Tesi

Andrea Locatelli

Indice

Capitolo 1

Secure multi-party computation

Secure multi-party computation (SMPC) è una sotto categoria della crittografia con l'obiettivo di creare metodi di calcolo congiunto di una funzione sugli input privati di 2 o più parti coinvolte.

A differenza dei compiti crittografici tradizionali, dove la crittografia assicura la sicurezza e l'integrità della comunicazione o dell'archiviazione e l'avversario è esterno al sistema dei partecipanti, la crittografia in questo modello protegge la privacy dei partecipanti l'uno dall'altro.

Siamo in un periodo in cui i dati sono sempre più centrali e oggetto di interesse per le aziende diventando un nuovo metodo di guadagno. La monetizzazione dei dati è oggi il modello di business primario comune a molte delle più grandi aziende, e la priorità di tante altre.

L'applicazione di un simile modello di business richiede tuttavia la raccolta di grandi quantità di dati, e il successo è tanto più significativo quante più correlazioni e rapporti di casualità sia possibile trovare combinando diverse fonti dati, si devono però considerare le problematiche.

Il sottoprodotto di questo processo è infatti una possibile violazione della privacy individuale, avendo le aziende molte nostre informazioni personali il loro possesso.

Si sono pensate allora soluzioni in cui sia possibile utilizzare fonti dati diverse senza la necessità di centralizzare l'informazione, garantendo che nessuna informazione sia rivelata dall'operazione. Queste sono le premesse della Secure multiparty computation (SMPC). In altri termini, saremmo così in grado di consentire ai Data Scientist e Data Analyst di operare sui dati senza il bisogno di esporli o spostarli dalla loro sede di storage.

1.1 Panoramica

In una SMPC un dato numero di partecipanti $p_1, p_2, ..., p_N$, ognuno dei quali ha dei dati privati $d_1, d_2, ..., d_N$. I partecipanti vogliono calcolare il valore di una funzione pubblica con i loro dati privati: $F(d_1, d_2, ..., d_N)$ mantenendo i loro input segreti.

Per esempio, supponiamo di avere tre parti Alice, Bob e Charlie, con i rispettivi input x, y e z che denotano i loro stipendi. Vogliono scoprire il più alto dei tre stipendi, senza rivelare all'altro quanto guadagna ognuno di loro. Matematicamente, questo si traduce in un calcolo: $F(x, y, z) = max(x, y, z)$

Se ci fosse una parte esterna fidata (diciamo che hanno un amico comune, Tony, che deve mantenere i segreti delle parti), ognuno potrebbe dire il proprio stipendio a Tony, lui potrebbe calcolare il massimo e dire quel numero a tutti loro.

L'obiettivo di SMPC è di progettare un protocollo in cui, scambiando messaggi solo tra di loro, Alice, Bob e Charlie possono ancora calcolare $F(x, y, z)$ senza rivelare chi fa cosa e senza dover dipendere da terzi. Non dovrebbero saperne di più, impegnandosi nel loro protocollo, di quanto saprebbero interagendo con un Tony incorruttibile e perfettamente degno di fiducia.

In particolare, tutto ciò che le parti possono sapere è ciò che loro possono apprendere dall'output e dal loro stesso input. Così nell'esempio precedente, se l'output è z, allora Charlie impara che il suo z è il valore massimo, mentre Alice e Bob imparano (se x, y e z sono distinti), che il loro input non è uguale al massimo, e che il massimo è uguale a z. Lo scenario di base può essere facilmente generalizzato ai casi in cui le parti hanno diversi input e output, e la funzione fornisce valori diversi alle diverse parti.

Un protocollo di calcolo sicuro multipartitico deve offrire alcune garanzie di sicurezza, persino se alcune delle parti fossero in collusione o cercassero di violarne le regole:

- **Input Privacy**: Nessuna delle parti corrotte (o suo sottoinsieme) deve essere in grado di derivare alcuna informazione sui dati appartenenti alle altre parti, a eccezione di quanto rivelato dal risultato dell'operazione
- **Correctness**: Nessuna delle parti corrotte (o suo sottoinsieme) deve essere in grado di indurre una parte onesta a produrre un risultato errato.

1.2 Garanzie di sicurezza

Grazie alle SMPC le parti coinvolte nelle computazioni non hanno bisogno di fidarsi reciprocamente, anche qualora alcune di queste intendano collaborare ai danni delle altre o quando reti e macchine siano state compromesse a loro insaputa in quanto nessuna informazione viene rivelata alle altre parti. Abbiamo quindi i seguenti aspetti positivi:

- Assenza di una necessaria terza parte fidata per mantenere i dati sicuri.
- Nessun compromesso di Privacy-vs-Utility, poiché non dobbiamo perturbare alcun dato per tutelarne la privacy.
- GDPR compliance persino per interazioni tra aziende europee, americane e asiatiche, perché i dati non lasciano mai i confini.
- Protezione di un segreto, che può essere utilizzato per svolgere operazioni senza mai crearlo effettivamente

Ovviamente non mancano gli aspetti negativi:

- Costo computazionale elevato, richiesto per la generazione dei numeri casuali che garantiscono la sicurezza nei calcoli.
- Costo comunicativo elevato, richiesto dal network per la condivisone con i partecipanti.
- Network bound

1.3 Two-party computation

Una particolare sotto-categoria delle SMPC è la Two-party computation. Questa categoria descrive uno scenario in cui 2 parti parti comunicano tra di loro per la risoluzione di un problema senza scambiarsi informazioni sensibili e senza l'utilizzo di una terza parte fidata.

Lo scenario a due parti è particolarmente interessante, non solo dal punto di vista delle applicazioni, ma anche perché si possono applicare tecniche speciali nel contesto a due parti che non si applicano nel caso a più parti. Infatti, il calcolo sicuro a più parti è stato presentato per la prima volta nell'impostazione a due parti. Il lavoro originale è spesso citato come proveniente da uno dei due articoli di Yao; anche se gli articoli non contengono effettivamente quello che ora è noto come il Yao's garbled circuit protocol.

Il protocollo di base di Yao è sicuro contro gli avversari semi-onesti ed è estremamente efficiente in termini di numero di passaggi, che è costante, e indipendente dalla funzione obiettivo che viene valutata. La funzione è vista come un circuito booleano, con ingressi in binario di lunghezza fissa. Un circuito booleano è un insieme di porte collegate con tre diversi tipi di fili: fili di ingresso al circuito, fili di uscita al circuito e fili intermedi. Ogni porta riceve due fili d'ingresso e ha un singolo filo d'uscita che potrebbe essere fan-out (cioè essere passato a più porte al livello successivo). La semplice valutazione del circuito viene fatta valutando ogni porta a turno; assumendo che le porte siano state ordinate topologicamente. Il gate è rappresentato come una tabella di verità tale che per ogni possibile coppia di bit (quelli provenienti dal gate dei fili di ingresso) la tabella assegna un unico bit di uscita; che è il valore del filo di uscita del gate. I risultati della valutazione sono i bit ottenuti nei fili di uscita del circuito.

