Relazione Progetto Architettura 2022

Studente: Joshua Micheletti
Matricola: 283057

Contenuti Relazione:

- Algoritmo implementato
- Implementazione dell’algoritmo
- Procedimenti di ottimizzazione
- Alternative ottimizzazioni

Algoritmo Implementato

L’algoritmo implementato è l’automa cellulare del **Gioco Della Vita** (Game Of Life), sviluppato dal matematico inglese John Horton Conway, nel 1970.

Un automa cellulare è un modello matematico usato per simulare e descrivere l’evoluzione di un sistema discreto (ogni entità del sistema ha un numero finito di stati).

Nel caso del Gioco Della Vita, l’algoritmo consiste nel simulare un sistema assimilabile ad un concetto di vita, dove ogni entità può nascere o morire, ogni entità può essere viva o morta (1, 0).

Ogni passo della simulazione si chiama “Generazione”, ogni generazione rappresenta uno stato del sistema.

Per calcolare la generazione successiva, bisogna controllare ogni entità (cella) presente nella simulazione (tavola) e determinare il suo stato futuro.

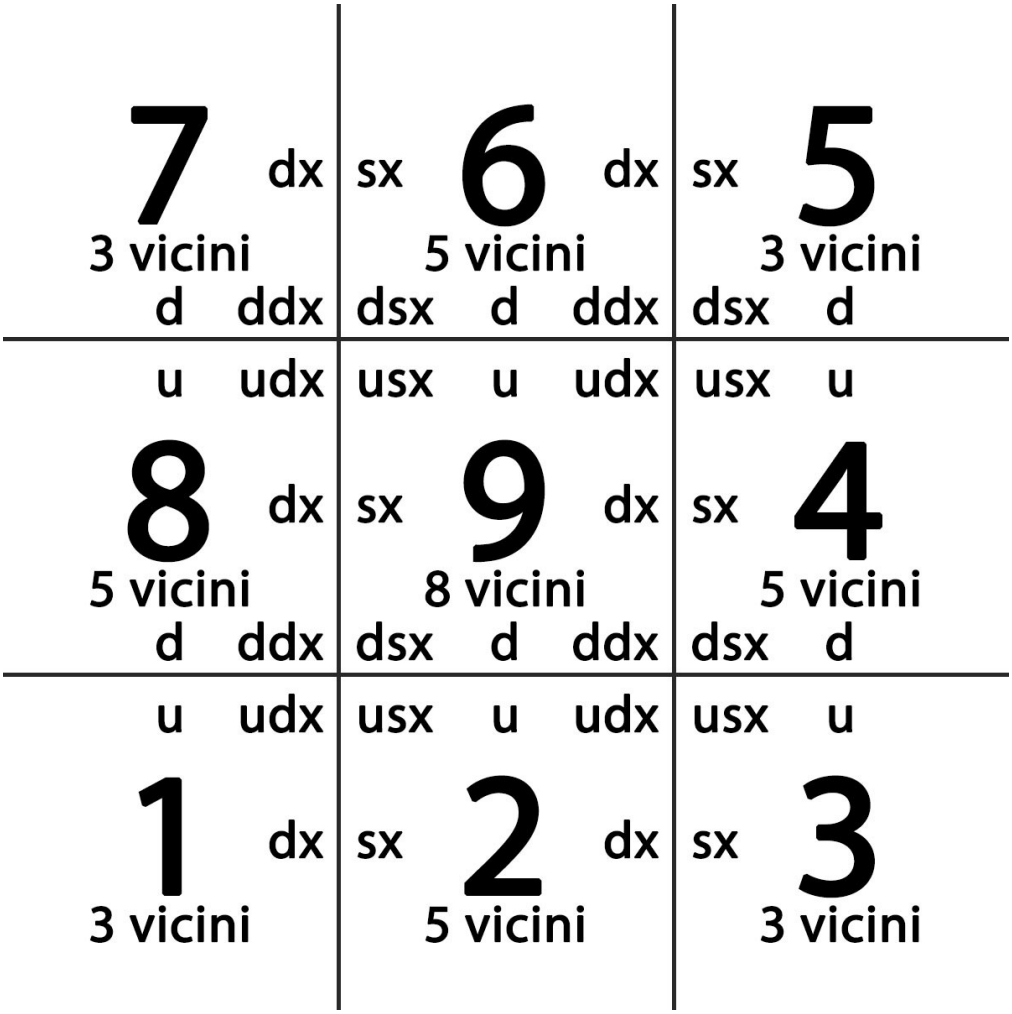
Il Gioco Della Vita fornisce delle regole per poter calcolare lo stato di ogni cella nella generazione successiva, e ogni regola dipende dai suoi vicini a distanza 1, quindi per calcolare la generazione successiva, bisogna controllare ogni cella e i suoi rispettivi vicini.

1 usx	0 u	0 udx
0 sx	0 cell	1 dx
0 dsx	0 d	1 ddx

Nell’immagine possiamo vedere che ogni cella ha 8 vicini, contrassegnati per posizione come:

- **dsx**: giù sinistra
- **d**: giù
- **ddx**: giù destra
- **dx**: destra
- **udx**: su destra
- **u**: su
- **usx**: su sinistra
- **sx**: sinistra

Ci sono però casi limite nei bordi della tabella in cui una cella può avere meno di 8 vicini:



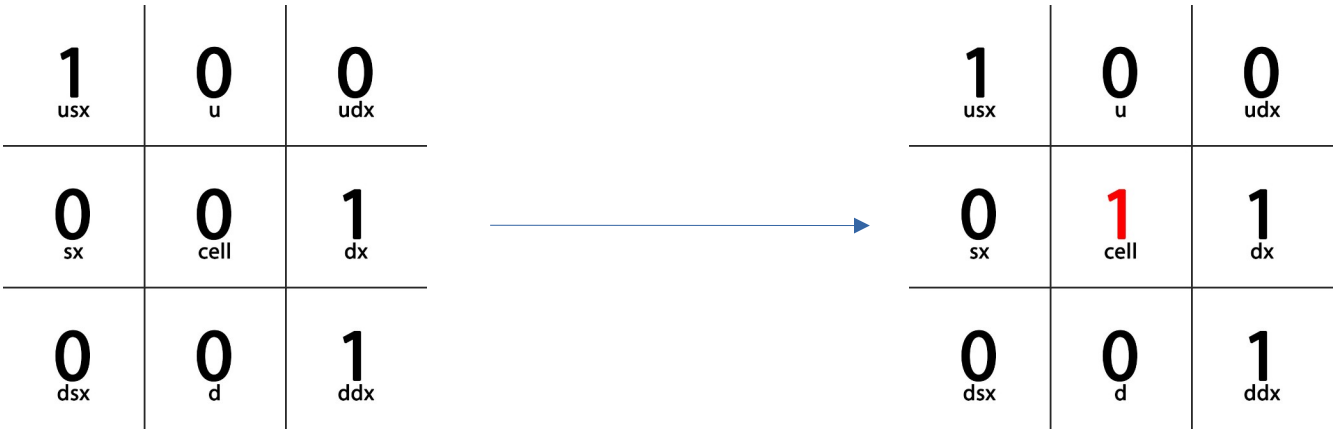
La gran parte delle celle fa parte della categoria di celle 9, però ci sono casi ai limiti della tabella (prima riga (123), ultima riga (765), prima colonna (187), ultima colonna (345)) in cui le celle hanno meno vicini validi.

