Università degli Studi di Verona – A.A. 2004/2005

Corso di Laurea in Informatica

Elaborato SIS – Architettura degli Elaboratori

Marco Oliani – VR048546
Gionata Pozza – VR041772

Introduzione: il progetto M.A.Y.R.A.

M.A.Y.R.A. (**M**ayra: **A** **Y**outhful **R**obotic **A**uthomatization) è il nome scelto per il progetto del corso di *Architettura degli Elaboratori* riguardante la parte di reti logiche svoltasi durante il secondo Quadrimestre di lezione. Le specifiche di tale progetto sono visibili nel file *specifiche.pdf* incluso nel tarball contenente anche questa relazione.

Il circuito realizzato consiste di un **controllore** e di un **datapath**: esamineremo dettagliatamente entrambi, a cominciare da quest'ultimo e dai singoli elementi che lo compongono.

1 . Datapath

Il datapath ha il compito di calcolare il risultato della formula

$$V_x = V_{x-1} + \Delta t/2 * (A_{x-1} + A_x)$$

ponendo l'uscita a 1 se la velocità V_x appena calcolata è minore della velocità *SPEED* impostata dall'esterno, 0 altrimenti. Per fare tutto questo abbiamo bisogno dei seguenti componenti:

- **due multiplexer a due ingressi di selezione da 8 bit ciascuno**
- **due registri a 8 bit**
- **due sommatori a 8 bit**
- **uno sfhiter a 8 bit**
- **un comparatore a 8 bit**

combinati secondo lo schema visibile in figura 1.1 qui in basso

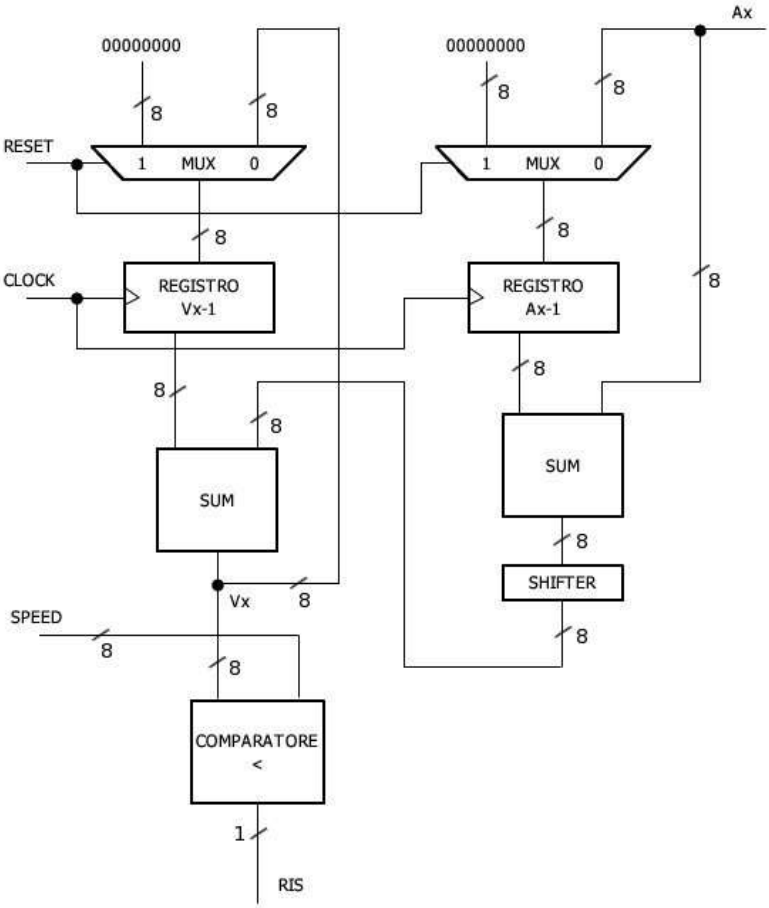

Figura 1.1 *– Schema logico del datapath* di M.A.Y.R.A.

1.1 Multiplexer

I multiplexer, come si può notare dalla figura 1.1, operano una selezione del valore della velocità e dell'accelerazione in base al valore che assume il bit di selezione *RESET* (passato dall'esterno): se questo vale 1, allora viene selezionato l'ingresso 00000000 (valore costante definito in *default_zero.blif*, di cui parleremo più avanti), altrimenti vengono selezionati rispettivamente il valore V_x ottenuto risolvendo la formula nel ciclo di clock corrente e il valore dell'accelerazione A_x che viene invece passato dall'esterno. Salvo che per l'uscita generale del circuito, gli ingressi e le uscite di tutti i componenti da qui in avanti sono da intendersi come cifre intere di 8 bit in complemento a due.

I valori di uscita dai multiplexer vengono memorizzati nei registri V_{x-1} per quel che riguarda la velocità e A_{x-1} per quel che riguarda invece l'accelerazione.

L'implementazione del multiplexer è contenuta nel file *mux8.blif* .

