SIS

Ho diviso questa dispensa in 6 capitoli come le 6 lezioni tenute dal professore in Laboratorio.

Questo file è stato creato e revisionato il 08/03/2021.

Qualsiasi modifica che verrà effettuata dovrà essere inserita dopo l'ultima pagina, alla fine. Così facendo, l'utente potrà rendersi conto degli aggiornamenti e delle modifiche che sono state apportate. È gradito, ovviamente, anche un nickname di chi ha effettuato le modifiche, una sorta di *firma*.

Questa dispensa è stata creata per dare un piccolo supporto o, meglio ancora, per essere spunto di ripasso prima della discussione con il professore riguardo l'elaborato SIS. Non intendo sostituire eventuali testi del professore.

Grazie dell'attenzione, buono studio o ripasso.

1 – Introduzione a SIS

1.1 – Che cos'è SIS?

SIS è un programma (*software*) che implementa l'algoritmo di *Quine – McCluskey* (in parte) per effettuare la minimizzazione di circuiti digitali. Dunque, quest'ultimi sono descritti all'interno di SIS seguendo determinate regole come i classici linguaggi di programmazione (C, C++, Java, …).

È così necessario utilizzare una distribuzione Linux per utilizzare SIS? Sì. Alcuni nostri colleghi provarono a costruire una versione Windows di questo *software*, ma è altamente sconsigliato visti i vari problemi. Certamente è possibile installare SIS su WSL ed utilizzare Visual Studio Code per utilizzare il terminale integrato.

In ogni caso, il professore consiglia l'utilizzo di una distribuzione Linux così da familiarizzare con questo Sistema Operativo (S.O.) che ci servirà anche per gli anni seguenti.

1.2 – Come installarlo?

Per installarlo basta avviare il terminale di Linux e scrivere la seguente riga di codice:

$$./\text{sis} - \text{installer.sh}$$

Una volta installato, aprire nuovamente il terminale e scrivere:

$$\text{gedit } .\text{bashrc}$$

Successivamente a questo comando, un file si aprirà sul terminale. Andiamo all'ultima riga di codice di questo file, premere due volte invio così da scrivere sotto l'ultima riga di codice e aggiungere questa riga di codice:

$$\text{PATH} = "/\text{usr/local/sis} - 1.3.6/\text{bin}/: \$\text{PATH}"$$

A questo punto basterà scrivere sis nel terminale per eseguire il software.

1.3 – Primo codice in SIS

Prima di spiegare la scrittura di un codice in SIS, è fondamentale ricordare che il caricamento di un programma in SIS avviene tramite la dicitura:

```
read_blif  nome_file
```

Questo è un esempio di un file su SIS:

```
. model implicazione
. inputs a b
. outputs c
. names a b c
00 1
01 1
11 1
```

Nella prima riga abbiamo la keyword . model che ha l'obbiettivo di "etichettare" il componente.

Nella seconda riga abbiamo la lista dei bit di input grazie alla keyword . inputs. In questo caso avremo il bit a e b come input della macchina.

Nella terza riga abbiamo gli output, in questo caso solo c, definiti con . outputs.

Infine, abbiamo la tabella di verità definita tramite il . names. Inseriamo i bit di input, quindi a e b, inseriamo il bit di uscita, in questo caso c. Successivamente scriviamo **SOLO** l'On-Set o l'Off-Set poiché SIS calcolerà l'uno o l'altro a seconda dell'insieme che gli abbiamo descritto. In questo caso, con la combinazione 00, 01 e 11 avremo 1 come uscita.

ATTENZIONE!

La scrittura della tabella di verità diversamente, potrebbe causare un miglioramento, quindi una riduzione dei letterali, oppure un peggioramento, un incremento dei letterali. Prendendo l'esempio appena creato, se scrivessimo la tabella di verità in questo modo: 10 0 (quindi considerando solo l'Off-Set), ci troveremo con 4 letterali in meno.

Per salvare il file, dovremo rinominarlo come di seguito, stando attenti all'estensione del file:

```
nomefile. blif
```

E sul terminale scriveremo:

```
read_blif  nomefile. blif
simulate 0 0
```

Il comando simulate ci permetterà di simulare la macchina e di verificare la corretta esecuzione.