Identificare i casi limite ci consente non solo di ridurre il numero di passaggi per cercare i vicini, ma evita anche di accedere a posizioni di vicini non validi.

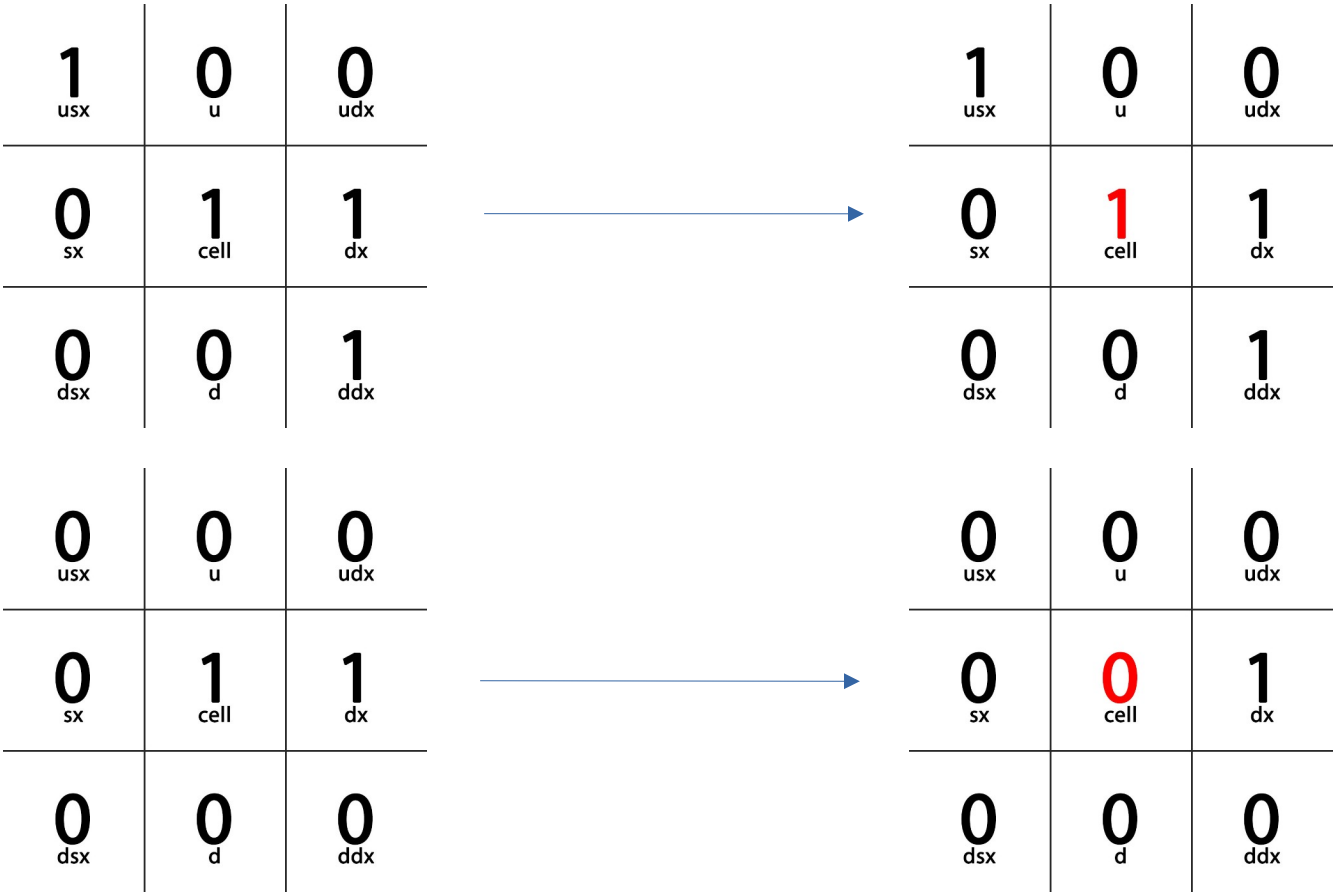
Per esempio, se fossimo in un caso 3 e ci spostassimo a destra o in giù, usciremmo dalla tabella!

Una volta verificati i vicini di ogni cella, possiamo determinare lo stato della cella esaminata nella generazione successiva, seguendo le regole del Gioco Della Vita:

- Se una cella morta ha **3** vicini vivi, la cella diventa viva



- Se una cella viva ha **2** o **3** vicini vivi, la cella rimane in vita, altrimenti muore



Implementazione dell'algoritmo

Il codice che implementa il Gioco Della Vita è codice assembly MIPS.

La tavola di riferimento è una tavola 16x16 in spazio piano (non toroidale), in totale questa tavola contiene 256 celle.

Per l'implementazione dell'algoritmo vengono utilizzati 512 Byte di memoria, divisi in 256 Byte per la tavola corrente e 256 Byte per la tavola contenente la generazione successiva, in questo modo ogni cella viene rappresentata con 1 Byte.