1.2 Registri

I registri V_{x-1} e A_{x-1} hanno il compito di tenere traccia della velocità e dell'accelerazione rilevate al ciclo di clock precedente: sono registri di tipo parallelo / parallelo e sono composti entrambi di 8 registri a un bit. Inizialmente posti a zero, a ogni ciclo memorizzano in ingresso i valori in uscita dai multiplexer sopra descritti e pongono in uscita i valori in precedenza contenuti, valori che rappresenteranno uno dei ingressi dei **sommatori**.

L'implementazione dei due registri è contenuta nei file *reg8.blif* (registro a 8 bit) e *registro.blif* (registro a 1 bit).

1.3 Sommatori

Ora che abbiamo ottenuto i valori V_{x-1} e A_{x-1} è giunto il momento di iniziare a risolvere alcune parti della formula richiesta per l'ottenimento della velocità istantanea del robot. Per farlo ci occorrono due sommatori, uno per sommare $A_{x-1} + A_x$ e l'altro per sommare il valore di uscita dal registro V_{x-1} con il risultato di $\Delta t/2 * (A_{x-1} + A_x)$, che però ancora non abbiamo.

Entrambi i sommatori sono composti da 8 sommatori a un bit.

Il sommatore che si occupa di sommare A_{x-1} e A_x prende in ingresso il valore in uscita dal registro A_{x-1} e l'accelerazione corrente A_x passato invece dall'esterno. Il valore in uscita da questo sommatore viene poi preso in ingresso da uno **shifter** la cui funzione sarà chiara tra poco.

L'altro sommatore, come già detto, prende in ingresso il valore in uscita dal registro V_{x-1} e il risultato ottenuto dal rimanente della formula: il valore ottenuto così in uscita sarà la **velocità istantanea del robot**, la quale dovrà essere confrontata col valore *SPEED* passato dall'esterno.

L'implementazione dei sommatori è contenuta nei file *sommatore8.blif* (per il sommatore a 8 bit) e *sommatore.blif* (per il sommatore a un bit).

1.4 Shifter

Nella traccia viene specificato che il valore Δt è uguale a 1, in quanto la frequenza di clock corrisponde a 1 Hz. Sostituendo questo valore nella formula otteniamo $1/2 * (A_{x-1} + A_x)$, il che

equivale a dividere per due il valore di uscita del sommatore delle accelerazioni A_{x-1} e A_x .

Per ottenere la divisione per due di un numero codificato in binario è sufficiente shiftare tutti i suoi bit di una posizione a destra: nel caso specifico bisogna prestare attenzione al fatto che i numeri trattati sono in complemento a due e che quindi potremmo trovarci di fronte a un numero negativo. Ne consegue che se il valore preso in ingresso dallo shifter presenta un 1 al bit più significativo questo deve essere mantenuto anche dopo che il valore è stato shiftato.

L'output dello shifter contenuto nel datapath di M.A.Y.R.A. viene passato in ingresso al sommatore che calcola la somma tra V_{x-1} e $\Delta t/2 * (A_{x-1} + A_x)$ e di cui abbiamo già parlato in precedenza.

L'implementazione dello shifter è contenuta nel file *shift8.blif*.

1.5 Comparatore

Una volta ottenuto il valore V_x lo dobbiamo comfrontare con il valore *SPEED* che viene passato in input dall'esterno. Per fare ciò abbiamo bisogno di un componente che ci indichi se la velocità corrente del nostro robot è minore di quella voluta accendendo il motore (e ponendo quindi l'uscita del comparatore e dell'intero datapath a 1), oppure spegnerlo nel caso contrario (ponendo quindi l'uscita del datapath a 0).

Le specifiche del progetto dicono solamente che la nostra *SPEED* **non deve essere nulla**: ne consegue, perciò, che il nostro robot possa muoversi in due sensi di marcia, **in avanti**, nel caso di *SPEED* positivo, oppure **in retromarcia**, nel caso di *SPEED* negativo.

Ciò comporta due problematiche: non è sufficiente confrontare cifre con **segno concorde** (entrambe positive o entrambe negative) come si è visto negli esercizi e nelle dispense di laboratorio, ma bisogna necessariamente confrontare anche cifre con **segno discorde** (una positiva e l'altra negativa e viceversa). Non solo: proprio a causa del doppio senso di marcia, i confronti tra V_x e *SPEED* vanno eseguiti tenendo conto del **valore assoluto** di entrambi i numeri (se concordi) e del **cambio di segno** tra il prima e il secondo (se discordi).

