Tesi

Andrea Locatelli

Indice

Capitolo 1

Secure multi-party computation

La Secure multi-party computation (SMPC) è una sottocategoria della crittografia che ha come obiettivo quello 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 garantisce 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.

Ci troviamo in un periodo in cui i dati diventano via via sempre più centrali e, ad oggi, infatti, oggetto di interesse per le aziende essi non saranno più semplici informazioni, bensì si trasformano in potenziali nuovi metodi di guadagno. La monetizzazione dei dati è il modello di business primario comune a molte delle più grandi aziende, nonché 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 è possibile trovare combinando diverse fonti dati. E' necessario però considerare tutte le problematiche derivanti da questa dinamica

Il sottoprodotto di questo processo è, ad esempio, una possibile violazione della privacy individuale, in quanto le aziende posseggono molte informazioni personali dei loro utenti.

Sono state pensate dunque, per far fronte a questo precedente, soluzioni in cui sia possibile utilizzare fonti dati diverse senza la necessità di centralizzare l'informazione. In questo modo è possibile garantire che nessuna informazione venga rivelata nel corso delle operazioni.

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

Un protocollo di calcolo a più parti deve essere sicuro per essere efficace. Nella crittografia moderna, la sicurezza di un protocollo è legata a una dimostrazione di sicurezza. La dimostrazione di sicurezza è una dimostrazione matematica in cui la sicurezza di un protocollo è ridotta a quella della sicurezza delle sue primitive sottostanti. Tuttavia, non è sempre possibile formalizzare la verifica di sicurezza del protocollo crittografico basata sulla conoscenza delle parti e sulla correttezza del protocollo. Per i protocolli MPC, l'ambiente in cui il protocollo opera è associato al paradigma del mondo reale/mondo ideale. Non si può dire che le parti non imparino nulla, poiché devono imparare l'output dell'operazione, e l'output dipende dagli input. Inoltre, la correttezza dell'output non è garantita, poiché la correttezza dell'output dipende dagli input delle parti, e si deve supporre che gli input siano corrotti.

Il paradigma del mondo reale/mondo ideale afferma due mondi:

- Nel **modello del mondo ideale**, esiste una parte incorruttibile fidata a cui ogni partecipante al protocollo invia il suo input. Questa parte fidata calcola la funzione da sola e rimanda l'output appropriato ad ogni parte.
- Al contrario, nel **modello del mondo reale**, non esiste una parte fidata e tutto ciò che le parti possono fare è scambiare messaggi tra loro. Un protocollo è detto sicuro se non si può imparare di più sugli input privati di ogni parte nel mondo reale di quanto si potrebbe imparare nel mondo ideale. Nel mondo ideale, nessun messaggio viene scambiato tra le parti, quindi i messaggi scambiati nel mondo reale non possono rivelare alcuna informazione segreta.

Il paradigma mondo reale/mondo ideale fornisce una semplice astrazione delle complessità di MPC per consentire la costruzione di un'applicazione sotto la premessa che il protocollo MPC nel suo nucleo sia in realtà un'esecuzione ideale. Se l'applicazione è sicura nel caso ideale, allora è sicura anche quando viene eseguito un protocollo reale.

A differenza delle applicazioni crittografiche tradizionali, come la crittografia o la firma, si deve assumere che l'avversario in un protocollo MPC sia uno dei giocatori impegnati nel sistema (o che controlla le parti interne). Quella/e parti corrotte possono colludere per violare la sicurezza del protocollo.

Sia n il numero di parti nel protocollo e t il numero di parti che possono essere avversarie. I protocolli e le soluzioni per il caso di $t < n/2$ (cioè quando si assume una maggioranza onesta) sono diversi da quelli in cui non viene fatta tale assunzione. Quest'ultimo caso include l'importante caso di calcolo a due parti in cui uno dei partecipanti può essere corrotto, e il caso generale in cui un numero illimitato di partecipanti è corrotto e collude per attaccare i partecipanti onesti.

Gli avversari affrontati dai diversi protocolli possono essere classificati in base a quanto sono disposti a deviare dal protocollo. Ci sono essenzialmente due tipi di avversari, ognuno dei quali dà luogo a diverse forme di sicurezza (e ognuno si adatta a diversi scenari del mondo reale):

- **Sicurezza semi-onesta (passiva)**: In questo caso, si assume che le parti corrotte cooperino semplicemente per raccogliere informazioni dal protocollo, ma non si discostino dalle specifiche del protocollo. Questo è un tipo di avversario semplice e ingenuo, che produce una sicurezza debole in situazioni reali. Tuttavia, i protocolli che raggiungono questo livello di sicurezza impediscono la perdita involontaria di informazioni tra le parti (altrimenti collaboranti), e sono quindi utili se questa è l'unica preoccupazione. Inoltre, i protocolli nel modello semi-onesto sono abbastanza efficienti, e sono spesso un primo passo importante per raggiungere livelli più alti di sicurezza.
- **Sicurezza malevola (attiva)**: In questo caso, l'avversario può arbitrariamente discostarsi dall'esecuzione del protocollo nel suo tentativo di ingannare. I protocolli che raggiungono la sicurezza in questo modello forniscono una garanzia di sicurezza molto alta. Nel caso di maggioranza delle parti che si comportano scorrettamente: L'unica cosa che un avversario può fare nel caso di maggioranza disonesta è far sì che le parti oneste interrompano avendo rilevato l'imbroglio. Se le parti oneste ottengono l'output, allora hanno la garanzia che sia corretto. La loro privacy è sempre preservata.

La sicurezza contro gli avversari attivi porta tipicamente ad una riduzione dell'efficienza che porta alla sicurezza nascosta, una forma ridotta di sicurezza attiva. La sicurezza nascosta cattura situazioni più realistiche, in cui gli avversari attivi sono disposti a barare ma solo se non vengono scoperti. Per esempio, la loro reputazione potrebbe essere danneggiata, impedendo la futura collaborazione con altre parti oneste.