Il programma sfrutta in totale **19** registri:

- **R0**: registro che contiene il valore 0
- **R1**: registro che contiene il valore 1
- **R2**: offset da applicare all'indirizzo di memoria della tavola corrente per accedere ad una cella
- **R3**: offset da applicare all'indirizzo di memoria della tavola contenente la generazione successiva per accedere ad una cella (sempre diverso da R2 con distanza 256)
- **R4**: offset da applicare all'indirizzo di memoria della tabella corrente per accedere ai vicini della cella corrente
- **R5**: valore della cella corrente (inizializzato al primo valore della tabella)
- **R6**: contatore di vicini vivi alla cella corrente
- **R7**: registro temporaneo per scambiare i valori di R2 e R3
- **R8**: registro temporaneo per register renaming, onde evitare dipendenze di tipo RAW sul registro R9 (valore del vicino visitato)
- **R9**: valore del vicino visitato
- **R12**: contatore decrescente che rappresenta la coordinata X per muoversi nella tavola
- **R13**: contatore decrescente che rappresenta la coordinata Y per muoversi nella tavola
- **R16**: valore limite delle coordinate X e Y (16)
- **R17**: numero di vicini per cui una cella morta nasce ed una cella viva resta in vita (3)
- **R18**: numero di vicini per cui una cella viva rimane in vita (2)
- **R28**: registro contenente il colore con cui rappresentare graficamente le celle (azzurro ■ , (30, 144, 255, 0))
- **R29**: numero per indicare al terminale la sua funzionalità (inizializzato a 5)

- **R30**: indirizzo di memoria contenente la funzionalità del terminale
  5 – disegna a schermo
  6 – pulisci il terminale testuale
  7 – pulisci il terminale grafico
  8 – leggi un Byte dal terminale

- **R31**: indirizzo di memoria contenente i dati da rappresentare a terminale

Il programma è strutturato in 5 fasi principali:
- **Rappresentazione della Cella**
- **Calcolo dei Vicini**
- **Calcolo della Generazione Successiva**
- **Scorrimento Tavola**
- **Scambio di Tavole**

Il flusso di esecuzione del programma consiste nel eseguire ogni passaggio delle prime 4 fasi (rappresentazione, vicini, generazione successiva, scorrimento tavola) per ogni cella, finito il calcolo della tabella intera, la tabella corrente e la tabella di generazione successiva vengono scambiate, così da poter calcolare la generazione ancora dopo, e così via.

Rappresentazione della Cella

controllaStato:
La prima fase inizia controllando il valore della cella corrente (precedentemente caricata in R5), nel caso in cui sia 1, passa al codice per gestire la rappresentazione di una cella viva, altrimenti passa alla fase successiva (non abbiamo bisogno di rappresentare celle morte)

```
bnez r5, vivo
j calcolaVicini
```

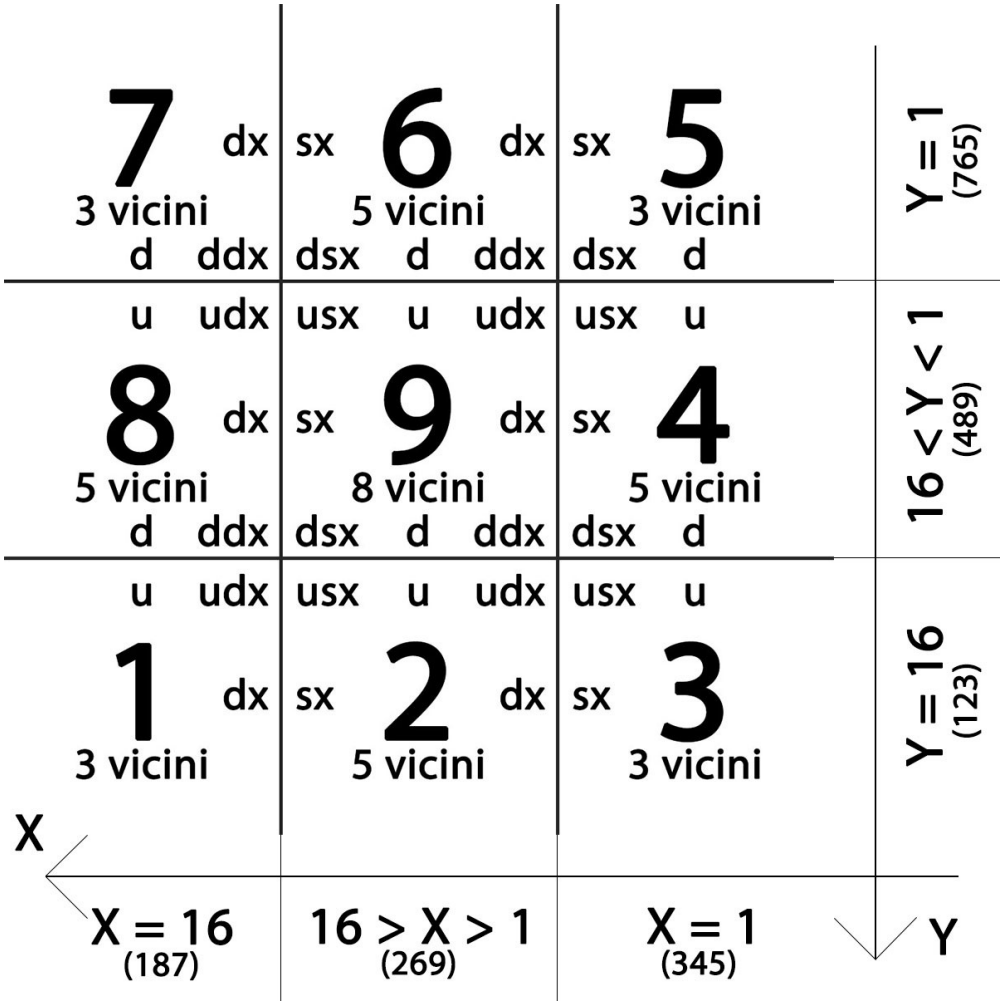
vivo:
Nel caso in cui la cella sia viva, dobbiamo rappresentarla. Per rappresentare una cella nel terminale grafico dobbiamo fornire al terminale informazioni riguardo al colore con cui disegnare, la posizione in cui disegnare una cella (x,y) e il comando effettivo di disegnare. Nel nostro caso il colore è contenuto in R28, le coordinate X e Y sono contenute in R12 e R13 rispettivamente, e il comando per disegnare è contenuto in R29 (con valore 5). Passando questi valori alle aree di memoria corrette (DATA (R31) e CONTROL (R30)), otteniamo la

rappresentazione di una cella.

```
sw r28, 0(r31) ; colore (R28) → DATA (0(r31))
sb r12, 5(r31) ; X (r12) → DATA (5(r31))
sb r13, 4(r31) ; Y (r13) → DATA (4(r31))
sd r29,  (r30) ; comando (r29) → CONTROL (r30)
```

Calcolo dei Vicini

calcolaVicini:
La fase di calcolo dei vicini consiste nel trovare il numero di vicini vivi intorno alla cella corrente.
Come spiegato sopra ci sono casi in cui alcune celle possono avere 8, 5 o 3 vicini, a seconda della loro posizione nella tavola (x,y).