La soluzione adottata per risolvere questo problema in SIS può apparire “bruttina” (e forse in realtà lo è), ma è efficace. Partendo dall'implementazione sottostante

```
.model LE8
.inputs A7 A6 A5 A4 A3 A2 A1 A0 B7 B6 B5 B4 B3 B2 B1 B0
.outputs out

.subckt XNOR A=A7 B=B7 X=X7
.subckt XNOR A=A6 B=B6 X=X6
.subckt XNOR A=A5 B=B5 X=X5
.subckt XNOR A=A4 B=B4 X=X4
.subckt XNOR A=A3 B=B3 X=X3
.subckt XNOR A=A2 B=B2 X=X2
.subckt XNOR A=A1 B=B1 X=X1
.subckt XNOR A=A0 B=B0 X=X0

.names A7 A6 A5 A4 A3 A2 A1 A0 X7 X6 X5 X4 X3 X2 X1 X0 out
# Confronto tra segni concordi
0-------0------- 1
-0------10------ 1
--0-----110----- 1
---0----1110---- 1
----0---11110--- 1
```

```
-----0--111110-- 1
------0-1111110- 1
-------011111110 1

# Confronto tra segni discordi
1-------0------- 1
-1------10------ 1
--1-----110----- 1
---1----1110---- 1
----1---11110--- 1
-----1--111110-- 1
------1-1111110- 1
-------111111110 1

.search xnor.blif
.end
```

abbiamo potuto verificare come i confronti dessero il risultato voluto solamente se le cifre erano entrambe positive oppure la prima negativa e la seconda positiva: tutti gli altri confronti davano uscite non corrispondenti al vero.
Il motivo di questa discrepanza è da ricercarsi nel fatto che l'*XNOR* utilizzato in caso di cifre a segno concorde non va bene per cifre di segno discorde e viceversa: lasciando tutte le possibili combinazioni in un unico `.names` è molto probabile che venga presa in considerazione un'uscita sbagliata.

Facciamo un esempio e confrontiamo 6 (`00000110`) e -3 (`11111101`): l'*XNOR* tra i due risulta essere `00000100`. Essendo due numeri di segno discorde procediamo col relativo confronto, che dà esito negativo. Sfortunatamente, però, abbiamo anche le combinazioni che verificano se due numeri di segno concorde sono uguali: ci accorgiamo subito che già alla prima (`0-------0------- 1`) abbiamo la condizione verificata e l'uscita del comparatore è quindi posta a 1. E' evidente che non è questo il risultato voluto, dato che 6 sicuramente non è minore di -3.

Per ovviare a questo inconveniente si è scelto di fare due confronti separati (quindi due `.names`), ponendo il risultato di ognuno in una uscita a un bit a sè stante, come si può notare sotto:

```
.names A7 A6 A5 A4 A3 A2 A1 A0 X7 X6 X5 X4 X3 X2 X1 X0 out1
# Confronto tra segni concordi
0-------0------- 1
-0------10------ 1
--0-----110----- 1
---0----1110---- 1
----0---11110--- 1
-----0--111110-- 1
------0-1111110- 1
-------011111110 1

.names A7 A6 A5 A4 A3 A2 A1 A0 X7 X6 X5 X4 X3 X2 X1 X0 out2
# Confronto tra segni discordi
1-------0------- 1
-1------10------ 1
--1-----110----- 1
---1----1110---- 1
----1---11110--- 1
-----1--111110-- 1
------1-1111110- 1
-------111111110 1
```

A seconda del valore delle uscite `out1` e `out2` e del segno dei due numeri da confrontare avremo il risultato corretto della comparazione. Per fare questo c'è bisogno di un terzo `.names` definito come segue:

```
.names A7 B7 out1 out2 out
# Vengono presi in ingresso i segni delle due cifre da
# confrontare (rappresentati dal bit più significativo di
# ogni cifra) e le due uscite derivate dai .names prece-
# denti, mentre l'uscita .out è quella dell'intero compa-
# ratore
0010 1
10-- 1
1110 1
```

Tutti i confronti non specificati danno come uscita 0 e quindi non vengono menzionati nel `.names`. (si noti che i due *don't care* in corrispondenza della seconda condizione del `.names` sono giustificati dal fatto che un numero di segno negativo è sempre minore di un numero di segno positivo).

Fatto ciò bisogna occuparsi del confronto **in valore assoluto** e del **cambio di segno**: per farlo è semplicemente necessario modificare il `.names` sopra definito ponendo a 1 alcune delle uscite precedentemente impostate a 0 e viceversa

```
.names A7 B7 out1 out2 out
# Vengono presi in ingresso i segni delle due cifre da
# confrontare (rappresentati dal bit più significativo di
# ogni cifra) e le due uscite derivate dai .names prece-
# denti, mentre l'uscita .out è quella dell'intero compa-
# ratore
0010 1
1101 1
01-- 1
10-- 1
```

Ora il comparatore è terminato ed è pronto ad assolvere il compito per il quale è stato progettato.
Si rammenta ancora una volta che il valore di uscita del comparatore è anche il valore di uscita dell'intero datapath e che questo valore verrà passato come ingresso al **controllore**, di cui parleremo in dettaglio più avanti in questa relazione.

L'implementazione del comparatore è contenuta nei file *le8.blif* (il comparatore vero e proprio) e *xnor.blif* (per quello che riguarda l'*XNOR*).

1.6 Costanti

Come già accennato in precedenza, per gli ingressi dei multiplexer abbiamo bisogno di una costante del valore zero su 8 bit.
Questa costante è definita nel file *default_zero.blif* .