Yao ha spiegato come confondere un circuito (nascondere la sua struttura) in modo che due parti, mittente e ricevitore, possano imparare l'uscita del circuito e nient'altro. Ad un alto livello, il mittente prepara il circuito confuso e lo invia al ricevitore, che ignaro valuta il circuito, imparando le codifiche corrispondenti all'uscita sua e del mittente. Poi si limita a rimandare le codifiche del mittente, permettendo a quest'ultimo di calcolare la sua parte di output. Il mittente invia la mappatura dalle codifiche di uscita dei ricevitori ai bit al ricevitore, permettendo a quest'ultimo di ottenere la propria uscita. Nel capitolo successivo andremo più in profondità spiegando dettagliatamente la logica di funzionamento.

Capitolo 2

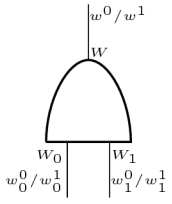
Yao Garbled Circuit

Si supponga che Alice e Bob siano disposti a calcolare in modo sicuro una funzione $f(x, y)$ mantenendo segreti i rispettivi input x e y.

Per fare ciò, essi modellano prima la funzione f come un circuito booleano, questo è possibile poiché esiste un circuito booleano C che calcola l'uscita di f per qualsiasi funzione f con ingressi di dimensione fissa. Tuttavia, il modo in cui tale modellazione viene eseguita può dipendere dalla funzione e non sarà ulteriormente discusso qui. Successivamente Alice confonderà il circuito booleano e:

1. Per ogni filo w_i del circuito C, sceglie casualmente due valori segreti w_i^0, w_i^1, dove w_i^j è il valore confuso del valore $j \in \{0, 1\}$ del filo w_i. Si noti che w_i^j non può rivelare j di per sé, quindi Alice deve tenere traccia di i e j. Questo deve essere fatto per ogni singolo filo di ingresso e di uscita di ogni porta logica del circuito, tranne che per le porte di uscita del circuito che possono essere lasciate in chiaro.
2. Alice dovrà costruire una tabella di verità confusa (GTT) T_i per ciascuna delle porte logiche G_i in C.

Queste tabelle devono essere tali che dati valori confusi lungo il suo insieme di fili d'ingresso, T_i permetterà di recuperare l'uscita confusa di questo G_i e nessun'altra informazione. Questo si ottiene attraverso la crittografia dei valori di uscita. Di seguito dettaglierò ulteriormente il garbling delle porte.



W_0	W_1	W
0	0	0
0	1	0
1	0	0
1	1	1

Figura 2.1: AND gate con etichette e tabella di verità.

In seguito, Alice può tradurre ogni bit del suo input nei suoi corrispondenti valori confusi sui fili di ingresso del circuito. Successivamente può inviare il circuito, ora confuso, dove ogni porta è sostituita dal suo GTT a Bob con il suo input criptato.

Dopo che Bob ha ricevuto il circuito confuso, poiché tutti i fili d'ingresso sono criptati e solo Alice conosce la mappatura dei valori criptati e i bit reali, Bob ha bisogno di eseguire un Oblivious transfer con Alice per ciascuno dei suoi bit d'ingresso, in modo che Alice possa informarlo di quali valori criptati corrispondono ai suoi bit d'ingresso e sapere quali sono i suoi bit d'ingresso reali.

> **Oblivius transfer**: tipo di protocollo in cui il mittente trasmette un pezzo di informazione a un ricevitore, tra tante potenziali, ma rimane ignaro al mittente quale pezzo di informazione sia stato trasmesso.

Quindi significa che per ogni filo di ingresso, Bob sceglierà una tra le due stringhe casuali w_i^0, w_i^1 che corrispondono rispettivamente a 0 e 1, ma senza conoscere il contenuto della stringa che non sceglie. E grazie alle proprietà del Oblivious transfer Alice non può conoscere l'input di Bob.

Allora Bob ha tutti i valori necessari per calcolare l'uscita del circuito, come discuterò in seguito. Una volta fatto ciò, può comunicare i valori di uscita ad Alice. Così Bob è stato in grado di ottenere l'uscita di f senza rivelare il suo input, né conoscere l'input di Alice, questo significa che Alice e Bob hanno simulato con successo una terza parte fidata e hanno eseguito un SMPC sicuro.

2.1 Garbling Logical Gates

La nozione di garbling delle porte logiche e della loro tabella di verità è cruciale. Senza perdita di generalità, considererò solo le porte logiche con due fili di ingresso e un filo di uscita. Come ho spiegato sopra, per una data porta $G \in C$ e i suoi fili d'ingresso W_0, W_1 e il suo filo di uscita W, Alice doveva scegliere sei diverse stringhe casuali, $w_0^0, w_0^1, w_1^0, w_1^1, w^0, w^1$ che ha assegnato a ciascun valore dei fili in una mappatura uno a uno, dove w_i^j rappresenta la stringa casuale assegnata al valore j del filo W_i.

Quindi, per confondere la tabella di verità di G in modo da non rivelare alcuna informazione dati due valori di ingresso w_0, w_1 eccetto il suo valore di uscita w, nemmeno il tipo di porta logica, Alice può criptare i valori di uscita w_0, w_1 usando i valori di ingresso confusi come chiavi, usando un dato schema di crittografia simmetrica $\mathbb{E}$. Uso la notazione $\mathbb{E}_{k_0,k_2}(x) = \mathbb{E}_{k_0}(\mathbb{E}_{k_1}(x))$ per indicare la cifratura doppia con due chiavi date k_0, k_1. Come esempio, criptiamo la tabella di verità della porta AND della figura:

W_0	W_1	W	Garbled value
w_0^0	w_1^0	w^0	$\mathbb{E}_{w_0^0,w_1^0}(w^0)$
w_0^0	w_1^1	w^0	$\mathbb{E}_{w_0^0,w_1^1}(w^0)$
w_0^1	w_1^0	w^0	$\mathbb{E}_{w_0^1,w_1^0}(w^0)$
w_0^1	w_1^1	w^1	$\mathbb{E}_{w_0^1,w_1^1}(w^1)$

La GTT T del gate G è semplicemente l'insieme $\left\{\mathbb{E}_{w_0^j,w_1^k}(w^{G(j,k)}) \mid j,k \in \{0,1\}\right\}$ dei valori confusi, dove $G(j,k)$ corrisponde all'uscita della porta G sotto ingressi (j,k).

2.2 Valutazione del Garbled Circuit

Una volta che Bob ha ricevuto il circuito confuso C da Alice e ha ottenuto i valori confusi del suo input attraverso diversi Oblivious transfer, può valutare il circuito.

È importante capire che un circuito confuso differisce da un normale circuito booleano. In un circuito booleano, semantica e sintassi sono fondamentalmente le stesse: stiamo assegnando ad ogni filo due possibili valori semantici, cioè Vero o Falso, che sintatticamente denoteremo come un segnale con valori 1 o 0 rispettivamente.

Questi segnali erano pubblici e gli stessi segnali erano associati ad ogni filo e chiunque poteva dire dal segnale quale valore semantico aveva. Questo cambia in un circuito confuso, poiché i valori semantici di ogni segnale, eccetto quelli di uscita del circuito, sono ora segreti e i segnali variano da un filo all'altro.