Controllando i valori di X (R12) e Y (R13) possiamo capire in che caso ci troviamo. Per esempio se controlliamo il valore di X (R12), possiamo già sapere se ci troviamo in uno dei 3 casi 187 – 269 – 345

```
beq r12, r16, caso187        ; caso 187
beq r12, r1, caso345         ; caso 345
...                          ; caso 269
```

Una volta identificato il gruppo di casi a seconda di X, possiamo identificare il caso specifico a seconda di Y:

caso187:
```
beq r13, r16, caso1   ; caso 1
beq r13, r1, caso7    ; caso 7
...                   ; caso 8
```

caso345:
```
beq r13, r16, caso3   ; caso 3
beq r13, r1, caso5    ; caso 5
...                   ; caso 4
```

calcolaVicini:
```
...                   ; caso 269
beq r13, r16, caso2   ; caso 2
beq r13, r1, caso6    ; caso 6
...                   ; caso 9
```

Conoscendo il caso della cella corrente, possiamo trovare i vicini validi tramite R4 e contare i vicini vivi in R6.
Per accedere alle posizioni intorno alla cella, bisogna spostarsi in memoria di un determinato numero di posizioni rispetto ad R2 (la cella corrente):

+15 usx	+16 u	+17 udx
-1 sx	0 cell	+1 dx
-17 dsx	-16 d	-15 ddx

Per accedere ad una cella in una riga successiva (**u**), bisogna accedere all'area di memoria 16 posizioni più avanti (offset = **+16**). Per accedere ad una cella inferiore (**d**), bisogna accedere all'area in 16 posizioni prima (offset = **-16**). Per accedere ad una cella a destra (**dx**), bisogna aumentare l'indirizzo di 1 (offset = **+1**). Per accedere ad una cella a sinistra (**sx**), bisogna applicare un offset di -1 (offset = **-1**).

Ogni volta che visitiamo un vicino, salviamo il suo valore nel registro R8 o R9 (due registri per evitare dipendenze RAW), e sommiamo i valori contenuti in R8 e R9 in R6, così da ottenere il numero di vicini vivi, in quanto solo i vicini vivi (1) contribuiranno al conteggio.

Per esempio nel caso 9, in cui tutti i vicini sono validi:

calcolaVicini:

```
...                     ; caso 9
daddi r4, r2, -17       ; dsx
lb r9, tavola0(r4)      ; carica il valore del vicino
daddi r4, r2, -16       ; d
dadd r6, r6, r9         ; conta i vicini visitati
lb r8, tavola9(r4)
daddi r4, r2, -15       ; ddx
daddi r6, r6, r8
lb r9, tavola0(r4)
daddi r4, r2, 1         ; dx
dadd r6, r6, r9
lb r8, tavola0(r4)
daddi r4, r2, 17        ; udx
dadd r6, r6, r8
lb r9, tavola0(r4)
daddi r4, r2, 16        ; u
dadd r6, r6, r9
lb r8, tavola0(r4)
daddi r4, r2, 15        ; usx
dadd r6, r6, r8
lb r9, tavola0(r4)
daddi r4, r2, -1        ; sx
lb r8, tavola0(r4)
dadd r6, r6, r9
dadd r6, r6, r8
j regole                ; calcolo della generazione successiva
```

Alternare l'uso di due registri consente di distribuire le istruzioni in modo da non avere letture subito dopo scritture (RAW).

Ogni altro caso è costituito da un sottoinsieme di questo caso generico (per esempio, il caso 3 dovrà controllare solo i vicini sx, usx e u).

Calcolo della Generazione Successiva

Questa parte di codice implementa le regole del Gioco Della Vita per calcolare la generazione successiva.

Il risultato delle regole viene salvato nella posizione della cella corrispondente nella tabella della generazione successiva, indicata da R3.

Le regole si distinguono in due casi principali, il caso in cui la cella iniziale sia viva e il caso in cui sia morta.

regole:

```
beqz r5, regoleVivo   ; caso cella viva
...                   ; caso cella morta
```

Nel caso in cui la cella sia morta, le regole del Gioco Della Vita indicano che se la cella ha 3 vicini, la cella nasce, altrimenti rimane morta (il valore 3 è salvato nel registro R17).

```
...
beq r6, r17, vivi     ; se r6 = 3, scrive 1 nella cella della
                      ; tabella successiva (indicata da r3)
                      ; vivi: sb r1, tavola0(r3)

                      ; r6 != 3, la cella muore
sb r0, tavola0(r3)    ; scrive 0 nella cella della tabella
                      ; successiva (indicata da r3)
```

Nel caso in cui la cella sia viva, le regole indicano che se la cella ha 2 o 3 vicini, la cella rimane viva, altrimenti muore (il valore 2 è salvato in R18).

regoleVivo:

```
beq r6, r17, vivi     ; se r6 = 3, scrive 1 nella cella della
                      ; tabella successiva (indicata da r3)
                      ; vivi: sb r1, tavola0(r3)
beq r6, r18, vivi     ; r6 = 2

                      ; r6 != 2 & r6 != 3, la cella muore
sb r0, tavola0(r3)    ; scrive 0 nella cella della tabella
                      ; successiva (indicata da r3)
j aggiorna            ; fase successiva
```

Finito questo passaggio, otteniamo la generazione successiva della cella corrente nella tavola indicata da R3.

Scorrimento Tavola

Una volta aver rappresentato, calcolato i vicini e calcolato la generazione futura di una cella, passiamo alla cella successiva.

Questa parte di codice consiste nel reimpostare il valore dei vicini a 0 per la prossima cella:

aggiorna:
```
daddi r6, r0, 0
```

Decrementare il contatore di X (R12) per passare alla cella successiva:

```
daddi r12, r12, -1
```

Incrementare i contatori di posizione dell'indirizzo in memoria della cella corrente:

```
daddi r2, r2, 1
```

E della cella corrispondente nella tabella della generazione successiva:

```
daddi r3, r3, 1
```
Caricare il valore della nuova cella in R5:

```
lb r5, tavola0(r2)
```

(Il controllo della validità della posizione della cella viene svolto dopo, il che rende questa operazione superflua nel caso in cui siamo arrivati all'ultimo elemento della tavola. Nonostante ciò, posizionare il caricamento della cella in questo punto consente di evitare dipendenze RAW sul registro R5)

Successivamente il programma controlla in che posizione siamo nella tabella, e verifica se bisogna passare alla riga successiva, o se abbiamo terminato il calcolo della tabella intera, e passa alla fase successiva:

```
bnez r12, controllaStato      ; r12 != 0 → calcola cella
                              ; r12 = 0 → nuova riga
daddi r13, r13, -1            ; decrementa il contatore per Y
daddi r12, r0, 16             ; resetta il contatore per X al
                              ; valore iniziale
bnez r13, controllaStato      ; r13 != 0 → calcola cella
                              ; r13 = 0 → fase successiva
```

Questa parte di codice rappresenta la fine del codice per il ciclo di rappresentazione di una cella, calcolo dei suoi vicini, calcolo della generazione successiva e passaggio alla cella successiva.