1.4 – Visualizzazione statistiche circuito

Con il comando:

print_stats

Verificheremo le statistiche del circuito. In particolare, quante porte utilizza il nostro circuito in input, quindi pi (*primary inputs*); quante porte utilizza in output, po (*primary outputs*); quanta area occupa il circuito (*nodes*); e quanti letterali costa il circuito, *lits*. È utile ricordare che per calcolare l'area e il ritardo del circuito relativo all'espressione considerata occorre disegnarlo utilizzando solo porte NOT, e AND/OR a due ingressi. L'area è data dal numero di porte utilizzate mentre per il ritardo si deve individuare il numero di porte attraversate sul cammino più lungo tra uno qualunque degli ingressi e una qualunque delle uscite (cammino critico).

1.5 – Combinazione di input e output

In SIS è possibile anche definire più di una tabella di verità:

```
. model implicazione
. inputs a b
. outputs c d
. names a b c
00 1
01 1
. names a b d
11 1
10 1
00 1
```

2 – Ottimizzazione esatta di circuiti combinatori a due livelli e funzioni non completamente specificate

2.1 – Minimizzazione di circuiti combinatori a 2 livelli tramite SIS

Per minimizzare un circuito combinatorio a 2 livelli, grazie a SIS, si utilizzerà il comando:

full_simplify

Ovviamente dopo aver caricato il modello da ottimizzare.

Dopo aver utilizzato l'algoritmo di ottimizzazione, potremo andare a vedere gli effetti grazie al comando:

write_eqn

Sia prima che dopo aver utilizzato il comando full_simplify. Questo comando ci permetterà di osservare l'equazione booleana corrispondente al file .blif. Perciò è consigliabile di eseguirlo sia prima che dopo l'ottimizzazione.

2.2 – Definizione di funzioni NON completamente specificate

Nel capitolo precedente abbiamo fatto vedere come è possibile specificare una funzione **completamente** specificata in SIS. È chiaro che a volte potrà capitare di imbattersi in funzioni **non completamente** specificate, cioè dotate di On-Set, Off-Set e DC-Set.

Grazie alla keyword

.exdc

Possiamo descrivere l'insieme dei don't care.

2.3 – Funzioni non completamente specificate – Esempio

```
. model esempio
. inputs a b c
. outputs z
. names a b c z     #definizione On – Set
001 1
100 1
111 1
. exdc     #definizione DC – Set
. names a b c z
010 1
101 1
110 1
. end
```

3 – Ottimizzazione di circuiti combinatori multilivello

3.1 – Problematiche con l'ottimizzazione multilivello

L'ottimizzazione a multilivello, a differenza di quella a due livelli, **non** consente di trovare l'ottimizzazione assoluta. Tramite alcune euristiche che saranno spiegate di seguito, otterremo un risultato buono ma non ottimo al 100%.

Le euristiche sono delle tecniche di calcolo per ottenere un risultato **vicino** all'ottimo. Statisticamente provato.

Consideriamo il circuito come un insieme di nodi interconnessi tra loro. Ad ogni nodo corrisponde una funzione booleana ad una sola uscita. Grazie a questa affermazione possiamo affermare che ogni .names crea un nodo.

3.2 – Euristiche

Le euristiche, come detto in precedenza, ci permettono di ottimizzare il circuito. Il risultato non è garantito perciò la situazione potrebbe migliorare, ma anche peggiorare!

Ogni euristica va a modificare ogni nodo del circuito, così facendo otterremo risultati migliori o peggiori.

- sweep: eliminazione dei nodi con un'unica linea di ingresso e di nodi con valore costante.

- eliminate: eliminazione di un nodo interno alla rete. Si consideri che il nodo *N* rappresenti la funzione $y = (a + b) * c$, l'eliminazione di *N* prevede la sostituzione della variabile y in tutti i nodi che la utilizzano con l'espressione booleana $(a + b) * c$.

- resub: sostituzione di un nodo interno con un insieme di nodi la cui funzionalità sia equivalente a quella del nodo sostituito. L'operazione viene effettuata per diminuire la complessità di un nodo.

- extract (fx): estrazione di una sotto espressione comune a più nodi che viene rappresentata con un nuovo nodo.