1.7 Testing

E' giunto ora il momento di testare l'intero datapath per vedere se il nostro lavoro è corretto.
Apriamo quindi SIS e leggiamo il file *datapath4.blif* che contiene tutti i componenti sopra descritti.

```
$ sis
UC Berkeley, SIS 1.3
        (compiled 25-Jan-02 at 10:59 AM)
sis>read_blif datapath4.blif
sis>
```

Simuliamo ora dei possibili input: come da richiesta della traccia, l'accelerazione deve essere costante e avere valore 10 (`00001010`) quando il motore è acceso e -10 (`11110110`) quando il motore è spento. Vogliamo che il nostro robot abbia una velocità pari a 20 (`00010100`), pertanto

```
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 0000101000000101
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 0000101000001111
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 0000101000011001
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 1111011000011001
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 1111011000001111
sis>
```

che è esattamente il comportamento che ci aspettavamo (il comportamento per accelerazione e velocità negative è il medesimo, lo si potrà verificare anche quando avremo a che fare con il progetto completo).
Testiamo anche il *RESET*, il quale deve porre i registri V_{x-1} e A_{x-1} al valore zero, come da traccia

```
sis> simulate 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 0000000000000000
sis>
```

Il comportamento è quindi corretto.
(si fa notare come in presenza del bit di reset settato a 1 i valori di accelerazione e velocità sono ininfluenti)

2 . Controllore

2.1 Funzionamento generale

Il **controllore** ha il compito di verificare che le condizioni che determinano l'accensione o lo spegnimento del motore del nostro robot permangano per due cicli di clock, tenendo conto anche del *RESET* che viene passato dall'esterno e che, in caso valga 1, deve riportare immediatamente l'uscita del controllore al valore 0. In ingresso, oltre a *RESET*, è presente anche il valore di uscita dal datapath nel ciclo di clock corrente.

Per realizzare tutto questo si è deciso di implementare una **FSM a tre stati**, lo schema rappresentato in figura 2.1:

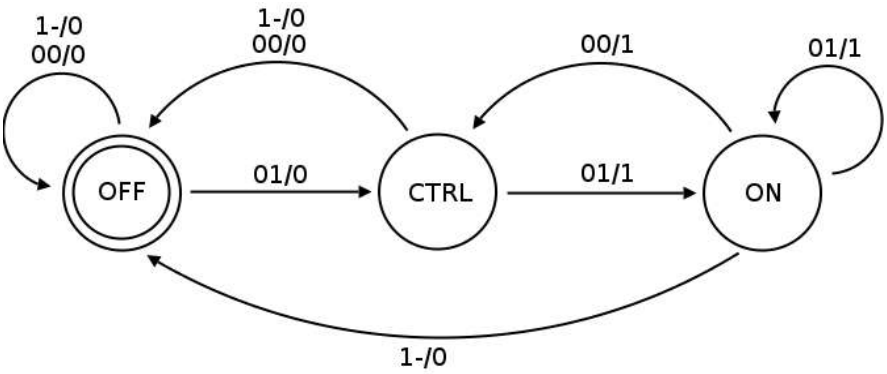

Figura 2.1: la FSM relativa al controllore di M.A.Y.R.A.

Abbiamo deciso di definire **OFF** come **stato di reset** in quanto alla partenza il motore deve essere spento. I due bit presenti sugli archi rappresentano rispettivamente il bit di *RESET* e il valore in uscita dal datapath: se al primo ciclo di clock il valore in ingresso del controllore vale **01** allora si cambia di stato e si passa in **CTRL**, uno stato intermedio che ha il compito di verificare se al prossimo ciclo di clock il motore deve essere acceso o deve rimanere spento (attenzione: l'uscita del controllore rimane ancora a 0!!). Se nel ciclo successivo si ha ancora 01 in ingresso allora lo stato cambia nuovamente e passa nello stato di **ON**, che sta a significare che il motore di M.A.Y.R.A. deve essere acceso (si deve avere, cioè, l'uscita uguale a 1).
In tutti gli altri casi (salvo reset del circuito) si rimane o si passa allo stato **OFF** e l'uscita del controllore viene lasciata a 0.

Dallo stato di **ON**, analogamente, bisogna attendere sempre due cicli di clock per far sì che il motore venga spento e si passi quindi nello stato di **OFF**. Se si ha come ingresso **00** allora lo stato cambia e passa da ON a **CTRL**, il quale si aspetta un'altra sequenza 00 per passare a OFF, mantenendo però l'uscita 1 e mantenendo acceso il motore. Se questa si presenta allora lo stato cambia da CTRL a **OFF** e si pone l'uscita del controllore a 0.
In tutti gli altri casi (salvo reset del circuito) si rimane o si passa allo stato **ON** e l'uscita del controllore viene lasciata a 1.

Il caso di reset del circuito è un caso particolare, in quanto, come già detto, bisogna porre l'uscita del controllore a 0 senza aspettare due cicli di clock. Ciò comporta che in qualsiasi stato ci si trovi, se in input si ha la sequenza **1-** lo stato deve passare immediatamente a **OFF**, che è appunto il mio stato di reset, e l'uscita del controllore è pari a **0**.
Si noti che in caso di *RESET* uguale a 1 non ha importanza quale valore esca dal datapath, quindi viene messo un *don't care*.