Scambio di Tavole

L'ultima parte del codice, consiste nel reimpostare le variabili usate per il calcolo della generazione successiva ai valori iniziali (R12, R13, R2, R3), la pulizia del terminale grafico per la rappresentazione della prossima generazione e lo scambio degli indirizzi di memoria (R2 e R3), così da poter usare la tavola corrente come tavola di destinazione per la generazione successiva, e la precedente tavola di destinazione come tavola corrente da rappresentare e da cui calcolare la generazione successiva.

Per semplicità di rappresentazione e funzionalità per l'utente, il programma ascolta un input dall'utente, così da fermare l'esecuzione fino a che l'utente non prema "Invio". Questa scelta facilita la visualizzazione di ogni generazione, ma allo scopo dell'algoritmo è puramente facoltativa.

scambiaTavola:

```
daddi r29, r0, 8      ; comando per leggere un input (pausa)
sd r29, (r30)         ; passa il comando al terminale

daddi r29, r0, 6      ; comando per pulire il testo
sd r29, (r30)         ; passa il comando al terminale

daddi r29, r0, 7      ; comando per pulire il buffer grafico
sd r29, (r30)         ; passa il comando al terminale

daddi r29, r0, 5      ; comando per disegnare
```

```
daddi r12, r0, 16     ; reimposta il contatore di X a 16
daddi r13, r0, 16     ; reimposta il contatore di Y a 16

daddi r2, r2, -256    ; reimposta l'indirizzo di memoria al
                      ; suo valore iniziale (ci siamo spostati
                      ; di 256 posizioni)

daddi r3, r3, -256    ; reimposta l'indirizzo di memoria al
                      ; suo valore iniziale (ci siamo spostati
                      ; di 256 posizioni)
...
```

Per scambiare effettivamente le tavole, basta scambiare i valori di R2 e R3, che richiede un registro temporaneo per salvare il valore di R2 prima che prenda il valore di R3, così da poter passare il suo valore ad R3 senza che venga perso:

```
...
dadd r7, r0, r2       ; salva r2 in r7
dadd r2, r0, r3       ; salva r3 in r2
dadd r3, r0, r7       ; salva r7 in r3

j controllaStato      ; inizia il calcolo della nuova
                      ; generazione
```

Procedimenti di Ottimizzazione

Per verificare l'efficienza del codice ho verificato le statistiche fornite da WinMIPS riguardo numero di cicli, numero di istruzioni, CPI, numero di stalli (RAW, WAW, WAR, Branch) e dimensione del codice per calcolare una, due e 76 generazioni del Gioco Della Vita basandosi su un pattern il cui comportamento è tipico e indicativo dell'algoritmo implementato:
la tabella di partenza costituisce una "astronave" (struttura ciclica e mobile) che viaggia fino a che non incontra un muro. Arrivata al muro si trasforma in un "aliante" (un altra struttura ciclica e mobile) che viaggia in diagonale, fino ad incontrare anch'essa un muro, ed evolversi in uno quadrato (2x2) stabile (non cambia stato).

In una tabella 16x16, servono 76 generazioni per raggiungere la stabilità di questo sistema.

In nessun momento il codice ha mai avuto stalli di tipo WAW, WAR, strutturali o di Branch Misprediction. Queste statistiche verranno omesse in quanto ad ogni momento sono tutte irrilevanti (0).

Il codice ha passato 6 fasi di ottimizzazione:

1) In questa fase il programma è funzionante ma senza grosse ottimizzazioni, il flusso del programma consiste nello scorrere tutta la tabella per rappresentarla, e poi scorrerla di nuovo per calcolare la generazione successiva:

(1)	1 Generazione	2 Generazioni	76 Generazioni
Cicli	16215	32492	1231444
Istruzioni	11434	22922	868718
CPI	1.418	1.418	1.418
Stalli RAW	3171	6362	240938
Stalli Branch	1606	3204	121784
Dimensione (B)	900	900	900

2) Come prima ottimizzazione, riordinando la posizione delle istruzioni nel codice per la rappresentazione delle celle, possiamo evitare dipendenze di tipo RAW nel passaggio alla cella successiva dopo il decremento della posizione X (stallo in R12), in quanto in questa versione il codice per passare alla cella successiva era contenuto nella fase di rappresentazione.

```
morto:                          ; funzione per gestire il caso di
                                ; rappresentazione di una cella
                                ; morta
...
daddi r10, r10, 1               ; contatore opposto ad r12,
                                ; inizialmente usato per
                                ; rappresentare le celle in
                                ; ordine

daddi r12, r12, -1              ; decremento di X

bnez r12, controllaStato        ; controllo riga
; Stallo RAW in r12
...
                                  |
                                  v

...
daddi r12, r12, -1
daddi r10, r10, 1
bnez r12, controllaStato
...
```

La stessa ottimizzazione è possibile nel codice simmetrico per la rappresentazione di celle vive.

Oltre a questa ottimizzazione, è anche possibile la rimozione di un salto per passare al caso in cui la cella da rappresentare sia morta:

controllaStato:

```
...
bnez r5, vivo          ; caso vivo
j morto                ; caso morto (salto non necessario)


morto:
...
```

```
                       |
                       V
```

```
...
bnez r5, vivo

morto:
...
```

Queste due ottimizzazioni hanno portato questi risultati:

(2)	1 Generazione	2 Generazioni	76 Generazioni
Cicli	15465	30998	1174094
Istruzioni	11187	22431	849771
CPI	1.382	1.382	1.382
Stalli RAW	2915	5850	221482
Stalli Branch	1359	2713	102837
Dimensione (B)	896	896	896

SpeedUp	1 Generazione	2 Generazioni	76 Generazioni
Cicli	-4.63%	-4.60%	-4.66%
Istruzioni	-2.16%	-2.14%	-2.18%
CPI	-2.54%	-2.54%	-2.54%
Stalli RAW	-8.10%	-8.05%	-8.07%
Stalli Branch	-15.38%	-15.32%	-15.56%
Dimensione (B)	-0.44%	-0.44%	-0.44%

3) Inizialmente il programma utilizzava R5 anche per salvare il valore del vicino visitato nella parte di codice per il calcolo dei vicini. Salvare il vicino in un solo registro crea molte dipendenze di tipo RAW, in quanto il registro R5 viene usato per caricare un valore e subito dopo leggerlo, per sommarlo al contatore dei vicini (R6).