- simplify: riduzione della complessità di ogni singolo nodo con algoritmo di Quine-McCluskey.

3.3 – La magia dello “script.rugged”

Come si evinto dal paragrafo precedente, le euristiche sono molteplici. Esiste uno script in grado di combinare queste euristiche, utilizzando alcuni parametri, ed ottenere risultati vicini all’ottimo. Lo script in questione è il famoso script. rugged e si può invocare scrivendo sul terminale di sis:

source script. rugged

Ovviamente dopo aver caricato un file . blif.

Qui di seguito ci sono la serie di comandi che vengono lanciati.

sweep; eliminate −1

simplify −m nocomp

eliminate −1

sweep; eliminate 5

simplify −m nocomp

resub −a

fx

resub −a; sweep

eliminate −1; sweep

full_simplify −m nocomp

3.4 – Variabili interne

In SIS è possibile creare delle variabili interne. Queste saranno utili quando avremo più funzioni.

Nel caso in cui ci sia la funzione booleana $(i) = f(a, b, c, d, e)$ con le seguenti funzioni:

$f = ac + bd$

$i = f + ad$

Potremmo costruire un . blif scrivendo:

. model esempio

. inputs a b c d e

. outputs i

. names a b c d e f #funzione f

... #tabella di verità

. names a b c d e f i

... #tabella di verità

4 – Mapping tecnologico

4.1 – Mapping tecnologico (teoria)

La realizzazione pratica di un circuito digitale può avvenire in due modi:

1. costruzione di un chip mediante stampa del circuito su un piccolo quadrato di semiconduttore (ad esempio silicio);
2. utilizzo di circuiti integrati programmabili (come le FPGA), ovvero un sistema con un insieme di porte logiche elementari che possono essere interconnesse a piacimento per creare la funzione richiesta. È chiaro che è molto più economico e semplice utilizzare una FPGA.

In entrambi i casi, dovremo utilizzare determinate porte logiche. Ovviamente nella prima saranno solo quelle, mentre nella seconda dovremo selezionarle da una sorta di lista.

Per tale motivo, viene effettuato il ***mapping tecnologico*** su una determinata **libreria tecnologica** fornita dal costruttore di semiconduttori o FPGA. Le porte logiche presenti nella libreria fornita hanno sia area che ritardo *reali*. Dunque, sarà possibile conoscere queste due caratteristiche dopo aver "mappato" un circuito.

4.2 – Mapping tecnologico in SIS

Dopo aver scritto il file in formato .blif, e dopo aver effettuato determinate minimizzazioni, si possono eseguire i seguenti comandi per effettuare il mapping tecnologico.

In particolare, utilizzeremo

read_library *nomelibreria*

Per caricare la libreria tecnologica di nome "*nomelibreria*". Le librerie sono specificate nel formato genlib (estensione .genlib, ad esempio quella che utilizzeremo noi sarà synch.genlib).

Se vorremo visualizzare le informazioni inerenti alla libreria caricata, basterà scrivere

print_library

Per eseguire il **mapping** vero e proprio dovremo utilizzare il comando

map

Questo comando ha molti parametri che possono essere visti utilizzando l'help di SIS con il comando help map. Per esempio, l'opzione −m 0 permette di ottenere un circuito minimizzato rispetto all'area. L'opzione −n 1 permette di ottenere un circuito minimizzato rispetto al ritardo. Infine, l'opzione −s permette di visualizzare alcune informazioni relative ad area e ritardo dopo il mapping. Dalle statistiche dobbiamo prendere in considerazione solo la voce "total gate area" la quale fornirà il valore dell'area come numero di celle standard della libreria tecnologica e la voce "maximum arrival time" la quale indica il ritardo.

Possiamo vedere la rappresentazione del circuito associata alle porte della libreria con il comando

write_blif − n

Possiamo stampare le informazioni relative al ritardo del circuito

print_delay

Infine, possiamo ridurre la lunghezza dei cammini critici. Comando da eseguire ***prima*** di ogni ottimizzazione

reduce_depth

5 – Circuiti sequenziali e Macchine a Stati Finiti (Final State Machine)

5.1 - Sincronismo dei circuiti (teoria)

I circuiti possono essere di due tipi: (1) **sincroni** se i valori delle uscite assumono significato in corrispondenza di un evento su un segnale di sincronismo, chiamato clock, oppure (2) **asincroni** se i valori delle uscite cambiano al variare degli ingressi senza tener conto del segnale di sincronismo.

