

CRITERI PER IL CALCOLO DEL COSTO DELLE

PORTE LOGICHE:

- CRITERIO DELLA CARDINALITÀ: Conta ogni porta logica come equivalente 1, più porte logiche significa più costo
- CRITERIO DEI LETTERALI: Conta il numero di lettere nella rappresentazione algebrica
- CRITERIO DEL NUMERO DI INPUT: Conta gli input a TUTTE LE PORTE LOGICHE (non solo gli input alle prime)

Le prestazioni di una porta logica sono legate al ritardo della loro funzione logica

Il ritardo è il massimo dei ritardi di propagazione dovuti alla porta fra uscita e ingresso

"PERCORSO CRITICO" = Insieme delle porte logiche che producono il ritardo massimo

STRUTTURE DELLE PORTE LOGICHE:

- Somma di prodotti (SOP)
- Prodotto di somme (POS)

Alcune SOP e POS vengono definite forme canoniche

"MINTERMINI" = è una funzione booleana a n ingressi che vale 1 in corrispondenza della sola configurazione di ingresso che vale 1. Esso corrisponde al prodotto di n letterali, ciascuno dei quali compare nella forma \times se nella corrispondente configurazione d'ingresso ha valore 1, nella forma \bar{x} se ha valore 0. Data una funzione ad n ingressi, possiamo avere fino a 2^n mintermini ed ogni mintermine è rappresentabile con una AND ad n ingressi

DEFINIZIONE DI FORMA CANONICA: Per ogni funzione booleana f , esiste uno ed un solo insieme di mintermini che la rappresenta. La somma di questi mintermini costituisce la prima forma canonica di f .

"MAXTERMINE" = Un maxtermine Mi è una funzione booleana a n ingressi che vale 0 in corrispondenza della sola configurazione di ingresso che vale i. Essa corrisponde alla somma di n letterali, ciascuno dei quali compare nella forma \bar{x} se nella corrispondente configurazione di ingresso ha valore 1, nella forma x se ha valore 0. Data una funzione di n ingressi, possiamo ovvero fino a 2^n maxtermini ed ogni maxtermine è rappresentabile con un OR sui n ingressi.

DEFINIZIONE SECONDA FORMA CANONICA: L'espressione della funzione è il prodotto logico di tutti (esclusi) i termini somma delle variabili di ingresso corrispondenti agli 0 della funzione.

MINIMIZZAZIONE COL METODO DI KARNAUGH:

Il metodo di Karnaugh serve per minimizzare funzioni booleane espresse come S.O.P., Ingherito portando dalla prima forma canonica

Questo metodo si propone di identificare forme minime a due livelli applicando la regola di riduzione distributiva + neutra

esempio:

$$az + \bar{a}z = (a + \bar{a})z = z$$

con z termine prodotto di $n-1$ variabili

La rappresentazione grafica del metodo di Karnaugh è data dalle mappe di Karnaugh.

La sua applicazione è semplice per un numero di variabili fino a 4, mentre risulta complessa per un numero di variabili da 5 a 6. È praticamente inutilizzabile per un numero di variabili superiori a 6.

Una mappa di Karnaugh per una funzione f di n variabili è una griglia (mappa) con 2^n celle. Le celle rappresentano i valori della funzione stessa mentre le righe e le colonne rappresentano i valori possibili degli ingressi

esempio:

$$f(a,b) = \text{ON}_{\text{SET}}(1, z)$$

	0	a_1
0	0	1
1	1	0

quando il numero di variabili è maggiore di due bisogna raggruppare le variabili in due insiemi e considerare le possibili combinazioni ordinate in maniera particolare

esempio:

ONSET (1,3,4,5,6)

		b	c		
		00	01	11	10
d	0	0	1	1	0
	1	1	1	0	1

OSSERVAZIONE: L'ordinamento nelle stesse righe e colonne mantiene la distanza di Hamming minima delle stesse (in sostanza: puoi notare solo di 1 bit fra righe e colonne adiacenti tra loro)

DEFINIZIONE: Un implicante è un termine prodotto di cui compiono solo alcune variabili (letterali) della funzione stessa. L'unione degli implicanti validi di una funzione è un insieme di mintermini.

esempio:

$$f(a,b,c,d) = \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}\bar{c}d \dots$$

convertendo in mappa di Karnaugh...