Con un solo registro non è possibile organizzare le istruzioni in modo da evitare dipendenze, per questo possiamo usare il registro R8 e alternare il valore dei vicini tra due registri, così da evitare letture subito dopo scritture dello stesso registro:

calcolaVicini:

```
...
daddi r4, r2, -17      ; dsx
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5

daddi r4, r2, -16      ; d
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5

daddi r4, r2, -15      ; ddx
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5

daddi r4, r2, -1       ; dx
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5

daddi r4, r2, 17       ; udx
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5

daddi r4, r2, 16       ; u
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5
```

```
daddi r4, r2, 15      ; usx
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5

daddi r4, r2, -1      ; sx
lb r5, tavola0(r4)
dadd r6, r6, r5
; Stallo RAW in r5

j regole
```

```
daddi r4, r2, -17     ; dsx
lb r5, tavola0(r4)

daddi r4, r2, -16     ; d
dadd r6, r6, r5
lb r8, tavola0(r4)

daddi r4, r2, -15     ; ddx
dadd r6, r6, r8
lb r5, tavola0(r4)

daddi r4, r2, 1       ; dx
dadd r6, r6, r5
lb r8, tavola0(r4)

daddi r4, r2, 17      ; udx
dadd r6, r6, r8
lb r5, tavola0(r4)


daddi r4, r2, 16      ; u
dadd r6, r6, r5
lb r8, tavola0(r4)

daddi r4, r2, 15      ; usx
dadd r6, r6, r8
lb r5, tavola0(r4)
```

```
daddi r4, r2, -1      ; sx
lb r8, tavola0(r4)
dadd r6, r6, r5
dadd r6, r6, r8

j regole
```

La stessa ottimizzazione è applicabile ad ogni altro caso specifico nella tabella, il codice rappresentato è il codice per il calcolo dei vicini nel caso 9, ma la stessa tecnica in ogni caso specifico porta ad decremento di 3, 5, 8 dipendenze RAW per cella calcolata.

(3)	1 Generazione	2 Generazioni	76 Generazioni
Cicli	13590	27248	1031594
Istruzioni	11187	22431	849771
CPI	1.215	1.215	1.214
Stalli RAW	1040	2100	78982
Stalli Branch	1359	2713	102837
Dimensione (B)	896	896	896

SpeedUp	1 Generazione	2 Generazioni	76 Generazioni
Cicli	-12.12%	-12.09%	-12.14%
Istruzioni	-0%	-0%	-0%
CPI	-12.08%	-12.08%	-12.16%
Stalli RAW	-64.32%	-64.10%	-64.34%
Stalli Branch	-0%	-0%	-0%
Dimensione (B)	-0%	-0%	-0%

4) In versioni precedenti, il codice per il calcolo della generazione successiva contava il numero di vicini decrementando il registro R6, e verificando se è uguale a 0 dopo n volte che è stato decrementato.

Questo procedimento richiede molti passaggi che possono essere evitati confrontando il valore di R6 con i valori specifici dell'algoritmo (2 e 3 nel caso di una cella viva, 3 nel caso di una cella morta). Questo approccio richiede un nuovo registro per contenere il valore 2 (il valore 3 era già salvato in R17). Per questo possiamo usare il registro R18 e confrontare R6 con R17 e R18:

regole:

```
...
beqz r6, muori ; caso in cui r6 = 0

daddi r6, r6, -1       ; r6 >= 1

beqz r6, muori ; r6 = 1
; Stallo RAW in r6
daddi r6, r6, -1       ; r6 >= 2

beqz r6, vivi          ; r6 = 2
; Stallo RAW in r6
daddi r6, r6, -1       ; r6 >= 3

beqz r6, vivi          ; r6 = 3
; Stallo RAW in r6
sb r0, tavola0(r3)     ; r6 > 3

j aggiorna
```

```
                              |
                              V
```

```
...
beq r6, r17, vivi      ; r6 = 2
beq r6, r18, vivi      ; r6 = 3
sb r0, tavola0(r3)     ; r6 != 2,3
j aggiorna
```

Ulteriormente nel codice per il passaggio alla cella successiva, si può evitare una dipendenza di tipo RAW in R12 nel controllo della posizione sulla riga, successiva all'incremento del contatore X:

aggiorna:

```
...
daddi r2, r2, 1
daddi r3, r3, 1
daddi r12, r12, -1
bnez r12, calcolaVicini
; Stallo RAW in r12
...
```

|
V

```
...
daddi r12, r12, -1
daddi r2, r2, 1
daddi r3, r3, 1
bnez r12, calcolaVicini
...
```

(4)	1 Generazione	2 Generazioni	76 Generazioni
Cicli	13289	26587	1008610
Istruzioni	11162	22340	847666
CPI	1.191	1.190	1.190
Stalli RAW	768	1536	58368
Stalli Branch	1355	2707	102572
Dimensione (B)	868	868	868