2.2 Implementazione del controllore in SIS

Dopo aver fatto uno schema cartaceo del controllore si è passati alla sua definizione utilizzando SIS. Per farlo si è dapprima creato un file (*controllore_reset_3stati.blif*) così definito:

```
.model CONTROLLORE
.inputs RESET RIS
.outputs GO

.start_kiss
.i 2
.o 1
.p 9
.s 3
.r OFF

00 OFF OFF 0
1- OFF OFF 0
01 OFF CTRL 0
01 CTRL ON 1
00 CTRL OFF 0
1- CTRL OFF 0
01 ON ON 1
00 ON CTRL 1
1- ON OFF 0
.end_kiss

.end
```

Nella nostra FSM, come si pò facilmente notare, sono presenti due input (indicati da `.i 2` all'interno della direttiva `.start_kiss` e corrispondenti nella fattispecie a *RESET* e *RIS* – il valore di uscita dal datapath) un output (`.o 1`, che corrisponde a *GO*), nove transizioni (`.p 9`), tre stati attraversati (`.s 3`) e lo stato di reset OFF (`.r OFF`).
Di seguito sono definite le transizioni, con gli ingressi, lo stato presente, lo stato prossimo e il relativo valore di uscita.

Da questo "scheletro" di FSM si è poi passati ad assegnare gli stati e a costruire la rete relativa attraverso i comandi SIS `state_assign jedi` e `stg_o_network`: il risultato di queste due operazioni è visibile all'interno del file *controllore_prova.blif* .

2.3 Testing

Verifichiamo il corretto funzionamento del nostro controllore. Lanciamo quindi SIS e leggiamo il file relativo al controllore stesso:

```
$ sis
UC Berkeley, SIS 1.3
        (compiled 25-Jan-02 at 10:59 AM)
sis>read_blif controllore_prova.blif
sis>
```

Proviamo ad accendere il nostro motore:

```
sis> simulate 0 1

Network simulation:
Outputs: 0
Next state: 00

STG simulation:
Outputs: 0
Next state: CTRL (00)

sis> simulate 0 1

Network simulation:
Outputs: 1
Next state: 11

STG simulation:
Outputs: 1
Next state: ON (11)

sis>
```

Il nostro robot si è messo in movimento esattamente dopo due cicli di clock, come volevamo. Ora proviamo a spegnere il motore:

```
sis> simulate 0 0

Network simulation:
Outputs: 1
Next state: 00

STG simulation:
Outputs: 1
Next state: CTRL (00)

sis> simulate 0 0

Network simulation:
Outputs: 0
Next state: 10

STG simulation:
Outputs: 0
Next state: OFF (10)
```

Anche in questo caso il funzionamento è corretto, perchè vengono attesi due cicli di clock prima di spegnere il motore.

Come ultima dimostrazione riaccendiamo il motore e resettiamo il circuito mentre ci troviamo ancora nello stato ON:

```
sis> simulate 0 1

Network simulation:
Outputs: 0
```

```
Next state: 00

STG simulation:
Outputs: 0
Next state: CTRL (00)

sis> simulate 0 1

Network simulation:
Outputs: 1
Next state: 11

STG simulation:
Outputs: 1
Next state: ON (11)

sis> simulate 1 0

Network simulation:
Outputs: 0
Next state: 10

STG simulation:
Outputs: 0
Next state: OFF (10)

sis>
```

Abbiamo così dimostrato il corretto funzionamento del controllore anche in presenza del reset.

3 . L'intero circuito

La nostra **FSMD**, ovvero l'intero nostro circuito, avrà la fisionomia visibile in figura 3.1

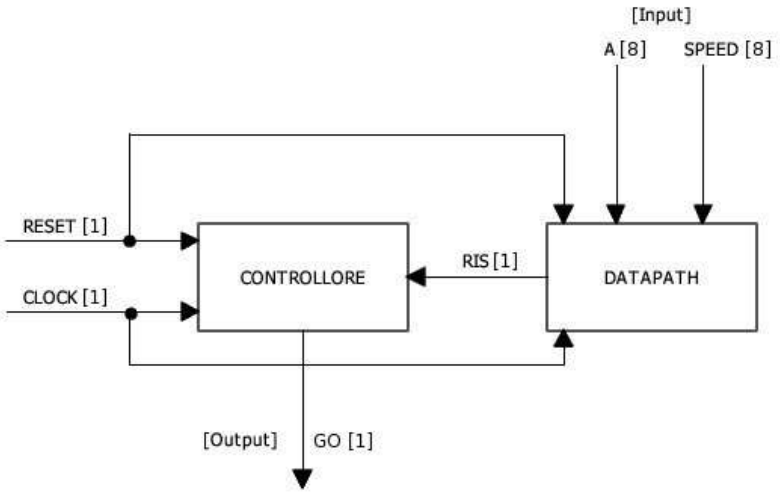

Figura 3.1: *schema esplicativo di M.A.Y.R.A.*

Lo schema è abbastanza elementare. La FSMD riceve in ingresso dall'esterno l'indicazione sul reset del circuito e i valori di accelerazione e velocità massima, mentre all'esterno restituisce l'indicazione di accensione/spegnimento del motore. La comunicazione tra controllore e datapath è data dall'uscita del datapath stesso, la quale fungerà da input per il controllore.