5.2 - Memoria

Poiché stiamo parlando di macchine sequenziali, la memoria è fondamentale. Essa ci permette di salvare i valori passati i quali influenzeranno la macchina nel presente. Questa memoria è rappresentata dallo stato. In elettronica abbiamo due componenti fondamentali in grado di memorizzare un bit: **latch** e **flip-flop**.

Una FSM con N stati necessita di $\log_2 N$ componenti di memoria elementari.

Per permettere al circuito di memorizzare gli stati della macchina, dobbiamo codificare quest'ultimi.

5.3 – Istruzioni passo – passo costruzione circuiti sequenziali in SIS

1.

Per iniziare a descrivere un circuito sequenziale, dobbiamo scrivere prima di tutto le keyword di segnalazione di una FSM. Così facendo SIS saprà che staremo descrivendo una FSM

.start_kiss

$5\ righe\ di\ informazioni$

$tabella\ delle\ transizioni$

.end_kiss

Successivamente dobbiamo scrivere 5 righe che specificano:

- numero di segnali di input
- numero di segnali di output
- numero di stati
- numero di transizioni
- stato di reset

Ed infine si potrà scrivere la tabella delle transizioni della FSM. Le transizioni dovranno essere specificate come un insieme di righe che riportano in ordine:

- valore degli ingressi;
- stato presente;
- stato prossimo;
- valore delle uscite.

Qui di seguito un esempio commentato.

```
.start_kiss
.i 7
.o 5
.s 9
.p 31
.r START

0------ START START 00000
1------ START INS1 00000
0------ INS1 START 00000
10101-- INS1 INS2 00000
11----- INS1 ERR_1 00000
1-0---- INS1 ERR_1 00000
1--1--- INS1 ERR_1 00000
```

L'inizio del codice con la delimitazione di .start_kiss, la descrizione delle informazioni sulla FSM, ovvero bit di input (7), bit di output (5), il numero di stati (9), il numero di transizioni (31), stato di reset (START). Dopo le due righe successive ci sono tutte transizioni della macchina.

2.

Dopo la tabella delle transizioni (dunque dopo .end_kiss) vengono riportate le istruzioni necessarie per codificare gli stati se si decidesse di fare una codifica **manuale**. La keyword per definire la codifica è ".code" seguita dal nome dello stato e dalla sua codifica binaria.

3.

Dopo aver modellato il circuito, è possibile minimizzare gli stati. A questo punto basterà caricare il file .blif ed eseguire la minimizzazione tramite il comando state_minimize_stamina.

4.

A questo punto è opportuno generare le funzioni δ e λ:

- Se è stata fatta la codifica **manuale**, il file conterrà già il .code, quindi occorrerà generare le funzioni semplicemente con il comando stg_to_network
- Se si è deciso di utilizzare la codifica automatica, a questo punto dovremo codificare e successivamente generare le funzioni. Per fortuna, SIS, in questo, ci viene incontro e ci permettere di fare entrambi i passaggi con un unico comando, ovvero state_assign jedi. **ATTENZIONE**, prima occorre minimizzare gli stati e poi farne l'assegnazione

5.

Infine, eseguire la minimizzazione delle funzioni tramite lo script.rugged o altri comandi.

6 – Progetto: Controllore-Datapath

6.1 - Modelli su più file

Per avere una corretta pulizia del codice, è buona norma creare in un primo momento un modello ad un bit e successivamente creare il modello di cui si ha bisogno con n bit. È chiaro che il modello di cui avremo bisogno, effettuerà n chiamate del file ad un singolo bit. Per esempio, un registro a 4 bit, chiamerà 4 volte (4 righe di codice), il componente "registro a 1 bit".

Per chiamare il file si utilizza la keyword

. subckt nome_componente ParamFormale = ParamAttuale ecc.