Quindi, i protocolli che sono segretamente sicuri forniscono meccanismi per garantire che, se alcune delle parti non seguono le istruzioni, allora sarà notato con alta probabilità. In un certo senso, gli avversari segreti sono quelli attivi costretti ad agire passivamente a causa di preoccupazioni esterne non crittografiche (ad esempio il business). Questo meccanismo stabilisce un ponte tra i due modelli nella speranza di trovare protocolli che siano efficienti e abbastanza sicuri nella pratica.

Come molti protocolli crittografici, la sicurezza di un protocollo MPC può basarsi su diversi presupposti:

- **Computazionale** (cioè basata su qualche problema matematico, come la fattorizzazione) o **incondizionata**, cioè basata sull'indisponibilità fisica dei messaggi sui canali (di solito con qualche probabilità di errore che può essere resa arbitrariamente ridotta).
- Il modello potrebbe assumere che i partecipanti utilizzino una rete sincronizzata, dove un messaggio inviato a un "tick" arriva sempre al "tick" successivo, o che esista un canale di trasmissione sicuro e affidabile, o che esista un canale di comunicazione sicuro tra ogni coppia di partecipanti dove un avversario non può leggere, modificare o generare messaggi nel canale, ecc.

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

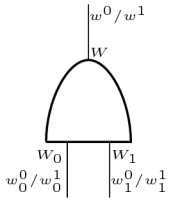
Yao Garbled Circuit

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 in questa dinamica. Senza perdita di generalità, considererò solo le porte logiche con due fili di ingresso e un filo di uscita. Come spiegato precedentemente, 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 e 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, adottando 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 infatti, semantica e sintassi sono fondamentalmente le stesse: vengono assegnati ad ogni filo due possibili valori semantici, cioè Vero o Falso, che sintatticamente si denotano come un segnale con valori 1 o 0 rispettivamente.

In questi segnali erano pubblici, gli stessi segnali sono associati ad ogni filo e chiunque può dedurre dal segnale quale sia il suo valore semantico. Questa condizione cambia però 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 et al., 1990a) e poi spiegato chiaramente nella tesi di Phillip Rogoway (Beaver et al., 1990b), 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

Il meccanismo di Permute-and-point 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. Permette dunque 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. In questo modo 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 rappresentazione chiara di questo trucco, quei bit possono essere visti come una coppia di colori, come rappresentato nella figura 2.2, in cui si notano le modalità con cui la tabella di verità viene modificata in modo da tenere conto di questo metodo.

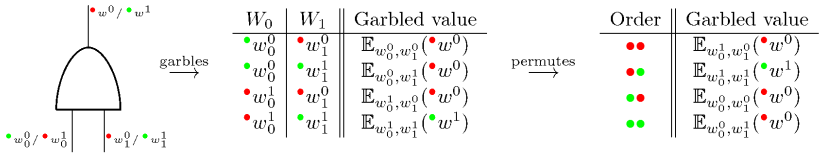


Figura 2.2: AND garbled con permute on point

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

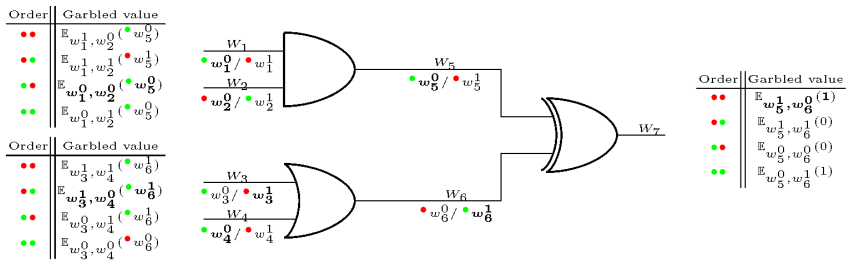


Figura 2.3: AND garbled con permute on point

Vediamo ora 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 nel paragrafo riguardante.

Bob comincerà quindi 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 $\bullet w_5^0$.

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

Potrà ora 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, ne 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 questo sistema 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

La sintesi dei circuiti

Nel capitolo precedente abbiamo discusso dell'importanza della presenza di un circuito logico all'interno del protocollo creato da Yao. La scelta di un circuito efficiente rende la computazione e i tempi di calcolo delle operazioni non elevati. Ciò che rende efficace la scelta è trovare il giusto bilanciamento tra numero di input che le 2 parti coinvolte devono immettere nel circuito e, nel caso di un dominio multivalore, scegliere un dominio non troppo elevato che andrebbe ad immettere, per ogni valore di input, troppi valori che porterebbero ad aumentare i costi totali del circuito.

La scelta di questi circuiti viene facilitata dall'utilizzo di strumenti chiamati *sintetizzatori*, questi tool sono stati creati per andare a ridurre le dimensioni dei circuiti dati loro in ingresso. All'interno di questo lavoro sono stati utilizzati 2 tool differenti: **MVSIS** e **ABC**, entrambi sono stati sviluppati nel tempo dall'università di Berkley, essi contengono diversi metodi di sintesi capaci di andare a eliminare nodi e ridondanze superflue nei circuiti e sintetizzare il circuito andandone a cambiare radicalmente la struttura interna de i nodi.

3.1 MVSIS

MVSIS è il primo tool in grado di manipolare circuiti con una logica multivalore. Per la sua creazione si è preso come modello **SIS**, un tool sviluppato e specifico per la logica binaria, cercando di mantenerne simile la logica di funzionamento.