Da notare che M.A.Y.R.A. è una **macchina a stati finiti sincrona** in quanto sensibile all'andamento del clock.

Si potrebbe obiettare sul fatto che vi è comunicazione solamente dal datapath al controllore e non viceversa: questa comunicazione, in effetti, avrebbe potuto esserci dando l'indicazione di reset del circuito solamente al controllore, il quale a suo volta l'avrebbe comunicata al datapath affinchè questo ponesse a zero i propri registri.Questa scelta non è stata fatta perchè, oltre ad essere equivalente a quella proposta, è anche più difficile da realizzare e implementare in SIS.

3.1 Implementazione della FSMD in SIS

La nostra FSMD è implementata in SIS richiamando semplicemente il controllore e il datapath analizzati in precedenza. Essa riceve come ingressi i valori di reset, accelerazione e velocità massima che il nostro robot deve raggiungere (ingressi che, come detto, verranno passati al datapath e, per quel che riguarda il reset, anche al controllore), mentre l'uscita *GO* è derivata dall'uscita del controllore.

La FSMD è contenuta nel file *robot.blif*.

3.2 Testing

Dopo aver verificato il corretto funzionamento del datapath e del controllore ora dobbiamo verificare il corretto funzionamento della FSMD. Apriamo SIS e leggiamo la nostra FSMD nel solito modo che abbiamo già visto:

```
$ sis
UC Berkeley, SIS 1.3
        (compiled 25-Jan-02 at 10:59 AM)
sis>read_blif robot.blif
sis>
```

Accendiamo il nostro motore e lasciamolo accelerare fino al raggiungimento della velocità massima impostata (sempre 20):

```
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 000000101000000101
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 110000101000001111
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 000000101000011001
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 100000101000100011
sis>
```

Osservando i valori contenuti in “Next state” possiamo avere conferma del corretto funzionamento del circuito: i primi due bit indicano lo stato, i successivi otto il valore dell'accelerazione e i restanti la velocità del robot.
Iniziamo a frenare. Si ricorda che in caso di frenata il valore dell'accelerazione diventa -10

```
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 101111011000100011
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 101111011000011001
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 001111011000001111
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011000000101
```

Riaccelereriamo e resettiamo il circuito:

```
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 110000101000000101
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 110000101000001111
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 000000101000011001
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 100000101000100011
sis> simulate 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 100000000000000000
sis>
```

E ora testiamo il nostro robot in retromarcia. Perchè si abbia la retromarcia è necessario che dall'esterno vengano impostate accelerazione e velocità massima negative: per comodità daremo accelerazione uguale a -10 e velocità uguale a -20 e partiremo dallo stato di reset, a motore spento e robot fermo:

```
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 001111011011111011
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011011110001
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 001111011011100111
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 101111011011011101
sis>
```

Come si può notare, il funzionamento è il medesimo. Ora freniamo, e in questo caso la nostra accelerazione varrà 10:

```
sis> simulate 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 100000101011011101
sis> simulate 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 100000101011100111
sis> simulate 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 000000101011110001
sis> simulate 0 0 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 110000101011111011
sis>
```

La retromarcia è in grado di funzionare anche senza che sia necessario resettare il circuito, anche se è fortemente consigliato farlo. Il comportamento è il seguente: per cambiare repentinamente senso di marcia è necessario invertire sia il valore dell'accelerazione, sia il valore della velocità massima (e.g. se si ha accelerazione 10 e velocità 20 e si vuole invertire il senso di marcia allora si dovranno settare come accelerazione -10 e come velocità -20): se il motore in quel momento è acceso il suo senso di marcia viene invertito (il motore viene acceso in senso contrario), se è spento vengono attesi due cicli di clock prima che questo venga riacceso.

Di seguito le simulazioni: la prima, che inverte il senso di marcia a motore già acceso

```
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 000000101000000101
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 110000101000001111
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 000000101000011001
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011000011001
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011000001111
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0
```

```
Network simulation:
Outputs: 1
Next state: 111111011000000101
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011011111011
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011011110001
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 001111011011100111
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 101111011011011101
sis>
```

e la seconda, che inverte il senso di marcia a motore spento

```
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 000000101000000101
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 110000101000001111
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 000000101000011001
sis> simulate 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 100000101000100011
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 001111011000100011
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011000011001
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
```

```
Next state: 111111011000001111
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011000000101
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011011111011
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011011110001
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 1
Next state: 001111011011100111
sis> simulate 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 0

Network simulation:
Outputs: 0
Next state: 101111011011011101
sis>
```

Si controllino sempre i valori dei registri e dello stato prossimo per verificare che il comportamento del circuito sia sempre quello voluto.