		00	01	11	10
		1			
a	0	1			
	1				
b	0				
b	1				
c	0				
c	1				
d	0				
d	1				

Dato un raggruppamento è possibile ricavare l'implicante corrispondente identificando le variabili

MINIMIZZAZIONE CON METODO DI QUINE - MCCLUSKY:

Il metodo di Quine-McClusky è un metodo di minimizzazione tabellare che è facile da tradurre in un algoritmo. Di fatto, il numero di variabili trattate è teoricamente illimitato ma, siccome il problema della identificazione sia degli implicanti primi sia della copertura ottima della funzione è di complessità esponenziale, è praticamente impossibile identificare una soluzione ottima per un numero di variabili che supera l'ordine della decina. Dato questo, il metodo è facile da estendere al caso di funzioni a più di un'uscita e consente di utilizzare diverse funzioni di costo purché additiva.

La struttura del metodo è molto simile alle mappe di Karnaugh:

1: Ricerca e identificazione di tutti gli implicanti primi (o esponente)

2 : Ricerca e identificazione della copertura ottima

FASE 1: Ricerca degli implicati primi
esempio:

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Sappiamo che gli implicati primi derivano dai mintermini della funzione (segnati con delle frecce)

LINGUAGGIO ASSEMBLY (RISC V):

REGISTRI:

application binary interface

registro:	minimo A B I	descrizione
x0	0000 0000 0000 0000	hardwired zero
x1	0000 0000 0000 0001	return address
x2	0000 0000 0000 0010	stack pointer
x3	0000 0000 0000 0011	global pointer
x4	0000 0000 0000 0100	thread pointer
x5 - 7	0000 0000 0000 1000 - 0000 0000 0000 0111	temporary registers
x8	0000 0000 0000 1000	saved register/frame pointer
x9	0000 0000 0000 1001	saved register
x10 - 11	0000 0000 0001 0000 - 0000 0000 0001 0000	function arguments/return values
x12 - 17	0000 0000 0010 0000 - 0000 0000 0011 0000	function arguments
x18 - 27	0000 0000 0100 0000 - 0000 0000 1000 0000	saved registers
x28 - 31	0000 0000 1000 0000 - 0000 0000 1111 0000	temporary registers

ISTRUZIONI ARITMETICO - LOGICHE:

ISTRUZIONE:

add x2, x3, x5
sub x3, x4, x6
addi x2, x3, 10

SIGNIFICATO:

x2 \leftarrow x3 + x5
x3 \leftarrow x4 - x6
x2 \leftarrow x3 + 10

PSEUDO - ISTRUZIONI:

ISTRUZIONE:

mop
mr r1, rs
neg r1, rs

SIGNIFICATO:

addi x0, x0, 0
addi r1, rs, 0
sub r1, r2, zero, rs

ISTRUZIONI LOGICHE BIT A BIT:

istruzione:

and x_2, x_3, x_5

or x_3, x_4, x_6

andi $x_2, x_3, 1$

significato

$x_2 \leftarrow x_3 \wedge x_5$

$x_3 \leftarrow x_4 \vee x_6$

$x_2 \leftarrow x_3 \wedge 1$

ISTRUZIONI DI SCORRIMENTO LOGICO:

istruzione:

slli $x_2, x_3, 10$

srl $x_2, x_3, 10$

significato

$x_2 \leftarrow x_3 \ll 10$

$x_2 \leftarrow x_3 \gg 10$

con gli shift logici possiamo moltiplicare e dividere numeri positivi (proprietà della codifica binaria)

esempio:

x_3 vale 6

$x_3 = 00000 \dots 000110$

Se vogliamo moltiplicare per 2 il valore di x_3 , basta shiftare a sinistra di una posizione il suo contenuto:

slli $x_3, x_3, 1$

Che fa posto ad uno zero in posizione meno significativa, ovviamente scrivendo lo 0 in posizione più significativa:

$x_3 = (0)00000 \dots 001100 \rightarrow 12$ in binario

aggiunto

↓

perde

Nel caso a una wordline si debba aggiungere una costante il cui valore è più ampio di 12 bit in complemento a due, bisogna usare le istruzioni lui e sottrarre la costante in due; ad esempio:

$$a = b + (1 \ll 15) + (1 \ll 3)$$

avrà trascritta in assembly $a5 = a, a4 = b$)

addi $t_0, t_0, 1$

addi $t_1, t_1, 1$

slli $t_0, t_0, 15$

slli $t_1, t_1, 3$

add t_0, t_0, t_1

add $a5, a4, t_0$

ISTRUZIONI DI TRASFERIMENTO DATI (LOAD e STORE):

La specifica riserva varie dimensioni di dati che si riflettono sulla specifica operazionale da usare per caricare o salvare quel dato in memoria:

dimensione	nome convenzionale	SUFFISSO:
8-bit	byte	b
16-bit	half word (metà parola)	h
32-bit	word (parola)	w
64-bit	double (o long) word (parola doppia)	d

Le istruzioni di trasferimento dati sono le **load** (caricamento da memoria) e le **store** (svilaggiamento in memoria). Gli operandi di ciascuna sono 3:

- un registro sorgente o destinazione
- un registro per la base
- una costante di offset (massimo di 12 bit in complemento a 2)

La base e la costante di offset specificano l'indirizzo sorgente (o load) o destinazione (store) in modalità di indirizzamento **base+offset**. Ecco alcuni esempi di load (in generale useremo M[a_0] per indicare, nel descrivere il significato dell'istruzione, la cella all'indirizzo a nella memoria centrale del calcolatore):

lh $t_0, 0(a_0)$
lw $t_0, 0(a_0)$
ld $t_0, 8(a_0)$

significato:
 $t_0 \leftarrow M[a_0 + 0] \rightarrow 16 \text{ bit}$
 $t_0 \leftarrow M[a_0 + 0] \rightarrow 32 \text{ bit}$
 $t_0 \leftarrow M[a_0 + 8] \rightarrow 64 \text{ bit}$

Per le store invece avremo:

sw $s_1, 100(a_0)$

significato:
 $M[a_0 + 100] \leftarrow s_1 \rightarrow 32 \text{ bit}$

esempio: manipolazione degli array tramite load e store:

in C:

```
int32_t a[20], b;  
...
```

```
a[12] = b + a[8];
```

in assembly:

```
lw t0, 32($3)  
addi t0, $2, t0  
sw t0, 48($3)
```

(ricorda che ogni cella di memoria per una word è di 32 bit, cioè 4 byte, per questo per ogni cella di memoria aggiungiamo un offset di 4)

Dunque l'indirizzazzione generica è supportata efficacemente; supponiamo di avere invece l'istruzione in linguaggio C:

$$h = h + a[i];$$

dove i è contenuto nel registro s_1 . Bisogna considerare che ciascun elemento è di quattro bytes quindi, per accedere ad $a[i]$ bisogna moltiplicare per 4 (o shiftare a sinistra di 2) il valore di i :

$sh\ t_1, s_1, 2$	$i \cdot 4$
$add\ t_1, s_3, t_1$	$a + i \cdot 4$
$lw\ t_0, 0(t_1)$	$M[a + i \cdot 4]$
$add\ s_2, s_2, t_0$	$h = h + M[a + i \cdot 4]$

ISTRUZIONI DI SALTO:

Tali istruzioni alterano l'ordine di esecuzione delle istruzioni. La prossima istruzione da eseguire non è necessariamente l'istruzione successiva all'istruzione corrente, ma quella individuata dalla destinazione del salto. Queste istruzioni permettono di realizzare i costrutti di controllo condizionali e ciclici.

In linguaggio assembly si specifica l'indirizzo dell'istruzione