3.1.1 Design specification

Un circuito multivalore (MV circuit) può essere dato come valore di input a **MVSIS** tramite l'apposito comando (`read_blifmv`), oltre a circuiti multivalore possono essere dati in input anche circuiti binari. Internamente, la rappresentazione del progetto è una rete di nodi MV; ogni nodo rappresenta una rappresenta una funzione MV con una singola uscita a più valori. Un'importante distinzione con alcuni altri metodi multivalore è che ogni variabile del nodo può avere un dominio diverso. L'intervallo per la variabile y_k è rappresentato dall'insieme $P_k={0,1,...,p_k -1 }. La funzione il cui esito dà UN SET dei mintermi (Una sola uscita con 0 o 1) per i quali $f_k = i$ (la funzione al nodo k è uguale al valore i) è chiamata *i-set* della funzione f_k e viene memorizzata in forma SOP (Somma di prodotti). Nel caso di logica a 2 valori, l'insieme 0 corrisponde all'off-set e l'insieme 1 all'on set. Una variabile MV yk è associata all'uscita del nodo k. Un margine congiunge k a j se uno qualsiasi degli i-set di j dipende esplicitamente da y_k. La rete ha un insieme di ingressi primari (che possono essere tutti a più valori) e un insieme di nodi di ingressi primari.

3.1.2 Semplificazione dei nodi

L'i-set (uno per ogni valore di output) di un nodo multivalore può essere semplificato con 2 comandi:

- SIMPLIFY
- FULLSIMP

FULLSIMP tra le 2 opzioni è quella che garantisce una semplificazione più efficace.Utilizza il CODC (Compatible Observability Don't Cares) e MV-image computation. Ogni i-set del nodo è poi semplificato da ESPRESSO-MV utilizzando i valori ricavati dalle 2 tecniche precedenti.

3.1.3 Kernel e Cube Extraction

Un altro importante step nell'ottimizzazione del circuito consiste nell'applicare metodi algebrici per estrarre nuovi nodi che hanno divisori comuni per altri nodi. Si sono sviluppate nuove tecniche algebriche per la logica MV che trattano uniformemente le variabili binarie e multi-valore. Esse includono metodi per trovare sottoespressioni comuni, divisione semi-algebrica, decomposizione di una rete multi-valore e fattorizzazione di una forma SOP. I comandi rilevanti e le brevi descrizioni delle loro capacità sono elencati di seguito.

1. **FX**: estrae buoni divisori comuni e crea nuovi nodi nella rete, reimpostando altri nodi in termini di questi.
2. **Decomp**: fa una completa fattorizzazione multivaloriale degli i-sets di ogni nodo e suddivide i nodi secondo queste fattorizzazioni. Dopo questo, **resub**, usando la divisione algebrica multivalore, può essere eseguita per eliminare i fattori duplicati.

3.1.4 Altri comandi per la manipolazione dei nodi

1. **Collapsing**: converte l'intera rete multilivello in modo che le forme SOP per ogni uscita siano in termini di soli ingressi primari. Così il numero di nodi nella rete sarà esattamente il numero di uscite primarie.
2. **Merging**: Prende tutti i nodi e forza un'unione creando un singolo nodo multi-valore costruendo in *i-set* per ogni combinazione di valori creata. Se ci sono più *i-set* creati uguali vengono uniti in un singolo nodo.
3. **Encoding**: è come l'inverso del merging di funzioni binarie. Cerca di trovare una buona codifica binaria per ogni variabile multi-valore nella rete, compresi gli ingressi e le uscite primarie. Alla fine, ogni segnale è codificato come un segnale binario. Quindi un file binario può essere scritto. Formato da 2 fasi:

    1. Inizia dagli ingressi e per ogni nodo, determina se uno dei suoi fanin può essere usato per codificare parzialmente il nodo.
    2. inizia dalle uscite e in ordine topologico inverso lavora a ritroso fino agli ingressi primari. Ad ogni nodo, le sue uscite sono codificate utilizzando le informazioni su come sono utilizzati i suoi fanout

4. **Pair decoding**: Simile al merging ma utilizza un altro modo per scegliere quali modi unire.
5. **Bi-decomposition**: crea dei nodi multi-valore intermedi. Prende una rete MV appiattita o parzialmente appiattita e ne genera un'altra composta da porte MAX e MIN multi-valutate a due ingressi e da iterali multi-valutati. Vengono sfruttate sia l'incompletezza della specifica iniziale che le flessibilità generate nel processo di composizione.

3.2 ABC

ABC è un sistema software in sviluppo per la sintesi e la verifica di circuiti logici sequenziali binari che appaiono in progetti hardware. ABC combina un'ottimizzazione logica scalabile basata su And-Inverter Graphs (AIGs), una mappatura tecnologica basata su DAG per tabelle di look-up e celle standard e algoritmi innovativi per la sintesi e la verifica sequenziale.

Questo programma nasce dalle esperienze assimilate dall'uso di SIS, VIS e MVSIS. Questi sistemi non forniscono un ambiente di programmazione flessibile per implementare le recenti innovazioni. In particolare, l'ambiente SIS è obsoleto e piuttosto inefficiente nella gestione di grandi circuiti. VIS, progettato come strumento di verifica formale per specifiche multi-valore, non fornisce abbastanza flessibilità per la sintesi binaria. MVSIS nonostante il suo ampio utilizzo:

- Le strutture dati e gli algoritmi di base di MVSIS possono essere resi considerevolmente più semplici e facili da usare assumendo reti binarie,
- Un posto centrale nel nuovo sistema dovrebbe essere dato a una nuova struttura dati, le AIG (reti logiche multilivello composte da AND e inverter a due ingressi), che promette miglioramenti nella qualità e nel tempo di esecuzione della sintesi e della verifica.