Ed alla fine del file si richiamerà solamente una volta il file di cui si ha bisogno usando la keyword

. search nomeFileDaRichiamare. blif

6.2 - Modelli su più file – Esempio (Sommatore)

Per capire meglio come implementare i le istruzioni in SIS, costruiamo, come esempio, un sommatore che somma due numeri ad 1 bit:

. model SOMMATORE

. inputs A B CIN

. outputs O COUT

. names A B K

10 1

01 1

. names K CIN O

10 1

01 1

. names A B CIN COUT

11 – 1

1 – 1 1

–11 1

. end

Se volessimo avere un sommatore che acquisisce due numeri a 2 bit ciascuno, basterà scrivere:

.model SOMMATORE2

.inputs A1 A0 B1 B0 CIN

.outputs O1 O0 COUT

.subckt SOMMATORE A = A0 B = B0 CIN = CIN O = O0 COUT = C0

.subckt SOMMATORE A = A1 B = B1 CIN = C0 O = O1 COUT = COUT

.search sommatore.blif

.end

Come si può vedere, viene inserita la keyword .subckt per indicare il collegamento dei bit.

C'è da notare un interessante collegamento tra le due righe di somma. Difatti, in questo esempio, avremo il primo numero formato dai bit A1 A0 e il secondo numero formato da B1 B0. Quindi al sommatore ad 1 bit passeremo le variabili A0 e B0 le quali andranno rispettivamente in A e B, quest'ultime variabili del "SOMMATORE" ad 1 bit.

In più utilizziamo CIN come Riporto in entrata (*Carry In*), O0 come output della prima somma e C0 come variabile interna. L'aspetto interessante è stato appena scritto, ovvero la variabile interna "C0" ha il compito di prendere un eventuale riporto in uscita dalla prima somma (COUT = *Carry Out*, quindi riporto in uscita) ed inserirlo come riporto in entrata nella seconda somma (CIN = C0, *Carry In*), lo troviamo nella seconda riga della somma.

Infine, c'è la chiamata al file di riferimento, in questo caso al file sommatore.blif.

6.3 – Registri in SIS

I registri in SIS sono un aspetto fondamentale, ci permetteranno di immagazzinare alcuni valori.

La dichiarazione di un registro a 1 bit è piuttosto articolata in SIS. La keyword per definire un registro è semplice, la cosa da ricordare bene sono i suoi parametri.

$$.\,\text{latch} \;<\text{input}><\text{output}><\text{type}><\text{control}><\text{init}-\text{val}>$$

- .latch: è la keyword per definire un registro.
- input: è l'ingresso del registro.
- output: è l'uscita del registro.
- type: è il tipo di registro. Esistono 5 tipi di registri: fe (*falling edge*), re (*rising edge*), ah (*active high*), al (*active low*), as (*asynchronous*). È utile sapere solo che utilizzeremo registri di tipo "re" ovvero che mutano sul fronte di salita del segnale.
- control: è il segnale di clock per il registro. Può essere di tre tipi: un clock del modello costruito, un'uscita di una qualsiasi funzione del modello oppure nessun clock interno. Anche qui basti sapere che utilizzeremo NIL per indicare l'assenza di un clock interno al registro. Ovviamente verrà utilizzato il clock generale del circuito in cui è inserito.
- init – val: è lo stato iniziale del registro, esso può essere $0, 1, 2, 3$. Zero e uno sono bit, due indica il don't care e tre indica "*unknown*" o non specificato.

6.4 – Registri in SIS – Esempio

Un esempio di un registro ad 1 bit può essere

```
.model REGISTRO
.inputs A
.outputs O
.latch A O re NIL 0
.end
```

Abbiamo scritto un registro che avrà come input A, come output O, come tipo di registro è un re (*rising edge*), come clock non utilizza quello interno ma quello del circuito (NIL) ed è posto a 0 come valore di default.

Come esempio lasciamo anche un modello di un registro a 4 bit.

```
.model REGISTRO4
.inputs A3 A2 A1 A0
.outputs O3 O2 O1 O0
.subckt REGISTRO A = A3 O = O3
.subckt REGISTRO A = A2 O = O2
.subckt REGISTRO A = A1 O = O1
.subckt REGISTRO A = A0 O = O0
.search registro.blif
.end
```

6.5 – Libreria di componenti in SIS

Nelle prossime pagine lasciamo una libreria di componenti scritti in SIS.