Così, per valutare il circuito, per ogni porta G_i del circuito, Bob può provare a decifrare i valori nella tabella di verità associata T_i usando i valori di ingresso della porta come chiavi. Una delle voci in T_i sarà poi decifrata nell'uscita del gate. Quindi sembra necessario avere un oracolo che confermi il successo della decrittazione delle voci di T_i, tuttavia un trucco che descriverò in dettaglio ora, chiamato permute-and-point usato per la prima volta in (Beaver, Micali, e Rogaway 1990) e poi spiegato chiaramente nella tesi di Phillip Rogoway (Rogaway 1991), permette di decidere quale voce della GTT deve essere decrittata dati gli input confusi, permettendo calcoli più veloci e impedendo comunque al valutatore di dedurre qualcosa dall'ordine delle voci della tabella di verità.

2.3 Permute-and-point

Questo meccanismo funziona nel seguente modo: per ogni filo di ingresso e uscita w_i, Alice concatena un bit casuale $a \in \{0, 1\}$ alla fine del suo valore confuso w_i^0 e concatena il suo valore inverso $b = \overline{a} = 1 - a$ alla fine di w_i^1. Quindi permette di associare ciascuna delle 4 permutazioni di 2 bit a una delle entrate della GTT, senza avere alcuna correlazione tra i bit e i valori della tabella della verità non confusa. Così Alice può semplicemente ordinare la GTT secondo l'ordinamento naturale e darla a Bob che sarà quindi in grado di dedurre quale voce deve decifrare su un dato input. Per ottenere una bella rappresentazione di questo trucco, quei bit possono essere visti come una coppia di colori, come li raffiguro nella figura 2.2, in cui vediamo come la tabella di verità viene modificata per tenere conto di questo metodo.

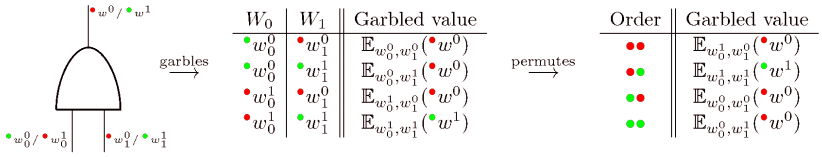


Figura 2.2: AND garbled con permute on point

Queste modifiche permettono a Bob di decifrare semplicemente la voce il cui indice corrisponde ai colori associati ai suoi fili di ingresso e quindi di ottenere il valore del filo di uscita e il suo colore, permettendogli di valutare ulteriormente il circuito.

2.4 Esempio di Valutazione

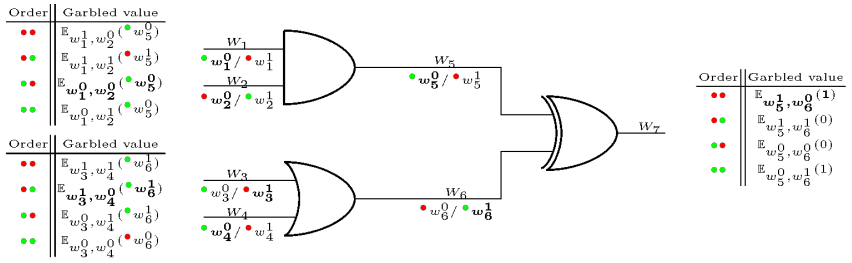


Figura 2.3: AND garbled con permute on point

Vediamo come si potrebbe valutare il circuito confuso rappresentato nella Figura 2.3 usando il metodo permute-and-point che abbiamo discusso sopra. Assumiamo che i valori semantici di ingresso di (W_1, W_2, W_3, W_4) siano $(0, 0, 1, 0)$, il che significa che l'input confuso effettivo è $(\bullet w_1^0, \bullet w_2^0, \bullet w_3^1, \bullet w_4^0)$ dove gli w_j^i sono i valori casuali che Alice ha scelto quando ha confuso il circuito, come visto sopra. Assumiamo anche che Alice abbia già fornito il suo input confuso, diciamo (W_1, W_3), e che Bob abbia già ottenuto il suo input confuso (W_2, W_4) da Alice attraverso due applicazioni di Oblivious transfer come descritto nella parte sul protocollo.

Quindi Bob comincerà a valutare prima la porta AND utilizzando l'ingresso $(\bullet w_1^0, \bullet w_2^0)$, dato che ha i colori $\bullet\bullet$ cercherà di decifrare la terza voce della GGT della porta AND, che funziona e quindi gli fornirà il valore confuso $\bullet w_5^0$.

Poi può continuare la sua valutazione con la seconda porta, che è la porta OR. Guarda il suo ingresso $(\bullet w_3^1, \bullet w_4^0)$ e prova a decifrare la voce corrispondente a $\bullet\bullet$ con le chiavi (w_3^1, w_4^0), essa si decifra nel valore confuso $\bullet w_6^1$.

Ora può decifrare la porta XOR finale usando l'ingresso calcolato ($\bullet w_5^0, \bullet w_6^1$), decifrando così la voce $\bullet\bullet$ che gli fornisce il risultato finale: 1.

Bob non sa quale sia stato l'input di Alice, conosce solo l'output finale "1" e le stringhe generate casualmente $w_1^0, w_2^0, w_3^1, w_4^0, w_5^0, w_6^1$. Egli può ancora, per esempio, dedurre dal circuito che i valori semantici di w_5^0 e w_6^1 sono opposti, tuttavia non gli permette di invertire il circuito confuso fino ai valori di ingresso di Alice.

Ci sono circuiti in cui non è assicurata la privacy, come per esempio un circuito che calcola la somma degli ingressi, tuttavia in questo caso di esempio, Bob - conoscendo i suoi valori semantici di ingresso - può semplicemente limitare gli ingressi di Alice a un sottoinsieme dei possibili ingressi, ma non può determinare in modo univoco i reali valori di ingresso di Alice.

Capitolo 3

Analisi

Il protocollo di Yao prevede, durante i suoi scambi, che le due parti concordino anche sulla trasmissione di un circuito a cui entrambi gli attori debbano inserire dei valori di input. Uno degli scopi della nostra tesi consiste nel testare l'efficacia dell'utilizzo di un multi valore rispetto all'utilizzo "classico" di un circuito booleano.

Per poter iniziare questa fase di analisi siamo partiti da dei circuiti binari per poi convertirli nella loro alternativa multivalore implementando un metodo di conversione efficace che non cambi il funzionamento di tali circuiti. Nel seguente capitolo descriverò come abbiamo concepito e applicato questa conversione.

3.1 Studio del circuito binario

La tecnica che abbiamo deciso di utilizzare è molto semplice ma efficace e si basa sul raggruppamento di n bit di ogni riga di ogni tabella di verità binaria che possano rappresentare il massimo numero possibile del dominio multivalore scelto.

$$\begin{array}{ccc} \overbrace{10} & \overbrace{11} & \overbrace{00} \\ \downarrow & \downarrow & \downarrow \\ 2 & 3 & 0 \end{array} \quad mod_3$$

Con questo approccio non andiamo a snaturare quella che è la logica della tabella originale e otteniamo una conversione efficace.

I vincoli dettati da questo stanno nel fatto che il numero di ingressi e uscite del circuito‘ devono essere pari.