La comprensione di queste problematiche ha spinto a ri-sviluppare i pacchetti di base di MVSIS risultando in un nuovo ambiente di programmazione chiamato ABC. Come suggerisce il nome, l'obiettivo primario è quello di mantenere strutture di dati semplici e flessibili per una vasta gamma di applicazioni.

L'obiettivo del progetto ABC è quello di fornire un'implementazione pubblica degli algoritmi di sintesi combinatoria e sequenziale allo stato dell'arte e, allo stesso tempo, creare un ambiente open-source, in cui tali applicazioni possano essere sviluppate e confrontate. La versione attuale di ABC può ottimizzare/mappare/realizzare progetti industriali a livello di gate con 100K porte e 10K elementi sequenziali cont tempi di calcolo non superiori al minuto in un computer moderno.

3.2.1 Sintesi combinatoria

I metodi di logica combinatoria utilizzati da ABC all'interno dei suoi script (resyn e resyn2) sono tipicamente 10-100 volte più veloci se comparati agli script utilizzati all'interno di SIS o MVSIS.

In ABC, i confini dei nodi sono inizialmente distrutti dall'hashing strutturale (comando *strash*), che trasforma una rete logica in un AIG. I confini possono essere ricreati su richiesta usando il comando *renode*, che può essere visto come un inverso del comando SIS *eliminate*. Nel flusso di sintesi presentato sopra la logica viene trasformata a livello di AIG senza creare nodi.

- **Balance**: prende come ingresso un AIG e lo bilancia.
- **Cleanup**: Rimuove i nodi logici che non sono a ventaglio in PO e latches.
- **Collapse**: Collassa tutto il circuito ad una rete ad un solo livello. Le funzioni dei nodi sono rappresentate utilizzando le BDD.
- **Dsd**: Applica la decomposizione disjoint-support utilizzando l'algoritmo di Bertacco/Damiani (Bertacco and Damiani, 1997).
- **fx**: Rileva la logica di condivisione estraendo i divisori a due cubi e i divisori a due lettere a un cubo usando l'algoritmo di Rajski/Vasudevamurthi (Silveira et al., 1992).
- **Multi**: Espande l'AIG a due ingressi in una rete di porte AND a più ingressi.
- **refactor**: Esegue il collasso iterativo e il refactoring dei coni logici nell'AIG, che cerca di ridurre il numero di nodi AIG e il numero di livelli logici.
- **renode**: Assume che l'input sia un AIG. Crea i confini dei nodi in questo AIG e collassa la logica intermedia per formare nodi più grandi.
- **rewrite**: Assume che l'input sia un AIG. Crea i legami tra i nodi in questo AIG e collassa la logica intermedia per formare nodi più grandi.
- **rr**: Esegue la rimozione della ridondanza per le AIG.
- **strash**: Trasforma la rete corrente in un AIG tramite un livello di hashing strutturale. L'AIG risultante è una rete logica composta da porte AND a due ingressi e invertitori rappresentati come attributi complementari sui bordi. L'hashing strutturale è una trasformazione puramente combinatoria, che non modifica il numero e le posizioni delle porte.
- **sweep**: sweep classico applicabile alla rete logica corrente che risulta in una rete logica. Sweep esegue i seguenti compiti: rimuove i nodi dangling (nodi senza fanout), collassa i buffer e gli inverter nei loro fanout, propaga le costanti, e rimuove i fanin duplicati. Sweep non può essere applicato a un AIG perché un AIG è strutturalmente hashed e quindi non ha buffer, invertitori e nodi costanti non propagati. Per rimuovere i nodi dandling nella rete logica, usate cleanup.

3.2.2 Sintesi sequenziale

La sintesi sequenziale trasforma la rete corrente modificando la sua logica insieme agli elementi di memorizzazione (latches o flip-flops), se presenti. La rete risultante può avere una codifica di stato e uno spazio di stato diversi rispetto alla rete originale, ma le due reti sono sequenzialmente equivalenti (cioè, partendo dagli stati iniziali, per le stesse sequenze di vettori d'ingresso, producono sequenze identiche di vettori d'uscita).

- **Cycle**: Simula la rete sequenziale con un input casuale e aggiorna il suo stato attuale.
- **init**: Ripristina gli stati iniziali di tutte le porte della rete corrente.
- **lcorr** - Implementazione suddivisa di registro-corrispondenza usando l'induzione semplice. Rileva e fonde registri sequenzialmente equivalenti.
- **retime**: Implementa diversi tipi di retiming:
  - most forward;
  - most backward;
  - minimum-register;
  - minimum-delay euristico;
  - delay-optimal retiming (Pan, 1997)

  Le porte sono condivise in modo ottimale attraverso gli archi di fanout quando il circuito è trasformato dall'AIG in una rete logica. Il calcolo degli stati iniziali dopo il retiming è ridotto a un problema SAT, che viene risolto usando MiniSat.
- **scleanup**: Esegue la pulizia sequenziale (rimuove i nodi e le latches che non sono a fanout).
- **ssw**: Implementa la segnali corrispondenti usando l'induzione K-step. Rileva e fonde i nodi sequenzialmente equivalenti.
- **undc**: Si può utilizzare questo comando prima di eseguire la verifica sequenziale, per convertire i registri con gli stati iniziali don't-care in registri con uno stato iniziale costante-0.
- **xsim**: Esegue la simulazione con valore X della rete sequenziale corrente.

Capitolo 4

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 concentrandoci sul fattore di costo computazionale delle operazioni.

Siamo partiti quindi selezionando un set di circuiti binari con differente numero di ingressi, uscite e livelli. Con questa base ci si è poi concentrati sul trovare una metodologia di conversione dei circuiti per portarli da un 'classico' dominio booleano ad uno multi-valore dove i valori non sono più rappresentati sa 0 e 1 ma da da un set più ampio di valori ma mantenendo le funzionalità del circuito. Una volta ottenuta una controparte per ogni circuito si è proceduto con un confronto dei circuiti in modo da individuare quali di questi set siano più efficaci.