Multiplexer a 4 ingressi 1 bit ciascuno

```
.model MUX1
.inputs S1 S0 i3 i2 i1 i0
.outputs out
.names S1 S0 i3 i2 i1 i0 out
001---1
01-1--1
10--1-1
11---11
.end
```

Multiplexer a 4 ingressi 2 bit ciascuno

```
.model MUX2
.inputs X1 X0 a1 a0 b1 b0 c1 c0 d1 d0
.outputs o1 o0
.subckt MUX1 S1 = X1 S0 = X0 i3 = a1 i2 = b1 i1 = c1 i0 = d1 out = o1
.subckt MUX1 S1 = X1 S0 = X0 i3 = a0 i2 = b0 i1 = c0 i0 = d0 out = o0
.search mux1.blif
.end
```

Multiplexer a 2 ingressi da 3 bit ciascuno

```
.model MUX3
.inputs A2 A1 A0 B2 B1 B0 S
.outputs O2 O1 O0
.names S A2 B2 O2
11 – 1
0 – 1 1
.names S A1 B1 O1
11 – 1
0 – 1 1
.names S A0 B0 O0
11 – 1
0 – 1 1
.end
```

Demultiplexer a 1 bit e 4 uscite

```
.model DEMUX
.inputs S1 S0 IN
.outputs X Y Z W
.names S1 S0 IN X
001 1
.names S1 S0 IN Y
011 1
.names S1 S0 IN Z
101 1
.names S1 S0 IN W
111 1
.end
```

Comparatore a 4 bit

```
.model UGUALE4
.inputs A3 A2 A1 A0 B3 B2 B1 B0
.outputs O
.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 X3 X2 X1 X0 O
1111 1
.search xnor.blif
.end
```

“Maggiore” a 6 bit

Maggiore tra apici perché viene effettuato un confronto tramite lo xor e successivamente un confronto tra bit tramite una tabella di verità.

.model 6_gt

.inputs A5 A4 A3 A2 A1 A0 B5 B4 B3 B2 B1 B0

.outputs AgtB

.subckt xor A = A5 B = B5 X = X5

.subckt xor A = A4 B = B4 X = X4

.subckt xor A = A3 B = B3 X = X3

.subckt xor A = A2 B = B2 X = X2

.subckt xor A = A1 B = B1 X = X1

.subckt xor A = A0 B = B0 X = X0

.names A5 A4 A3 A2 A1 A0 X5 X4 X3 X2 X1 X0 AgtB

1−−−−−1−−−−− 1

−1−−−−01−−−− 1

−−1−−−001−−− 1

−−−1−−0001−− 1

−−−−1 − 00001 − 1

−−−−−1000001 1

.search xor.blif

“Minore-uguale” a 6 bit

. model 6_le

. inputs C5 C4 C3 C2 C1 C0 D5 D4 D3 D2 D1 D0

. outputs CleD

. subckt 6_gt A5 = C5 A4 = C4 A3 = C3 A2 = C2 A1 = C1 A0 = C0 …

B5 = D5 B4 = D4 B3 = D3 B2 = D2 B1 = D1 B0 = D0 AgtB = z

. names z CleD

0 1

. search 6_gt. blif

. end

Generazione di un valore costante a 0 (di un bit)

```
.model zero1
.outputs uscita
.names uscita
.end
```

Generazione di un valore costante a 1 (di un bit)

```
.model uno1
.outputs uscita
.names uscita
1
.end
```

Generazione del valore costante 10 (su 4 bit)

```
.model dieci4
.outputs O3 O2 O1 O0
.subckt uno1 uscita = O3
.subckt zero1 uscita = O2
.subckt uno1 uscita = O1
.subckt zero1 uscita = O0
.end
```

6.6 – Modellazione di una FSMD in SIS

Per costruire una FSMD è molto semplice. Prima di tutto bisognerà avere la FSM (controllore.blif) già codificata con i comandi che si potranno trovare al capitolo 5, paragrafo 3.

Successivamente si dovrà modellare il Datapath tramite una interconnessione di componenti funzionali, ovvero registri, sommatori, multiplexer, ecc.

Infine, dovremo costruire un file FSMD.blif nel quale sarà presente la giusta connessione tra FSM e Datapath. Si ricorda di specificare, con il comando .search, la FSM codificata!

Credits