Dulcis in fundo, esiste anche un controllo all'accensione del circuito che avvisa l'utente (sempre dopo due cicli di clock) di accendere il motore qualora egli abbia sì impostato una velocità positiva (negativa), ma abbia erroneamente dato accelerazione negativa (positiva):

```
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 0
Next state: 001111011011111011
sis> simulate 0 1 1 1 1 0 1 1 0 0 0 0 1 0 1 0 0

Network simulation:
Outputs: 1
Next state: 111111011011110001
sis>
```

4 . Ottimizzazione del circuito

Il circuito finora analizzato non era ottimizzato. Passeremo ora alla ottimizzazione dei componenti del circuito in modo da ridurne per quanto possibile nodi letterali, area e ritardo. Come da richiesta, poi, il circuito sarà mappato sulla libreria tecnologica *synch.genlib*.
Per ottimizzare faremo largo uso dello `script.rugged` e, per quanto riguarda il controllore, anche del comando SIS `state_minimize stamina` per ridurre il numero degli stati del nostro circuito.
Per visualizzare le statistiche si userà il comando SIS `print_stats`.

Andremo ora ad operare sui componenti del datapath: i componenti che non sono presenti in questa sezione (registri, multiplexer e shifter) si intendono non ottimizzabili, quindi vengono omessi.

4.1 Ottimizzazione dei sommatori

Carichiamo il nostro sommatore in SIS e viualizziamone le statistiche:

```
sis> read_blif sommatore8.blif
sis> print_stats
SOMMATORE8      pi=17   po= 9   nodes= 24       latches= 0
lits(sop)= 112
```

Dopo la chiamata allo script.rugged abbiamo quanto segue:

```
sis> source script.rugged
sis> print_stats
SOMMATORE8      pi=17   po= 9   nodes= 24       latches= 0
lits(sop)=  96
```

Abbiamo ottenuto un risparmio di 16 letterali e il numero di nodi è rimasto invariato.
Si fa presente che minimizzare l'intero sommatore a 8 bit o i singoli sommatori a un bit porta allo stesso risultato.

4.2 Ottimizzazione del comparatore

Carichiamo il comparatore in SIS e anche per esso visualizziamo le statistiche:

```
sis> read_blif le8.blif
sis> print_stats
LE8             pi=16   po= 1   nodes= 11       latches= 0
lits(sop)= 132
```

La situazione dopo l'ottimizzazione è la seguente:

```
sis> source script.rugged
sis> print_stats
LE8             pi=16   po= 1   nodes= 14       latches= 0
lits(sop)=  63
```

Notiamo che, a fronte di un leggero aumento del numero di nodi nel circuito è più che dimezzato il numero di letterali impiegati.

4.3 Ottimizzazione del datapath

Una volta ottimizzati i singoli componenti c'è bisogno di ottimizzare tutto l'insieme, quindi l'intero datapath. Prima, però, è bene vedere com'era la situazione prima delle ottimizzazioni appena effettuate:

```
sis> read_blif datapath4.blif
sis> print_stats
DATAPATH        pi=17   po= 1   nodes= 92       latches=16
lits(sop)= 428
```

Questa è invece la situazione che si ha con i singoli componenti ottimizzati:

```
sis> print_stats
DATAPATH        pi=17   po= 1   nodes= 95       latches=16
lits(sop)= 327
```

Notiamo già una buona ottimizzazione, ma si può far meglio ottimizzando anche l'intero datapath. Questi sono i risultati ottenuti:

```
sis> source script.rugged
sis> print_stats
DATAPATH        pi=17   po= 1   nodes= 65       latches=16
lits(sop)= 251
```

Abbiamo quindi risparmiato quasi 30 nodi e poco meno di 200 letterali rispetto alla situazione iniziale.

4.4 Ottimizzazione del controllore

Per ottimizzare il controllore dobbiamo dapprima ridurre il numero di stati attraversati. Queste le statistiche del controllore non ancora minimizzato e ottimizzato:

```
sis> read_blif controllore_prova.blif
sis> print_stats
CONTROLLORE     pi= 2   po= 1   nodes=  3       latches= 2
lits(sop)=  20  #states(STG)=   3
```

Dobbiamo ora minimizzare il numero di stati

```
 sis> state_minimize stamina
 Running stamina, written by June Rho, University of
Colorado at Boulder
```

```
Number of states in original machine : 3
Number of states in minimized machine : 3
```

In questo caso il numero di stati era già minimo, il che significa che non erano ridondanti. Se avessimo definito degli stati ridondanti, l'algoritmo *stamina* li avrebbe trovati e ridotti a un solo stato.

Dopo la minimizzazione è necessario riassegnare riassegnare gli stati e creare la rete

```
sis> state_assign jedi
Running jedi, written by Bill Lin,  UC Berkeley
sis> stg_to_network
```

Dopo di che minimizzo con lo `script.rugged` al fine di ridurre nodi e letterali:

```
sis> source script.rugged
sis> print_stats
CONTROLLORE     pi= 2   po= 1   nodes=  3       latches= 2
lits(sop)=  11  #states(STG)=   3
```

Come si può vedere abbiamo nove letterali in meno rispetto al controllore non ottimizzato.