4.1 Studio del circuito binario

Analizzando i circuiti per testare la nostra tesi abbiamo dovuto sviluppare una tecnica di conversione dei circuiti binari di partenza. Le caratteristiche di una naturale rappresentazione di un circuito consiste nella sua divisione in nodi, ognuno di essi rappresentato da una tabella di verità i cui input e output vanno a collegarsi tra di loro tramite dei collegamenti. Soffermandoci sulle tabelle di verità abbiamo notato come andando ad applicare un raggruppamento di n bit per ogni riga della tabella si riuscirebbe a rappresentare un valore di dominio più grande di quello binario del circuito in analisi riducendo il numero di input richiesti dal gate.

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

Con questo approccio non andiamo a snaturare quella che è la logica della tabella originale poiché, sapendo il qual è il nuovo dominio applicato, riusciamo ad invertire il processo di conversione ritornando al circuito originale.

I vincoli di questo approccio di conversione dei circuiti booleani risiede nel fatto che, per ora, la conversione possa essere fatta solamente con un numero pari di input e che il numero di output sia un divisore del numero di bit utilizzati per il raggruppamento.

Una volta definito il metodo di conversione si è passati alla scelta del nuovo dominio da applicare ai circuiti con l'obiettivo di trovare il giusto compromesso per ottenere:

- Una riduzione degli input che faccia diminuire il costo del circuito di ingresso per il circuito di ingresso
- Utilizzare un dominio che risulti controproducente: avere un dominio superiore a quello booleano ma troppo elevato significherebbe che, per ogni input, ogni parte dovrebbe portare inserire tanti valori di input quanti quelli del dominio richiesto e, se si sceglie un dominio troppo grande, nonostante la riduzione degli input ci si troverebbe comunque con un costo totale del circuito elevato. Si è deciso quindi di sperimentare 2 approcci alla scelta di questo valore, uno statico e uno dinamico.
  - **Statico**: per ognuno dei circuiti andremo ad applicare lo stesso valore di dominio, il valore scelto è 3, andremo quindi a raggruppare ogni 2 valori per riga della tabella di verità;
  - **Dinamico**: ogni circuito viene analizzato andando a trovare il numero di input e output, successivamente viene calcolato il valore di M.C.D. e:
    * Se il valore trovato consente alla tabella di verità convertita di avere almeno 2 valori di input si utilizza quel valore per raggruppare i bit delle righe, altrimenti il valore viene dimezzato
    * Se il valore risultante è un numero che creerebbe un dominio troppo grande viene utilizzato il dominio 3, come nel caso statico.

Definite tutte le logiche di conversione si è dovuto cercare un formato di file che rappresentasse i circuiti che rispettasse le esigenze per applicare tutte le elaborazioni sopra descritte e che sia utilizzato come uno standard di rappresentazione in modo che i risultati ottenuti possano essere utilizzati dai software di sintesi che andremo ad utilizzare successivamente durante l'elaborazione della sperimentazione. Si necessita dunque di una rappresentazione che abbia la possibilità che i circuiti con output multipli siano rappresentati sulla stessa linea all'interno della tabella della verità.

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

4.1.1 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
```

Questo `PLA` di esempio rappresenta un circuito composto da una sola tabella di verità, la sintassi va a descrivere:

- `.i`: numero di valori di input;
- `.o`: numero di valori di output;
- `.ilb`: nomi e ordine delle variabili di input;
- `.ob`: nomi e ordine delle variabili di output.

Il resto del file rappresenta la tabella di verità con i possibile valori che si ottengono combinando i valori di input_file per ottenere gli output.Il circuito sopra descritto presenta 4 variabili di input e 2 valori di output. Come possiamo notare all'interno della tabella di verità non tutti i valori sono booleani ma è presente anche il valore `-`, esso prende il nome di **don't care** e può assumere 2 significati:

- Quando `-` è presente negli output di una riga significa che l'output per quella determinata riga non mi interessa;
- Quando `-` è presente negli input significa che in quella posizione la variabile a cui fa riferimento può assumere un qualsiasi valore non andando ad influire sul valore di output che si andrà ad ottenere.

4.1.2 La gestione dei don't care durante la conversione

I dont't care sono una caratteristica dei file pla che deve essere gestita in modo funzionale ai cambi di dominio che andremo a fare all'interno delle tabelle di verità dei nostri circuiti in quanto, se affrontato nel modo sbagliato, modificherebbe la logica dell'intero circuito rendendo la conversione errata. Per spiegare meglio come tratteremo questa caratteristica dei **PLA** presenterò l'esempio di un esempio utilizzando un circuito semplice che 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
```

Una volta deciso il nostro nuovo dominio di conversione sappiamo di quanti bit dobbiamo raggruppare per rappresentare il nostro nuovo dominio, nel nostro caso essendo il domino 3, il numero di bit per rappresentare il valore più grande di questo dominio abbiamo bisogno di un raggruppamento a 2 bit. Procediamo quindi a scandire riga per riga il circuito raggruppando i bit a blocchi di 2 tutti i valori di input e di output.

```
0- -0 00
00 01 01
0- 11 --
1- 11 01
01 01 10
10 -- 01
11 -- 00
```

Una volta creati questi cluster di bit ci accorgiamo come i dont't care in alcuni casi ci rendano impossibile la conversione al nuovo dominio, per risolvere questo problema dovremo andare a sviluppare questi valori di don't care in 'normali' valori booleani.