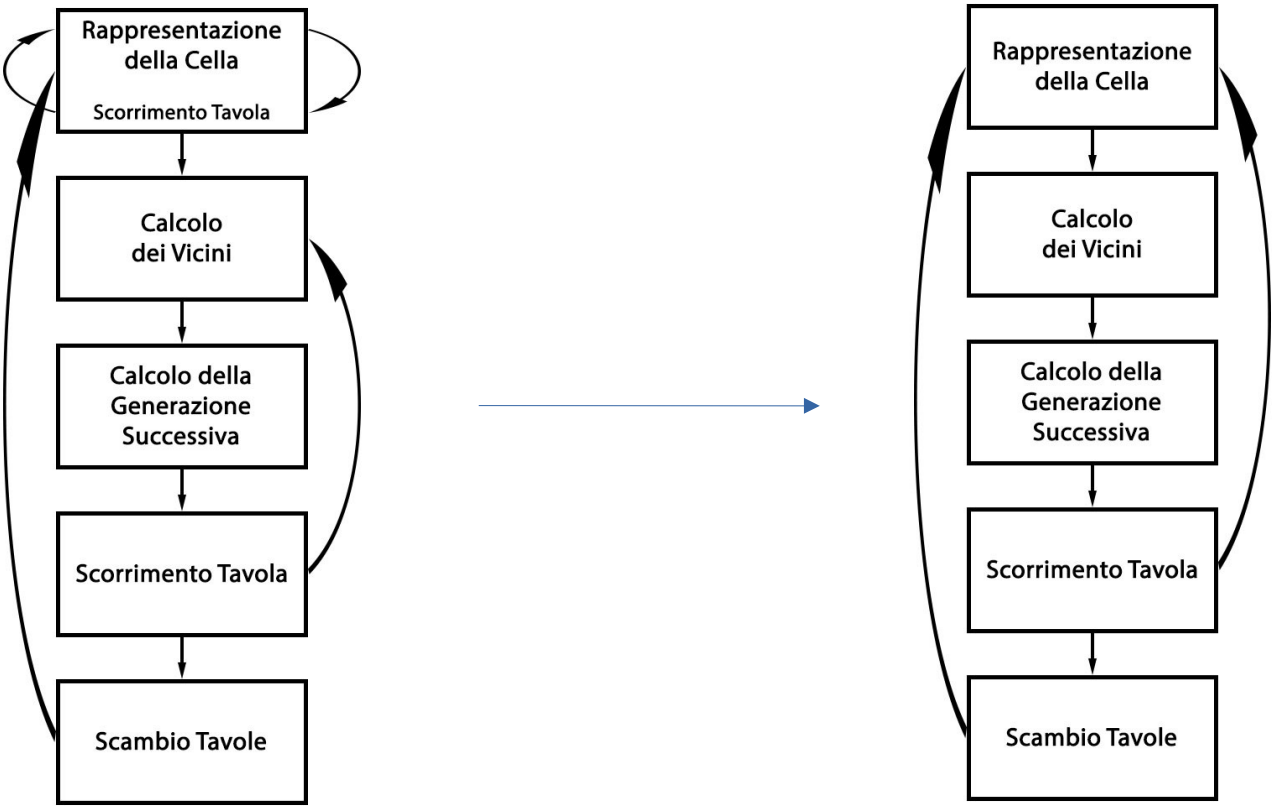
SpeedUp	1 Generazione	2 Generazioni	76 Generazioni
Cicli	-2.21%	-2.43%	-2.23%
Istruzioni	-0.22%	-0.41%	-0.25%
CPI	-1.98%	-2.06%	-1.98%
Stalli RAW	-26.15%	-26.86%	-26.10%
Stalli Branch	-0.29%	-0.22%	-0.26%
Dimensione (B)	-3.13%	-3.13%	-3.13%

5) Questa fase contiene il maggior numero di ottimizzazioni.
Il cambiamento principale consiste nella ristrutturazione del codice, che passa dallo scorrere tutta la tavola, rappresentarla e poi riscorrere tutta la tavola per calcolare la generazione successiva, alla rappresentazione e calcolo della generazione successiva ad ogni iterazione. In questo modo la tavola viene visitata solo una volta per generazione, riducendo il numero di istruzioni per gestire i contatori di posizione (R2, R12, R13), oltre che al codice ridondante per passare a righe successive nella tabella (inizialmente 2 parti del codice compievano lo stesso compito).

Oltre a questo, strutturando il codice in questo modo e prestando attenzione a quando si carica il valore della cella corrente (R5) e quando si controlla il suo valore (per controllare come rappresentare la cella), si può evitare ogni dipendenza di tipo RAW su ogni registro. Per fare questo bisogna però utilizzare un indirizzo diverso da R5 per salvare il valore dei vicini nel calcolo del numero di vicini ad una cella. Possiamo usare il registro R9 al posto di R5.

Possiamo anche evitare di utilizzare contatori specifici per la rappresentazione della tabella: inizialmente il codice utilizzava dei contatori R10 e R11 che si muovono al contrario di R12 e R13, così da rappresentare le celle dal basso verso l'alto e da sinistra verso destra. Essendo una funzionalità superflua, possiamo utilizzare R12 e R13 anche per la rappresentazione delle celle, così da risparmiare molte operazioni per la gestione di due coordinate superflue.

Una ottimizzazione ulteriore consiste nell'eliminare il codice per gestire la rappresentazione di una cella morta, in quanto effettivamente non abbiamo bisogno di gestirla, in quanto non necessita di essere rappresentata.



(5)	1 Generazione	2 Generazioni	76 Generazioni
Cicli	11399	22802	865395
Istruzioni	10050	20113	763329
CPI	1.134	1.134	1.134
Stalli RAW	0	0	0
Stalli Branch	1345	2685	102062
Dimensione (B)	776	776	776

SpeedUp	1 Generazione	2 Generazioni	76 Generazioni
Cicli	-14.22%	-14.24%	-14.20%
Istruzioni	-9.96%	-9.97%	-9.95%
CPI	-4.79%	-4.71%	-4.71%
Stalli RAW	-100.00%	-100.00%	-100.00%
Stalli Branch	-0.74%	-0.81%	-0.50%
Dimensione (B)	-10.60%	-10.60%	-10.60%

6) Per come è strutturato il codice, ci sono casi in cui il flusso di esecuzione viene interrotto da un salto per passare ad un caso specifico, questo avviene se dobbiamo accedere a codice non contiguo in memoria. Quando passiamo da una fase all'altra però c'è sempre un caso in cui non vengono richiesti salti, ovvero il caso definito subito sopra la fase successiva (non richiede salto in quanto il codice da eseguire nella prossima fase è il codice subito sotto in memoria).

Se organizziamo il codice in modo da favorire il flusso privo di salti nei casi più comuni, possiamo evitare molti stalli di tipo Branch:

- Nella fase di rappresentazione, sappiamo che in generale, l'algoritmo del Gioco Della Vita genera molte più celle morte che celle vive. Possiamo ordinare il codice in modo da evitare un salto alla fase successiva in caso di una cella morta, e forzare il salto solo nel caso di una cella viva.

- Nella fase di calcolo dei vicini, il caso 9 è molto favorito rispetto agli altri casi, in quanto in una tavola 16x16, si presenta 196 volte su 256 celle.

- Nella fase di calcolo della generazione successiva, il caso in cui stiamo calcolando una cella precedentemente morta è molto più probabile che una cella viva (di nuovo per la natura dell'algoritmo).

Tutti questi casi però sono più o meno efficaci a seconda della configurazione della tavola (una tavola piena di celle vive sarà più lenta da calcolare), e dalla dimensione della tavola (una tavola 2x2 o 3x3 o 4x4 hanno poche se non nulle istanze di celle di tipo 9).