Per la scelta del dominio multivalore si sono voluti sperimentare 2 diversi approcci:

- Per tutti i circuiti utilizzare un dominio multivalore standard di valore 3
- Per ogni circuito viene calcolato M.C.D tra input e output e:
  - Se il valore di M.C.D consente di avere almeno 2 input viene tenuto quello
  - Se non consente al minimo 2 input viene dimezzato

Per potere effettuare queste operazioni si è dovuto cercare un formato standard di descrizione dei circuiti che rappresentasse per ogni riga della tabella delle verità del circuito tutti gli input e tutti gli output.

Tra i vari formati a disposizione la scelta è ricaduta sul formato `PLA` già ampiamente utilizzato per la rappresentazione di circuiti.

3.1.1 Analisi della struttura PLA

Un file PLA ha la seguente struttura

```
.i 4
.o 2
.ilb x1 x2 y1 y2
.ob f1 f2
0--0 00
0001 01
0-11 --
1-11 01
0101 10
10-- 01
11-- 00
.end
```

Come possiamo vedere questo circuito presenta:

- 4 input
- 2 output

Notiamo che nella tabella di verità non sono presenti solamente valori booleani ma anche un simbolo - esso si prende il nome di **don't care** e rappresenta:

- Negli output implica che quella specifica combinazione non mi interessa
- Negli input che può essere qualsiasi valore del dominio.

3.1.2 La gestione dei don't care durante la conversione

Nelle successive fasi di conversione dobbiamo gestire i don't care in maniera funzionale al dominio multi valore che utilizzeremo, per spiegare meglio come tratteremo questa caratteristica dei PLA utilizzerò un esempio utilizzando un circuito semplice e come valore di dominio di conversione il valore 3. Il circuito avrà la seguente struttura.

```
.i 4
.o 2
.ilb x1 x2 y1 y2
.ob f1 f2
0--0 00
0001 01
0-11 --
1-11 01
0101 10
10-- 01
11-- 00
.end
```

Per passare al nuovo dominio analizziamo quanti bit sono necessari per poter rappresentare il massimo numero possibile, nel nostro caso il 3, e notiamo che abbiamo bisogno di 2 bit poiché la rappresentazione binaria del numero 3 è 11. Andiamo quindi a raggruppare in blocchi di 2 bit gli ingressi e le uscite del circuito andando a convertire i valori binari dati dal raggruppamento in valore del dominio scelto.

Procedendo con la conversione ci accorgiamo che ci sono alcuni casi in cui dobbiamo andare a sviluppare i dont't care perché se non lo facessimo perderemmo dei valori significativi per il nostro sviluppo. Andiamo quindi a distinguere 2 casi:

- Quando il numero di '-' è uguale al numero di bit del raggruppamento multivalore
  - *es.*. 10 – 01: in questo caso viene messo semplicemente il simbolo - non causa nessuna perdita di valori nel circuito. Quindi

$$\begin{array}{ccc} 10 & -- & 01 \\ & \downarrow & \\ 2 & - & 1 \end{array}$$

- Quando nel gruppo di bit raggruppati il numero di - è minore del numero di bit richiesti
  - *es.* 1-11 01: in questo caso non posso subito convertire nel nostro nuovo dominio ma devo prima sviluppare tutte le possibili combinazioni e poi procedere alla conversione. Quindi
    - Sviluppo il dont'care

$$\begin{array}{ccc} 10 & 11 & 01 \\ 11 & 11 & 01 \end{array}$$

    - Procedo a convertire nel dominio scelto

$$\begin{array}{ccc} 2 & 3 & 1 \\ 3 & 3 & 1 \end{array}$$

Così facendo riusciamo a sviluppare tutto il circuito binario e ottenere una conversione esatta. Il circuito convertito sarà quindi.

```
00 0
02 0
10 0
12 0
01 1
03 -
13 -
23 1
11 2
2- 1
3- 0
```

Con questa struttura posso proseguire con l'analisi dei circuiti proseguendo con l'attività di sintesi logica.

3.2 Implementazione della conversione

La conversione dei circuiti in analisi viene implementata utilizzando il linguaggio di programmazione Python nella versione 3.8 senza l'ausilio di nessun package esterno, vengono utilizzate solamente librerie comprese nel linguaggio, questo fa si che il sistema sia "ready to use" una volta installato il linguaggio di programmazione se non già presente all'interno del SO.

3.2.1 Analisi del circuito

```
def read_pla(path_file):
    inp = None
    out = None
    inp_array = []
    out_array = []
    truth_table = []
    with open(path_file, 'r') as input_file:
        for line in input_file.readlines():
            if '.i' in line and line[2] == ' ':
                inp = line.split(' ')[1]
            elif '.o' in line and line[2] == ' ':
                out = line.split(' ')[1]
            elif '.ilb' in line:
                inp_array = line.strip().split(' ')[1:]
```

```
            elif '.ob' in line:
                out_array = line.strip().split(' ')[1:]
            elif '.end' in line:
                continue
            else:
                line = {
                    'inp':  line.strip().split(' ')[0],
                    'out':  line.strip().split(' ')[1]
                }
                truth_table.append(line)
    return inp, out, inp_array, out_array, truth_table
```

La funzione prende in ingresso il percorso di un circuito e memorizza all'interno di un dizionario informazioni tipo:

- Tabella delle verità
- Numero di input
- Numero di output

Queste informazioni serviranno successivamente per convertire il circuito e calcolare costi.

3.2.2 Espansione dei don't care

Una volta ottenuto tutte le informazioni disponibili dal circuito dato bisogna andare ad identificare all'interno delle tabelle di verità quali sono i *don't care* a cui bisogna espandere i valori e quelli che si possono ignorare. Questa parte è stata la parte più impegnativa di questa parte di funzionalità del programma.

```
n_dont_care = ''
    for i in range(dv):
        n_dont_care += '-'
```

Questo semplice ciclo va a replicare il numero di - consecutivi che rispecchiano i gruppi di don't care da ignorare in base al dominio di conversione dato al circuito.

```
len_truth_table = len(truth_table)
i = 0
while i < len_truth_table:
    if '-' in truth_table[i]['inp']:
        truth_table[i]['inp'] = ''.join(truth_table[i['inp'])
        truth_table[i]['inp'] = [truth_table[i]['inp'[a:a+dv]
                                 for a in range(0, le(truth_table[i]['inp'], dv)]
        for a in range(len(truth_table[i]['inp'])):
            if truth_table[i]['inp'][a] == n_dont_care:
                truth_table[i]['inp'][a] = 'k'*le(n_dont_care)
        truth_table[i]['inp'] = ''.join(truth_table[i['inp'])
        new_lines = resolve_dont_care(truth_table[i],'inp')
        truth_table = truth_table[:i] + new_lines +truth_table[i+1:]
        len_truth_table = len(truth_table)
    i += 1
```

Per andare a differenziare quali siano i *don't care* da espandere e quali no a quest ultimi viene sostituito il simbolo - con un valore k in modo da poterli gestire meglio nelle funzioni successive. Alla fine delle operazioni verranno ripristinati con il simbolo corretto.