Non è sempre necessario andare a sviluppare i don't care, andiamo a differenziare 2 possibili casistiche di intervento:

- Quando il numero di '-' è uguale al numero di bit del raggruppamento multivalore, in questo caso non c'è la necessità di sviluppare i valori, sarà sufficiente inserire al posto della coppia un unovo – rappresentante il don't care.

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

- Quando nel gruppo di bit raggruppati il numero di – è minore del numero di bit richiesti. In questo caso dobbiamo procedere con l'espansione del dont't care a valori di verità in modo da poter affrontare correttamente la conversione nel nuovo dominio.

$$\begin{array}{ccc} 1- & 11 & 01 \\ & \downarrow & \\ 10 & 11 & 01 \\ 11 & 11 & 01 \\ & \downarrow & \\ 2 & 3 & 1 \\ 3 & 3 & 1 \end{array}$$

Così facendo è possibile 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 è dunque possibile procedere con l'analisi dei circuiti proseguendo con l'attività di sintesi logica.

4.2 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. Questa dinamica fa si che il sistema sia "ready to use" una volta installato il linguaggio di programmazione, se non già presente all'interno del SO.

4.2.1 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 input un circuito binario in formato PLA e, riga per riga, analizza le informazioni in base al prefisso all'interno di ogni riga del file che va a differenziare le informazioni del circuito. Scandendo il fil recupera:

- Tabella delle verità
- Numero di input
- Numero di output
- Array contenente il nome delle variabili di input;
- Array contenente il nome delle variabili di output;

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

4.2.2 Espansione dei don't care

Una volta ottenute tutte le informazioni disponibili dal circuito in input dobbiamo andare ad analizzare le righe delle tabelle di verità per individuare ed espandere i dont't care secondo le regole descritte nel capitolo precedente.

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

Con questo ciclo vado a creare un array di - consecutivi che rispecchiano i cluster di don't care da ignorare nel durante l'analisi delle tabelle di verità. Avrò quindi un array di n - tanti quanti il numero di bit per rappresentare i numeri del nuovo dominio.

```
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
```

Tramite questo ciclo *while* l'intero circuito viene scandito e:

- I valori di input vengono divisi in cluster di n bit per rappresentare i nuovi valori del dominio;

```
truth_table[i]['inp'] = [
   truth_table[i]['inp'[a:a+dv]
   for a in range(0, le(truth_table[i]['inp'], dv)]
 ]
```

- Se all'interno dei cluster sono presenti solamente don't care il valore - viene sostituito con il valore k per differenziarli dai valori dont't care che successivamente verranno espansi

```
if truth_table[i]['inp'][a] == n_dont_care:
  truth_table[i]['inp'][a] = 'k'*le(n_dont_care)
```

Ora la tabelle è finalmente pronta per l'espansione dei don't care tramite l'appostia funzione sviluppata

Si è utilizzato un ciclo *while* in questa funzione nonostante in Python sia meno efficiente vista la natura della sua implementazione in quanto la lunghezza della tabella della verità aumenta dinamicamente con l'andare dell'espansione dei don't care individuati.

4.2.2.1 Resolve don't care

La funzione inizia andando ad individuare quanti don't care sono presenti nella linea

```
def resolve_dont_care(line, in_out):
  n_dc = pow(2, line[in_out].count('-'))
```

Successivamente viene iniziato un ciclo che scandisce ogni elemento della linea della tabella di verità, viene creato un array binario della di tante righe quante saranno dopo l'espansione dei don't care.

```
for h in range(line[in_out].count('-')):
  val_array = create_0_1_array(n_dc, pow(2, h))
```

Successivamente vengono differenziate 2 casistiche:

- Prima iterazione

```
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])
```

- Tutte le successive

```
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
```

Ogni riga viene scandita a partire dal fondo per iniziare a convertire i don't care corrispondenti ai valori meno significativi, sfruttando l'array binario creato spora andiamo a sostituire il simbolo '-' con con il valore booleano corrispondente e creare una nuova linea di un array che conterrà tutti i valori ottenuti dall'espansione.

```
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]
```

Come possiamo notare dal `return` della funzione questa operazione di espansione è possibile sia sugli input che su gli output. Come detto nel capitolo precedente anche l'espansione dei valori di don't care degli output è importante per non perdere l'integrità del circuito.

L'oggetto restituito dalla funzione è un array contenente un vocabolario con la stessa struttura che abbiamo usato durante l'estrazione dei dati dal file `PLA`, questo array sostituirà la riga contenente i don't care espansi.

4.2.2.2 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.

Una volta terminato tutto il ciclo tutti i dont't care saranno risolti e la tabella sarà pronta per la conversione nel nuovo valore di dominio.

4.3 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.

Inizialmente la funzione 'spezza' la stringa dei valori di input in gruppi di tanti elementi quanti sono 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 scandita tutta la tabella di verità siamo giunti al circuito convertito, l'utlimo passo ora consiste nell'andare a creare un file compatibile per la sintesi dei circuiti.

4.4 La creazione del file blfmv

Per i circuiti multi valore non viene adottato il formato PLA in quanto viene utilizzato solamente per i circuiti binari. Facciamo uso di 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. Scegliamo questa tipo di file e sintassi poiché nelle fasi successive utilizzeremo dei software che accettano questa sintassi per valutare e sintetizzare i circuiti.

4.5 La sintesi

Una volta ottenuto sia il circuito booleano che quello multivalore procediamo con la sintesi dei circuiti. Quest'ultima 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 in modo da avere dei costi totali minori.

I programmi utilizzati per la sintesi logica 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

4.5.1 MVSIS

E' il primo programma utilizzato, al suo interno troviamo diversi metodi di sintesi da poter applicare che sfruttano tecniche differenti. La particolarità di questo tool consiste nel fatto che il programma accetti come input sia circuiti binari che multivalore. 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.