I dati studiati sono testati su una tavola con la configurazione indicata (astronave + aliante) che contiene molte celle vuote. Con configurazioni diverse si possono ottenere risultati diversi, specialmente adesso che la struttura del codice favorisce alcune configurazioni piuttosto che altre.

Le configurazioni favorite rappresentano comunque i casi generalmente più comuni.

(6)	1 Generazione	2 Generazioni	76 Generazioni
Cicli	9757	19539	738672
Istruzioni	9320	18660	707021
CPI	1.047	1.047	1.045
Stalli RAW	0	0	0
Stalli Branch	433	875	31647
Dimensione (B)	748	748	748

SpeedUp	1 Generazione	2 Generazioni	76 Generazioni
Cicli	-14.40%	-14.31%	-14.64%
Istruzioni	-7.26%	-7.22%	-7.38%
CPI	-7.67%	-7.67%	-7.85%
Stalli RAW	-0.00%	-0.00%	-0.00%
Stalli Branch	-67.81%	-67.41%	-68.99%
Dimensione (B)	-3.61%	-3.61%	-3.61%

In confronto alla prima versione del codice, l'insieme di tutte le ottimizzazioni presentate produce questa differenza di statistiche:

SpeedUp (1)	1 Generazione	2 Generazioni	76 Generazioni
Cicli	-39.83%	-39.87%	-40.02%
Istruzioni	-18.49%	-18.59%	-18.61%
CPI	-26.16%	-26.16%	-26.30%
Stalli RAW	-100.00%	-100.00%	-100.00%
Stalli Branch	-73.04%	-72.69%	-74.01%
Dimensione (B)	-16.89%	-16.89%	-16.89%

Alternative Ottimizzazioni

Al momento il programma usa molta più memoria di quanto serva.
Vengono utilizzati 256B di memoria per tavola, ma ogni Byte contiene solo 1 valore binario, rappresentabile anche con un solo bit.

Si può ridurre la dimensione di memoria utilizzata salvando i valori delle celle in singoli bit (8 celle per Byte), così da ridurre la dimensione della memoria utilizzata da 512B (256B + 256B) a 512bit (64B).

Il problema di questa ottimizzazione è che l'architettura può leggere come minimo 1B dalla memoria, quindi per accedere al valore di una cella singola, dovremmo ogni volta che accediamo ad un area di memoria (contenente mezza riga della tabella), dobbiamo applicare operazioni di shift per accedere al bit specifico che ci interessa.

Questa operazione consiste nello spostare la posizione dei bit a destra di 1 e mettere il Byte contenente le celle in AND con "1".

Per esempio se abbiamo un Byte contenente 8 celle "00011011" e vogliamo sapere il valore della seconda cella contenuta nel Byte, dobbiamo spostare ogni cifra di 1 a destra e moltiplicarlo per 1:

`00011011 → 00001101 AND 00000001 → 00000001`

In questo modo possiamo ottenere il valore interessato, ma richiede almeno N shift a destra, con N = posizione della cella interessata e un istruzione AND per isolare il valore.

Un altra possibile ottimizzazione è l'utilizzo del registro R2 come indicatore di posizione nella tabella:
R2 contiene il valore da sommare all'indirizzo di memoria della tabella per accedere ad una cella specifica.
Conoscendo il valore di R2 è possibile identificare le coordinate X e Y della cella corrente.
Sfruttando R2 per questo scopo, è possibile eliminare i registri R12 (X) e R13 (Y) ed ogni operazione su di essi (incremento e decremento per scorrere le celle).

Il problema in questo caso però è che R2 è un numero puro che rappresenta la posizione nella tabella. Per controllare casi specifici (nel caso del calcolo di vicini, o per rappresentare le celle) bisogna usare operazioni aritmetiche come l'operatore modulo e la divisione:

R2 % 16 == 0: rappresenta il caso in cui siamo in fondo ad una riga della tavola

R2 % 16 == 15: rappresenta il caso in cui siamo all'inizio di una riga della tavola

R2 / 16 <= 1: rappresenta il caso in cui siamo nella prima riga

R2 / 16 >= 15: rappresenta il caso in cui siamo nell'ultima riga

L'operatore modulo non è un operazione elementare, il che richiede più operazioni per essere implementato.

Un opzione per accelerare questo calcolo consiste nel codificare per quali valori di R2 ci troviamo in casi specifici:

per casi specifici come la prima riga e l'ultima riga, basta controllare che R2 sia compreso tra 0 e 15 (per la prima riga) e tra 239 e 255 (per l'ultima riga).
Il problema sono i casi specifici per Y, in cui avremo due valori (per esempio nella seconda riga, le celle ai casi 8 e 4 sono R2 = 16 e R2 = 31).

I valori degli estremi possono ancora essere calcolati senza l'uso del modulo, semplicemente verificando se R2 = n * 16 oppure R2 = n * 16 – 1.

Il che comunque richiede fino a 16 volte un controllo per ogni cella.

Oltre al problema del modulo, si pone anche il problema di come ottenere le coordinate precise per poter rappresentare la cella nella posizione giusta durante la fase di disegno.

Per poter disegnare una cella, dobbiamo conoscere le sue coordinate X e Y.

Se usassimo soltanto R2, dovremmo estrarre le coordinate ad ogni iterazione del programma.

Dividendo il valore di R2 per 16 possiamo ottenere il valore di Y, e il resto della divisione rappresenta il valore di X.

Questo però richiede una calcolo ulteriore per ogni cella, comparabile al numero di istruzioni necessarie tenendo un contatore per X e un contatore per Y.
Una ottimizzazione tecnicamente possibile è lo srotolamento dei cicli:

se invece di compiere salti per ritornare all'inizio del codice dopo lo scorrimento della tavola, riscrivessimo il codice 16 x 16 volte il codice sovrastante di seguito all'ultima istruzione del ciclo per il calcolo completo di una cella, e dopo aver calcolato la tavola intera, ricopiare tutto il codice n-generazioni volte, scambiando le tavole tra ogni generazione, potremmo evitare molti stalli per salti.

Questa soluzione è altamente impraticabile in quanto le dimensioni del codice crescerebbero a dismisura:

Se il codice al momento è 748B, applicare questa tecnica porterebbe il codice ad essere circa 191KB per generazione (circa in quanto tramite questa tecnica, alcuni passaggi (salti) non sono più necessari).

Se volessimo calcolare 76 generazioni come abbiamo fatto nella fase di test delle performance, il codice peserebbe circa 14,5MB!

Il codice sorgente e le fasi di ottimizzazione del codice sono disponibili su GitHub: ProgettoArchitettura