3.2.2.1 Resolve don't care

```
def resolve_dont_care(line, in_out):
    # Conto quanti - e creo 2^n nuove linee
    n_dc = pow(2, line[in_out].count('-'))

    input_entry = line[in_out]
    new_array = []

    for h in range(line[in_out].count('-')):
        val_array = create_0_1_array(n_dc, pow(2, h))
        if h == 0:
            for i in range(n_dc):
                it = 0
                new_line = []
                for j in range(len(input_entry)-1, -1, -1):
                    if input_entry[j] == '-' and it == 0:
                        new_line.append(val_array[(len(val_array)-1) - i])
                        it += 1
                    else:
                        new_line.append(line[in_out][j])
                new_array.append(new_line[::-1])
        else:
            it = 0
            for c, l in enumerate(new_array):
                for j in range(len(l)-1, -1, -1):
                    if l[j] == '-' and it == 0:
                        l[j] = val_array[(len(val_array)-1) - c]
                        it += 1
                it = 0

    if in_out == 'inp':
        return [{'inp':  l, 'out':  line['out']} for l in new_array]
    else:
        return [{'inp':  line['inp'], 'out':  l} for l in new_array]
```

Questa funzione prende in ingresso la linea da espandere e per ogni sua iterazione *(it)* va a sostituire ogni simbolo - con un valore di verità. La funzione restituirà la l'espansione della linea.

Come possiamo notare dal `return` della funzione questa operazione di espansione è possibile farla sia sugli input che su gli output.

3.2.2.2 create_0_1_array

```
def create_0_1_array(le, pad):
    count_1 = pad
    count_0 = pad
    return [str(1) if i % (count_1 + count_0) < count_1
        else str(0) for i in range(le)]
```

Questa funzione restituisce una tabella di verità data una line con all'interno un numero di *don't care* che necessitano di espansione.

Questo tabella creata verrà inserita al posto della linea con i *don't care* del circuito in così da avere la tabella corretta per la conversione.

3.3 La conversione in multi valore

Le funzioni viste nelle sezioni precedenti restituiscono come valore di output una matrice corrispondente alla tabella della verità sviluppata. Questa tabella verrà utilizzata dalla funzione di conversione seguendo lo schema spiegato all'inizio del capitolo.

```
def create_mv_truth_table(truth_array, dv):
    conv_truth = []
    for line in truth_array:
        line['inp'] = ''.join(line['inp'])
        line['inp'] = [line['inp'][i * dv:(i + 1) * dv]
            for i in range((len(line['inp']) + dv - 1) // dv)]
        if '-' not in line['out']:
            line['out'] = ''.join(line['out'])
            line['out'] = [line['out'][i * dv:(i + 1) * dv]
                for i in range((len(line['out']) + dv - 1) // dv)]
            l_supp_inp = []
            l_supp_out = []
            for val in line['inp']:
                if '-' not in val:
                    l_supp_inp.append(int(val, 2))
                else:
                    l_supp_inp.append('-')
            for val in line['out']:
                l_supp_out.append(int(val, 2))
            conv_truth.append({
                'inp':  l_supp_inp,
                'out':  l_supp_out
            })
    return conv_truth
```

La funzione prende in ingresso la matrice espansa creata precedentemente e il numero di bit da utilizzare per rappresentare il massimo numero del nuovo dominio multivalore.

La funzione inizialmente 'spezza' la stringa dei valori di input in gruppi di tanti elementi quanti i bit per rappresentare il massimo numero, successivamente viene controllato se il gruppo è composto da:

- **dont't care**: si procede sostituendo con un singolo simbolo '-'
- **numeri binari**: si procede con la conversione tramite il metodo int(), ad esso servono 2 parametri:
  - un numero o una stringa di numeri da convertire
  - la base a cui si vuole fare la conversione

Una volta scandito tutta la tabella di verità abbiamo il circuito convertito, l'utlimo passo ora consiste nell'andare a creare un file compatibile per la sintesi dei circuiti.

3.4 La creazione del file blfmv

Per i circuiti multi valore non viene usato il formato PLA in quanto viene utilizzato solamente per i circuiti binari. Utilizziamo un altro formato standard appositamente creato per la logica multi valore, il formato *blfmv*.

Questo formato deriva dal formato *blif* utilizzato per la logica binaria.

```
import string
mv_input = [i for i in list(string.ascii_lowercase)[
    :len(mv_table[0]['inp'])]]
mv_output = ['o{}'.format(i) for i in range(len(mv_table[0]['out']))]
with open('{}/blfmv/{}.mv'.format(working_dir, nomefile), 'w') as blif:
    blif.write('.model {}\n'.format(working_dir, nomefile))
    blif.write('.inputs {}\n'.format(
        ' '.join(map(str, mv_input))))
    blif.write('.outputs {}\n'.format(
        ' '.join(map(str, mv_output))))
    blif.write('.mv {} {}\n'.format(
        ', '.join(map(str, mv_input)), mv))
    blif.write('.mv {} {}\n'.format(
        ', '.join(map(str, mv_output)), mv))
    for count, out in enumerate(mv_output):
        blif.write('.table {} {}\n'.format(' '.join(map(str,mv_input)), out))
        for line in mv_table:
            blif.write('{} {}\n'.format(
                ' '.join(map(str, line['inp'])), line['out'][count]))
    blif.write('.end\n')
```

Questa funzione crea un file *.mv* relativo al circuito creato. Utilizziamo questa tipo di file e sintassi perché nelle fasi successive utilizzeremo per valutare e sintetizzare i circuiti dei software che accettano questa sintassi.

3.5 La sintesi

Una volta ottenuto sia il circuito booleano che quello multivalore procediamo con la sintesi dei circuiti, la sintesi dei circuiti si pone l'obiettivo di ridurre e ottimizzare la struttura dei circuiti stessi andando a diminuire il numero di ingressi e di tabelle della verità modo da avere dei costi totali minori.

I programmi utilizzati per la sintesi logica utilizzati sono entrambi sviluppati dall'università di Berkley e sono disponibili con licenza open source.

Per la sintesi durante l'analisi sperimentale sono state utilizzate 2 alternative:

- MVSIS
- ABC

3.5.1 MVSIS

Primo programma utilizzato per la sintesi, contiene diversi metodi da poter utilizzare in che utilizzano tecniche differenti con scopi differenti. La particolarità di questo tool consiste nel fatto che il programma accetti come input sia circuiti binari che multivalore e tratta quest ultimi come multivalore, non leggendoli e convertendoli in circuiti binari per poi sintetizzare. Il lato negativo dell'utilizzo di questo porgramma è che l'ultima versione di questo software risale al 2005 e quindi non è più mantenuto.

3.5.2 ABC

ABC, come la sua alternativa descritta sopra, mette a disposizione delle tecniche di sintesi con io vantaggio di avere degli script pronti che uniscono più metodi in modo da avere la certezza di non commettere errori o di utilizzare una concatenazione di metodi inefficace.

ABC è un'evoluzione di MVSIS e del più vecchio SIS, viene tuttora mantenuto ma, a differenza di MVSIS, se gli viene dato in ingresso un valore di input viene successivamente convertito in binario e poi si possono utilizzare i metodi di sintesi.