4.5.2 ABC

ABC, come la sua alternativa descritta sopra, mette a disposizione delle tecniche di sintesi con il 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 esso viene successivamente convertito in binario e solo in seguito è possibile utilizzare i metodi di sintesi.

Una caratteristica molto utile di questi programmi sta nel fatto che essi accettano 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.

4.5.3 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 come input uno dei circuiti creati precedentemente e, applicando i metodi di sintesi crea un altro file contenente il circuito sintetizzato. In questo modo è successivamente possibile fare dei successivi confronti una volta sintetizzati tutti i circuiti.

4.6 Il calcolo dei costi del circuito

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

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

Per calcolare questo tipo di informazioni si devono effettuare delle operazioni di lettura sui file precedentemente creati analizzando 2 fattori:

- Gli 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.)
- Il dominio dei valori di input che devono inserire le parti.

4.6.1 blfmv vs pla

Per fare il confronto sopra citato si è preferito utilizzare file che avessero 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. Per questo motivo si è deciso di utilizzare il formato *blif* per rappresentare i circuiti binari. *Blif* è l'alternativa 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 sarà possibile una comparazione 1:1 tra binario e multivalore.

Per effettuare questa conversione vengono in aiuto i tool *ABC* e *MVSIS* poiché 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

4.6.2 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 circuito sia binario che mutli-valore, esso prende tutte le informazioni utili per poter effettuare il calcolo

```
costo = 0
for t in circuito['tabelle']:
  intersection = len(set(circuito['input']).intersection(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 MULTIVALORE;INPUT ALICEINPUT BOB\n')
  for blfmv in listdir('{}/blfmv'.format(working_dir)):
    if blfmv.endswith('.mv'):
      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_var_mv,
```

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

4.7 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.

4.7.1 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 path.exists('{}/blfmv'.format(working_dir)):
        rmtree('{}/blif'.format(working_dir))
        rmtree('{}/blfmv'.format(working_dir))

makedirs('{}/blif/synth'.format(working_dir))
makedirs('{}/blfmv/synth'.format(working_dir))
makedirs('{}/blfmv/synth/abc'.format(working_dir))
makedirs('{}/blfmv/synth/mvsis'.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* per avere un formato identico di rappresentazione tra i circuiti binari e quelli multivalore.

4.7.1.1 pla_to_blif

```
def pla_to_blif(working_dir):
  with open('./pla_to_blif.mvsis', 'w') as file:
    file.write("source abc_alias.abc\n")
    for ele in os.listdir('{}/pla'.format(working_dir)):
      file.write('read_pla {}/pla/{}\n'.format(working_dir,ele))
      file.write('write_blif {}/blif/{}.blif\n\n'.format(
        working_dir, ele.split('.')[0]))
```

Questa funzione restituisce un file leggibile come input da *mvsis* o *abc* in grado di cambiare automaticamente formato dei file da `PLA` a `BLIF`. In questo modo la conversione viene fatta automaticamnte da uno dei programmi scelti senza ricorrere ad ulteriore codice o incappare in errori di conversione. Il file ottenuto ha la seguente struttura.

```
source abc_alias.abc
read_pla ./prova/pla/alu2.pla
write_blif ./prova/blif/alu2.blif

read_pla ./prova/pla/br2.pla
write_blif ./prova/blif/br2.blif

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

In questo modo ogni circuito viene convertito semplicemente chiamando

```
call(['abc', '-F', './pla_to_blif.mvsis'])
```

4.7.1.2 bin_to_mv_mcd

```
def bin_to_mv_mcd(working_dir):
  for pla in listdir('{}/pla'.format(working_dir)):
    if pla.endswith('.pla'):
      inp, out, inp_array, out_array, truth_table = read_pla(
        '{}/pla/{}'.format(working_dir, pla))
      dv = euclide(int(inp), int(out))
      if int(inp.strip()) == int(out.strip()) or dv == int(inp):
        dv = dv/2
      if dv >= 8:
        dv = 2
        mv = pow(2, dv)
        print('{}\ninp: {}\nout:{}\nmulti-valore:{}\ndivide ogni:{}\n'.format(pla,
          inp.strip(), out.strip(), int(mv), int(dv)))
        truth_table = expand_dont_care(truth_table, int(dv))
        mv_table = create_mv_truth_table(truth_table, int(dv))
        crete_blif_mv(working_dir, mv_table, int(mv),
          basename(pla).split('.')[0])
```

In questa funzione vengono racchiuse le funzioni descritte precedentemente per la conversione del circuito da binario a multivalore e la creazione dei file utilizzabili dai motori di sintesi. Come detto nei capitoli precedenti utilizzo 2 tecniche di conversione in base al circuito, una statica e una dinamica, qui mostro solo la soluzione dinamica in quanto più complessa ma dalla struttura analoga alla proposta statica.

```
dv = euclide(int(inp), int(out))
  if int(inp.strip()) == int(out.strip())
    or dv == int(inp):
    dv = dv/2
  if dv >= 8:
    dv = 2

def euclide(a, b):
  while(b != 0):
    R = a % b
    a = b
    b = R
  return a
```

In questa porzione avvengono le fasi di individuazione del M.C.D. tramite la funzione *euclide* e viene effettuato il controllo in caso che il dominio trovato si troppo grande o che ci sia un solo input utilizzando questo dominio.

```
synth(working_dir)

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

L'ultimo passo consiste nella sintesi dei circuiti ottenuti e nella loro controparte binaria, la funzione `synth` fa in modo che vengano creati 2 file da dare in ingresso ad `MVSIS` o `ABC` che sintetizzino tutti i circuiti e salvino il risultato in dei file.