4.5 Ottimizzazione della FSMD e Mapping tecnologico

Non resta che ottimizzare l'intero circuito. Per prima cosa vediamo di quanti nodi e letterali è composto il circuito non ottimizzato in nessuna delle sue parti:

```
sis> read_blif robot.blif
sis> print_stats
ROBOT           pi=17   po= 1   nodes= 95       latches=18
lits(sop)= 448
```

Mappando il tutto sulla libreria tecnologica *synch.genlib* otteniamo questi valori in termini di area e ritardo:

```
sis> read_library synch.genlib
sis> map -W -m 0 -s
>>> before removing serial inverters <<<
# of outputs:          19
total gate area:       6984.00
maximum arrival time: (44.40,44.40)
maximum po slack:     (-4.00,-4.00)
minimum po slack:     (-44.40,-44.40)
total neg slack:      (-324.20,-324.20)
# of failing outputs:  19
>>> before removing parallel inverters <<<
# of outputs:          19
total gate area:       6952.00
maximum arrival time: (44.40,44.40)
maximum po slack:     (-4.00,-4.00)
```

```
minimum po slack:     (-44.40,-44.40)
total neg slack:      (-324.40,-324.40)
# of failing outputs:  19
# of outputs:          19
total gate area:       6456.00
maximum arrival time: (44.20,44.20)
maximum po slack:     (-4.00,-4.00)
minimum po slack:     (-44.20,-44.20)
total neg slack:      (-323.40,-323.40)
# of failing outputs:  19
```

Vediamo ora di quanti nodi e letterali si compone il circuito non ottimizzato, ma avente controllore e datapath ottimizzati:

```
sis> print_stats
ROBOT           pi=17   po= 1   nodes= 68       latches=18
lits(sop)= 262
```

Rispetto alla situazione precedente abbiamo una forte riduzione di nodi e letterali. In termini di area e ritardo, il mapping risulta essere il seguente:

```
sis> read_library synch.genlib
sis> map -W -m 0 -s
>>> before removing serial inverters <<<
# of outputs:          19
total gate area:       5728.00
maximum arrival time: (53.00,53.00)
maximum po slack:     (-4.60,-4.60)
minimum po slack:     (-53.00,-53.00)
total neg slack:      (-390.60,-390.60)
# of failing outputs:  19
>>> before removing parallel inverters <<<
# of outputs:          19
total gate area:       5696.00
maximum arrival time: (53.20,53.20)
maximum po slack:     (-4.60,-4.60)
minimum po slack:     (-53.20,-53.20)
total neg slack:      (-392.20,-392.20)
# of failing outputs:  19
# of outputs:          19
total gate area:       5360.00
maximum arrival time: (52.60,52.60)
maximum po slack:     (-4.60,-4.60)
minimum po slack:     (-52.60,-52.60)
total neg slack:      (-387.80,-387.80)
# of failing outputs:  19
```

A fronte di un aumento del ritardo abbiamo ottenuto una riduzione abbastanza consistente dell'area Ora ottimizziamo l'intero circuito con *script.rugged*

```
sis> source script.rugged
sis> print_stats
ROBOT           pi=17   po= 1   nodes= 67       latches=18
lits(sop)= 269
```

Vediamo come i letterali siano aumentati di pochissimo, mentre il numero di nodi è drasticamente diminuito. In termini di mapping abbiamo quanto sotto:

```
sis> read_library synch.genlib
sis> map -W -m 0 -s
>>> before removing serial inverters <<<
# of outputs:          19
total gate area:       5304.00
maximum arrival time: (64.40,64.40)
maximum po slack:     (-4.20,-4.20)
minimum po slack:     (-64.40,-64.40)
total neg slack:      (-480.20,-480.20)
# of failing outputs:  19
>>> before removing parallel inverters <<<
# of outputs:          19
total gate area:       5288.00
maximum arrival time: (64.60,64.60)
maximum po slack:     (-4.20,-4.20)
minimum po slack:     (-64.60,-64.60)
total neg slack:      (-481.20,-481.20)
# of failing outputs:  19
# of outputs:          19
total gate area:       5192.00
maximum arrival time: (64.40,64.40)
maximum po slack:     (-4.20,-4.20)
minimum po slack:     (-64.40,-64.40)
total neg slack:      (-480.20,-480.20)
# of failing outputs:  19
```

Abbiamo ulteriormente ridotto l'area, anche se pagando un sovrapprezzo in termini di ritardo.

I file con i componenti ottimizzati si trovano nella directory *progetto_ottimizzato* all'interno del tarball che contiene anche questa relazione.

5 . Conclusioni

Visti i risultati seguiti alle varie ottimizzazioni e ai vari mapping, riteniamo che il miglior compromesso area / ritardo sia quello in cui sono ottimizzati controllore e datapath, ma **non** l'intero circuito. Rispetto all'ottimizzazione completa, infatti, abbiamo un leggero incremento dell'area (5360.00 contro 5192.00) che però viene compensato da un ritardo decisamente inferiore (64.40 contro 52.60). Ci sentiamo di dire che quella trovata è la giusta soluzione di compromesso tra area e ritardo per il nostro progetto.