering.

Conclusioni

Sono riuscito ad ottimizzare il codice diminuendo cicli, istruzioni e stalli anche se il mio vero obbiettivo era di diminuire il CPI. Analizzando meglio il codice e soprattutto l'esecuzione di quest'ultimo tramite WinMIPS64 ho notato che il principale motivo del CPI così alto e del grande numero di stalli era causato dall'elevato numero di cicli che necessita la divisione.

Detto ciò WinMIPS64 ci permette di modificare il numero di cicli che la divisione utilizza, lasciando sempre un numero minimo che necessita per il funzionamento.

Codice Con divisione standard (24 cicli)

```
Execution

5915 Cycles
1202 Instructions
4.921 Cycles Per Instruction (CPI)

Stalls

7956 RAW Stalls
0 WAW Stalls
0 WAR Stalls
156 Structural Stalls
185 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

60 Bytes
```

Codice con divisione minima (10 cicli)

```
Execution

3731 Cycles
1202 Instructions
3.104 Cycles Per Instruction (CPI)

Stalls

3588 RAW Stalls
0 WAW Stalls
0 WAR Stalls
156 Structural Stalls
185 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size

60 Bytes
```

Con questa modifica vediamo che il numero scendono il numero di:

- Cicli
- Stalli
- CPI

Finalmente sono riuscito a far scendere drasticamente il CPI, circa del 40% riuscendo ad eliminare anche un grande numero di stalli.
Rimane pur sempre un CPI elevato ed un elevato numero di stalli ma credo sia impossibile riuscire ad abbassarlo ancora di molto ed a eliminare gli stalli poiché il calcolo dei numeri primi necessita della divisione che è la causa dell'elevato CPI e numero di stalli.