4.7.1.3 Synth

```
def synth(working_dir):
  with open('./synth_bool.abc', 'w') as file:
    file.write("source abc_alias.abc\n")
    for ele in os.listdir('{}/blif'.format(working_dir)):
      if ele.endswith('.blif'):
        file.write('read_blif {}/blif/{}\n'.format(working_dir,ele))
        file.write('cl\n')
        file.write('resyn2\n')
        file.write('cl\n')
        file.write('write_blif {}/blif/synth/{}.blif\n\n'.format(working_dir,ele.split
  with open('./synth_mv.abc', 'w') as file:
    file.write("source abc_alias.abc\n")
    for ele in os.listdir('{}/blfmv'.format(working_dir)):
```

```
      if ele.endswith('.mv'):
        file.write('read_blif_mv {}/blfmv/{}\n'.format(working_dir, ele))
        file.write('strash\n')
        file.write('compress2\n')
        file.write('cl\n')
        file.write('resyn2\n')
        file.write('cl\n')
        file.write('write_blif_mv {}/blfmv/synth/abc/{}\n\n'.format(working_dele))

with open('./synth_mv.mvsis', 'w') as file:
  for ele in os.listdir('{}/blfmv'.format(working_dir)):
    if ele.endswith('.mv'):
      file.write('read_blif_mv {}/blfmv/{}\n'.format(working_dir, ele))
      file.write('sweep\n')
      file.write('eliminate -l 1\n')
      file.write('simplify -m nocomp\n')
      file.write('eliminate -l 1\n')
      file.write('sweep\n')
      file.write('eliminate -l 5\n')
      file.write('simplify\n')
      file.write('sweep\n')
      file.write('eliminate -l 1\n')
      file.write('sweep\n')
      file.write('fullsimp -m nocomp\n')
      file.write('write_blif_mv {}/blfmv/synth/mvsis/{}\n\n'.format(working_dir, ele))
```

- **synth_bool**: crea il file per la sintesi utilizzata da **ABC** per i circuiti booleani;
- **synth_mv.abc**: crea il file per la sintesi utilizzata da **ABC** per i circuiti multivalore;
- **synth_mv**: crea il file per la sintesi utilizzata da **MVSIS** per i circuiti multivalore;

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

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

Infine i costi dei circuiti possono essere calcolati

```
calcolo_costi_synth(working_dir)
calcolo_costi_no_synth(working_dir)

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

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

Capitolo 5

Risultati sperimentali

Una volta che il processo di conversione dei circuiti e che tutto il sistema di automazione dei processi è terminato si è proceduto con l'analisi sperimentale per valutare se questo nuovo approccio sviluppato possa essere considerato una possibile alternativa alle proposte già presenti in letteratura.

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

5.1 MVSIS

Nelle prime fasi di sperimentazione si e' decido di utilizzare *MVSIS* come motore di sintesi sia per i circuiti binari che per quelli multivalore. Questo primo approccio nasce dal voler replicare le operazioni che si svolgono 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 andiamo a sintetizzare il circuito preso in ingresso 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 è stato possibile 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.

5.1.1 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.

5.1.2 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 vengono mostrati i costi dei circuiti senza sintesi. Osserviamo che nella maggior parte dei casi il dominio multivalore senza sintesi ha un costo minore, questo è figlio del minor numero di input che le parti dovranno inserire all'interno del protocollo durante la computazione.

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 trovare 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.

5.2 ABC

Notando che inizialmente il costo del circuito multivalore possiede dei costi più bassi per via del minor numero di input e ma la logica binaria presenta degli strumenti di sintesi più ottimizzati, 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.

5.3 ABC e dominio fisso

Come con *MVSIS* abbiamo inizialmente utilizzato un dominio fisso per svolgere 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
in5	446	7	7	2	1336	7	7

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
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.

5.4 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
in5	446	7	7	2	1336	7	7

NOME CIRC	COSTO BOOL	INPUT ALICE	INPUT BOB	DOMINIO MULTI	COSTO MULTIVALORE	INPUT ALICE	INPUT BOB
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 sono differenze stanziali, questo potrebbe essere dato da come *abc* legge i circuiti multivalore in input.

5.5 Conclusioni sui risultati

Come abbiamo potuto notare dai risultati ottenuti da questa sperimentazione l'utilizzo della logica multivalore potrebbe rivelarsi interessante per la riduzione dei costi dati dai valori di input richiesti dai circuiti. La limitazione principale che abbiamo potuto riscontrare sta nella carenza di strumenti di sintesi multivalore a disposizione, solamente **MVSIS** è in grado di gestire e interpretare questo tipo di logica ma resta un programma ora mai datato e non più mantenuto. I nuovi strumenti messi a disposizione, come **ABC** accettano ion ingresso circuiti multivalore ma nel loro motore di sintesi vengono convertiti in circuiti booleani, l'efficienza e l'efficacia delle nuove funzioni di sintesi di **ABC** rende l'utilizzo di questo software la prima scelta da utilizzare. Applicando un approccio ibrido sfruttando la riduzione degli input durante la conversione del dominio combinata all'alta efficacia degli strumenti di sintesi messi a disposizione di **ABC** risulta essere il migliore approccio testato durante questa sperimentazione.

Bibliografia

Beaver, D., Micali, S., and Rogaway, P. (1990a). *The Round Complexity of Secure Protocols*. STOC '90. Association for Computing Machinery, New York, NY, USA.

Beaver, D., Micali, S., and Rogaway, P. (1990b). The round complexity of secure protocols. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, page 503–513, New York, NY, USA. Association for Computing Machinery.

Bertacco and Damiani (1997). The disjunctive decomposition of logic functions. In *1997 Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 78–82.

Pan, P. (1997). Continuous retiming: algorithms and applications. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 116–121.

Silveira, L. M., White, J. K., Neto, H. C., and Vidigal, L. M. (1992). On exponential fitting for circuit simulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 11(5):566–574.