Una caratteristica molto utile di questi programmi sta nel fatto che essi accettino come parametri di ingresso degli script contenenti tutte le istruzioni da eseguire, successivamente mostrerò come tutto il processo di analisi viene automatizzato sfruttando questa caratteristica.

3.5.3 La sintesi tramite i software

La letteratura mette già a disposizione delle sequenze di metodi di sintesi e pulizia dei circuiti efficace ed efficiente, questi comandi si possono chiamare semplicemente dando in input ai due programmi un file contenente un alias per questa sequenza di istruzioni.

```
source abc_alias.abc
read_blif_mv ./prova/blfmv/amd.mv
strash
compress2
cl
resyn2
cl
write_blif_mv ./prova/blfmv/synth/amd.mv
```

Il programma, in questo caso *abc*, non fa altro che prendere prendere in input uno dei circuiti creati precedentemente e applicare i metodi di sintesi per poi creare un altro file contenente il circuito sintetizzato in modo da poter fare dei successivi confronti.

3.6 Il calcolo dei costi del circuito

Il costo del circuito viene calcolato sulla base 2 fattori fondamentali:

- Quanti input devono inviare entrambe le parti
- In base al dominio quanti valori devono inviare per ciascun valore di input

Per calcolare questo tipo di informazioni si devono effettuare delle operazioni di lettura sui file precedentemente creati andando a guardare:

- ingressi di ogni tabella di verità del circuito, andando a controllare quali dei valori di input devono essere inseriti da una delle parti e quali sono ingressi di valori di output delle altre tabelle.
- Dominio dei valori di input che devono inserire le parti.

3.6.1 blfmv vs pla

Per fare il confronto sopra citato si è preferito utilizzare file che abbiano la stessa sintassi di rappresentazione del circuito, sia nel caso multi valore che in quello booleano.

Il formato *pla* non dispone di un'alternativa per i file multivalore ma non è ancora supportata dai tool che abbiamo in utilizzo, si è deciso di utilizzare quindi il formato *blif* per rappresentare i circuiti binari. *Blif* è l'altranativa binaria a *blfmv*, la sintassi è la stessa con la differenza che non viene specificato il dominio dei valori di input in quanto sempre booleano.

La differenza con *pla* invece sta nel fatto che gli output non possono essere più di 1 per tabella nella rappresentazione, avrò quindi, per ogni nodo del circuito, tante tabelle quanti gli output di quel nodo. Questa tipologia di rappresentazione è utilizzata anche nei file *blfmv* quindi avrò uno comparazione 1:1 tra binario e multivalore.

Per effettuare questa conversione vengono in aiuto i tool *ABC* e *MVSIS*, entrambi contengono metodi di conversione automatica da *pla* a *blif*. Questa operazione viene effettuata tramite uno script contenente le istruzioni da eseguire e dato come parametro di ingresso al programma.

```
read_pla ./prova/pla/alu2.pla
write_blif ./prova/blif/alu2.blif
```

Ora abbiamo a disposizione tutti i file per poter fare il confronto dei costi

3.6.2 Implementazione

```
with open('{}/{}'.format(working_dir, circuito)) as circ:
    input = None
    output = None
    mv = int(0)
    table_array = []
    for line in circ.readlines():
        if '.inputs' in line.strip():
            input = line.strip().split(' ')[1:]
        if '.outputs' in line.strip():
            output = line.strip().split(' ')[1:]
        if '.mv' in line.strip():
            if mv < int(line.strip().split(' ')[-1]):
                mv = int(line.strip().split(' ')[-1])
        if '.table' in line.strip():
            l = line.strip().split(' ')
```

```
            table_array.append(
                {
                    'input':    l[1:len(l)-1],
                    'output':   l[-1]
                }
            )
        if '.names' in line.strip():
            l = line.strip().split(' ')
            table_array.append(
                {
                    'input':    l[1:len(l)-1],
                    'output':   l[-1]
                }
            )
    # i blif non hanno .mv, quindi gli do il valore di dominio
    if mv == 0:
        mv = 2
    return {
        'dominio':   mv,
        'input':    input,
        'output':   output,
        'tabelle':  table_array
    }
```

In questa funzione, dato un cricuito sia binario che mutlivalore, esso prende tutte le informazioni utili per poter effettuare il calcolo

```
costo = 0
for t in circuito['tabelle']:
    intersection = len(set(circuito['input']).intersectio(t['input']))
    costo = costo + pow(circuito['dominio'], intersection)
return costo
```

Fatto questo ho ottenuto tutti i valori di costo sia dei circuiti booleani che binari.

```
with open('calcolo_costi.csv', 'a') as file:
    file.write(
        'NOME CIRC;COSTO BOOLEANO;INPUT ALICE;INPUT BOB;DOMINIO MULTIVALORE;COSTO MULTI
    for blfmv in listdir('{}/blfmv'.format(working_dir)):
        if blfmv.endswith('.mv'):
            print(blfmv)
            circ_mv = info_circuito(
                '{}/blfmv/synth'.format(working_dir),blfmv)
            costo_mv = calcolo_costo_circuito(circ_mv)
            if len(circ_mv['input']) % 2 != 0:
```

```
                alice_var_mv = int(len(circ_mv['input'])/ 2)
                bob_var_mv = int(len(circ_mv['input']) /2) + 1
            else:
                alice_var_mv = int(len(circ_mv['input'])/ 2)
                bob_var_mv = int(len(circ_mv['input']) /2)
            circ_bool = info_circuito(
                '{}/blif/synth'.format(working_dir), '{}blif'.format(blfmv.split('.')[0]))
            costo_bool = calcolo_costo_circuito(circ_bool)
            if len(circ_bool['input']) % 2 != 0:
                alice_var_bool = int(len(circ_boo['input']) / 2)
                bob_var_bool = int(len(circ_bool['input') / 2) + 1
            else:
                alice_var_bool = int(len(circ_boo['input']) / 2)
                bob_var_bool = int(len(circ_bool['input') / 2)
            file.write('{};{};{};{};{};{};{};{}\n'.forma(blfmv.split('.')[
                0], costo_bool, alice_var_bool,bob_var_bool, circ_mv['dominio'],costo_mv, alice_v
```

Una volta ottenute tutte le informazioni, per avere una migliore visione per l'analisi, vengono salvate all'interno di una file *CVS* con la funzione sopra mostrata.

3.7 Automatizzazione dei processi

Tutte le fasi mostrate sopra, più altre secondarie, sono state automatizzate in modo che non si debbano effettuare operazioni ridondati per ogni circuito e generare perdite di dati causate da errori umani.

3.7.1 Struttura

```
.
├── abc_alias.abc           #alias con metodi sintesi ABC
├── automate.py             #automatizzatore dei processi
├── calcolo_costi.csv
├── calcolo_costi_no_synth.csv
├── facili  #directory contenente i file d'analisi
│   ├── blfmv
│   │   └── synth # circuiti sintetizzati
│   ├── blif
│   │   └── synth
│   └── pla_extended #pla espansi per debugging
├── lib_mv  # libreria con tutti le funzioni create
│   ├── automate_mvsis.py
│   ├── bin_to_mv.py
│   └── caloclo_costo.py
├── pla_to_blif.mvsis   #file con sintesi per MVSIS
├── synth_bool.abc      #file di sintesi  binaria per ABC
└── synth_mv.abc        #file di sintesi multivalore per ABC
```

```
working_dir = './prova'
if path.exists('{}/blif'.format(working_dir)) and pathexists('{}/blfmv'.format(working_
    rmtree('{}/blif'.format(working_dir))
    rmtree('{}/blfmv'.format(working_dir))

makedirs('{}/blif/synth'.format(working_dir))
makedirs('{}/blfmv/synth'.format(working_dir))
if path.exists('{}/synth_out.mvsis'.format(working_dir)):
    remove('{}/synth_out.mvsis'.format(working_dir))

if not path.exists('{}/pla'.format(working_dir)):
    print('CARTELLA PLA NON PRESENTE')
    exit
```

In questa fase viene creata la struttura delle directory che conterranno i file con i circuiti sintetizzati, se questa struttura è già presente vengono cancellate tutte le cartelle e i file all'interno e successivamente ricreate le cartelle vuote.

```
bin_to_mv_mcd(working_dir)
if exists('./pla_to_blif.mvsis'):
    remove('./pla_to_blif.mvsis')
pla_to_blif(working_dir)
```

in questa sezione vengono chiamate 2 utility create con lo scopo di generare dei file utilizzabili da *ABC* e *MVSIS*.

- **bin_to_mv_mcd**: ha lo scopo di creare i file *blfmv* di ogni circuito all'interno della directory di lavoro **pla_to_blif**: prende dalla directory di lavoro i file *pla* contenenti i circuiti e li converte in *blif*, come visto nei paragrafi precedenti.

```
call(['abc', '-F', './pla_to_blif.mvsis'])
synth(working_dir)
call(['abc', '-F', './synth_bool.abc'])
call(['abc', '-F', './synth_mv.abc'])
```

Una volta creati questi file compatibili con i nostri tool di sintesi essi vengono passati come parametro ad *ABC* o *MVSIS* s seconda delle esigenze di analisi.

```
if exists('./calcolo_costi.csv'):
    remove('./calcolo_costi.csv')
if exists('./calcolo_costi_no_synth.csv'):
    remove('./calcolo_costi_no_synth.csv')
```

Viene poi controllato se è presente già un file contenente dei costi e, se c'è, lo elimina

```
calcolo_costi_synth(working_dir)
calcolo_costi_no_synth(working_dir)
```

Le operazioni di calcolo dei costo vengono effettuate e infine viene generato il file *CSV* contenente i risultati.

Capitolo 4

Risultati sperimentali

Una volta che il il processo di conversione dei circuiti e che tutto il sistema di automazione dei processi si è proceduto con l'analisi sperimentale per studiare come se questo nuovo approccio possa risultare migliore a livello di costi prestazionali.

Per ottimizzare i costi sono stati utilizzati diversi approcci sfruttando entrambi i tool di sintesi disponibili e diversi parametri per la selezione del dominio.

Le operazioni computazionali sono state effettuate su un DELL-XPS 13 9350, la macchina he le seguenti specifiche:

- Processore: Intel i5-6200U@2.30GHz con 2 core e 4 thread
- Memoria: 8Gb LPDDR3@1866MHz
- Sistema Operativo: Ubuntu 21.04

4.1 MVSIS

Nelle prime fasi di sperimentazione si e' decido di utilizzare *MVSIS* come motore di sintesi si per i circuiti binari che per quelli multivalore. Questo primo approccio nasce dal voler replicare le operazioni che si fanno solitamente sulla logica binaria e applicarle a quella multivalore.

Per le operazioni di sintesi si sono utilizzati i seguenti comandi

```
read_blif_mv ./prova/blfmv/amd.mv
sweep
eliminate -l 1
simplify -m nocomp
eliminate -l 1
sweep
eliminate -l 5
simplify
sweep
eliminate -l 1
sweep
fullsimp -m nocomp
write_blif_mv ./prova/blfmv/synth/mvsis/amd.mv
```

Con questi comandi si prende il circuito iniziale, appena preso in ingresso, vengono fatte delle operazioni di pulizia e di rimozione di nodi inutili con i comandi *eliminatre* e *sweep*. successivamente tramite *simplify* e *fullsimp* il circuito viene semplificato sia a livello di nodi del circuito che nella sua interezza.

Questi comandi sono la conversione della variante *MVSIS* di *script.rugged* usato con il tool per la logica binaria *SIS*. Non si e' riusciti a fare una conversione 1:1 dello script in quanto non tutti i comandi sono replicabili.

In rete e in altri paper viene citato uno script chiamato *mvsis.rugged* che dovrebbe rappresentare la vera conversione adattata per il multivalore ma non si è riusciti a reperire questo file in nessun repository.

4.1.1 Conversione multivalore fissa

Come primo approccio alla conversione da binaria a multivalore abbiamo optato per lo stesso valore di dominio per tutti i circuiti. La scelta è stata quella di andare ad aggiungere altri 2 valori rispetto a ad un circuito binario utilizzando il modulo 3.

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTI	INPUT ALICE	INPUT BOB
amd	469	7	7	3	47568	3	4
tms	322	4	4	3	2048	2	2
pdc	2818	7	7	3	1310720	4	4
mlp4	594	4	4	3	784	2	2
apla	343	5	5	3	1029	2	3
f51m	261	4	4	3	340	2	2
m4	974	4	4	3	1601	2	2
newtpla2	48	5	5	3	2048	2	3
test1	1095	4	4	3	1280	2	2
m2	339	4	4	3	1121	2	2
br2	179	6	6	3	16384	3	3
alu1	83	6	6	3	5122	3	3
sqr6	197	3	3	3	276	1	2
bench	198	3	3	3	256	1	2
in5	446	7	7	3	117440512	6	6
newtpla1	33	5	5	3	1024	2	3
m3	571	4	4	3	1169	2	2
newapla	83	6	6	3	9728	3	3
l8err	454	4	4	3	337	2	2
t4	238	6	6	3	10240	3	3
br1	252	6	6	3	16384	3	3
fout	442	3	3	3	320	1	2
mp2d	136	7	7	3	7	3	4
alu2	225	5	5	3	1282	2	3
t3	131	6	6	3	4	3	3
p3	385	4	4	3	1408	2	2
m1	125	3	3	3	225	1	2
bcd_div3	54	2	2	3	32	1	1
alu3	139	5	5	3	4	2	3
in7	182	7	7	3	23085056	6	7

Questa tabella racchiude i risultati ottenuti dalla sintesi dei circuiti multivalore e dei circuiti binari. Possiamo notare che nella maggior parte dei casi il circuito binario mantiene ancora una grande efficienza rispetto a quello multivalore.

4.1.2 MVSIS e dominio variabile

Nella sperimentazione successiva si è voluto testare un dominio variabile in base al MCD, come descritto nel capitolo precedente. Nella seguente tabella viene mostrati i costi dei circuiti senza sintesi. Vediamo che nella maggiorn parte dei casi il dominio multivalore senza sintesi ha un costo minore, questo è figlio del minor numero di input che le parti durante la computazione dovranno inserire all'interno del protocollo.

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
amd	393216	7	7	4	196608	3	4
tms	4096	4	4	16	1024	1	1
pdc	2621440	8	8	4	1310720	4	4
mlp4	2048	4	4	16	512	1	1
apla	12288	5	5	4	6144	2	3
f51m	2048	4	4	16	512	1	1
m4	4096	4	4	16	1024	1	1
newtpla2	4096	5	5	4	2048	2	3
test1	2560	4	4	4	1280	2	2
m2	4096	4	4	16	1024	1	1
br2	32768	6	6	16	8192	1	2
alu1	32768	6	6	16	8192	1	2
sqr6	705	3	3	8	256	1	1
bench	512	3	3	4	256	1	2
in5	1835008	9	9	4	117440512	6	6
newtpla1	2048	5	5	4	1024	2	3
m3	4096	4	4	16	1024	1	1
newapla	40960	6	6	4	20480	3	3
l8err	1283	4	4	16	512	1	1
t4	32768	6	6	16	8192	1	2
br1	32768	6	6	16	8192	1	2
fout	640	3	3	4	320	1	2
mp2d	229376	7	7	128	32768	1	1
alu2	8192	5	5	4	4096	2	3
t3	32768	6	6	16	8192	1	2
p3	3074	4	4	4	1792	2	2
m1	768	3	3	8	256	1	1
bcd_div3	64	2	2	4	32	1	1
alu3	8192	5	5	4	4096	2	3
in7	1310720	9	9	4	335544320	6	7

Una vota applicato la sintesi dei circuiti i risultati sono i seguenti.

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
amd	469	7	7	4	47568	3	4
tms	322	4	4	16	1024	1	1
pdc	2818	7	7	4	1310720	4	4
mlp4	594	4	4	16	512	1	1
apla	343	5	5	4	1029	2	3
f51m	261	4	4	16	272	1	1
m4	974	4	4	16	1024	1	1
newtpla2	48	5	5	4	2048	2	3
test1	1095	4	4	4	1280	2	2
m2	339	4	4	16	1024	1	1
br2	179	6	6	16	8192	1	2
alu1	83	6	6	16	4097	1	2
sqr6	197	3	3	8	200	1	1
bench	198	3	3	4	256	1	2
in5	446	7	7	4	117440512	6	6
newtpla1	33	5	5	4	1024	2	3
m3	571	4	4	16	1024	1	1
newapla	83	6	6	4	9728	3	3
l8err	454	4	4	16	512	1	1
t4	238	6	6	16	8192	1	2
br1	252	6	6	16	8192	1	2
fout	442	3	3	4	320	1	2

I costi vengono notevolmente abbassati nel caso del booleano mentre in quello multivalore rimangono più elevati. Questo fenomeno potrebbe avere spiegazione nel fatto che la sintesi sui circuiti binari è più sviluppata e sono stati trovati metodi di sintesi più efficaci rispetto ad una logica multivalore.

Il tool *MVSIS* infatti, come detto prima, non è più sviluppato e mantenuto, si presenta infatti più lento nella sintesi ed ha una gestione della memoria che porta a dump di memoria durante l'analisi di circuiti di grandi dimensioni.

4.2 ABC

Notando che inizialmente il costo del circuito multivalore possiede dei costi più bassi per via del minor numero di input e che gli strumenti di sintesi siano però più ottimizzati per la logica binaria si è provato ad utilizzare un approccio ibrido.

Sfruttando come input un circuito multivalore ad ABC, questo tool converte automaticamente il circuito in binario per poi utilizzare i suoi algoritmi di sintesi ottimizzati.

```
read_blif_mv ./prova/blfmv/amd.mv
strash
compress2
cl
resyn2
cl
write_blif_mv ./prova/blfmv/synth/abc/amd.mv
```

Rispetto a prima però l'output post sintesi non sarà più un circuito multivalore ma uno binario.

4.3 ABC e dominio fisso

Come con *MVSIS* abbiamo inizialmente utilizzato un dominio fisso per fare una prima analisi.

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
amd	469	7	7	2	1860	7	7
tms	322	4	4	2	198	4	4
pdc	2818	7	7	2	6032	7	7
mlp4	594	4	4	2	764	4	4
apla	343	5	5	2	213	5	5
f51m	261	4	4	2	808	4	4
m4	974	4	4	2	643	4	4
newtpla2	48	5	5	2	48	5	5
test1	1095	4	4	2	354	4	4
m2	339	4	4	2	320	4	4
br2	179	6	6	2	110	6	6
alu1	83	6	6	2	83	6	6
sqr6	197	3	3	2	263	3	3
bench	198	3	3	2	47	3	3

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
in5	446	7	7	2	1336	7	7
newtpla1	33	5	5	2	33	5	5
m3	571	4	4	2	381	4	4
newapla	83	6	6	2	81	6	6
l8err	454	4	4	2	434	4	4
t4	238	6	6	2	197	6	6
br1	252	6	6	2	153	6	6
fout	442	3	3	2	214	3	3
mp2d	136	7	7	2	315	5	6
alu2	225	5	5	2	554	5	5
t3	131	6	6	2	160	6	6
p3	385	4	4	2	139	4	4
m1	125	3	3	2	116	3	3
bcd_div3	54	2	2	2	46	2	2
alu3	139	5	5	2	406	5	5
in7	182	7	7	2	1676	7	7

Possiamo notare come i risultati siano già migliorati e in molti casi la sintesi del circuito multivalore sia migliore di quella binaria tranne in sporadici casi.

4.4 ABC e dominio variabile

Come ultimo test, come in *MVSIS*, si utilizzato anche l'approccio MCD.

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
amd	469	7	7	2	1860	7	7
tms	322	4	4	2	184	4	4
pdc	2818	7	7	2	6032	7	7
mlp4	594	4	4	2	669	4	4
apla	343	5	5	2	213	5	5
f51m	261	4	4	2	344	4	4
m4	974	4	4	2	658	4	4
newtpla2	48	5	5	2	48	5	5
test1	1095	4	4	2	354	4	4
m2	339	4	4	2	350	4	4
br2	179	6	6	2	130	6	6
alu1	83	6	6	2	774	6	6
sqr6	197	3	3	2	234	3	3
bench	198	3	3	2	47	3	3

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
in5	446	7	7	2	1336	7	7
newtpla1	33	5	5	2	33	5	5
m3	571	4	4	2	388	4	4
newapla	83	6	6	2	81	6	6
l8err	454	4	4	2	338	4	4
t4	238	6	6	2	248	6	6
br1	252	6	6	2	136	6	6
fout	442	3	3	2	214	3	3
mp2d	136	7	7	2	1677	7	7
alu2	225	5	5	2	554	5	5
t3	131	6	6	2	354	6	6
p3	385	4	4	2	139	4	4
m1	125	3	3	2	112	3	3
bcd_div3	54	2	2	2	46	2	2
alu3	139	5	5	2	406	5	5
in7	182	7	7	2	1676	7	7

In questo caso possiamo notare che tra i due approcci non ci siano differenze stanziali, questo potrebbe essere dato da come *abc* legge i circuiti multivalore in input.

Beaver, D., S. Micali, e P. Rogaway. 1990. *The Round Complexity of Secure Protocols*. STOC '90. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/100216.100287.

Rogaway, Micali, P. 1991. «The round complexity of secure protocols (Doctoral dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science).»