

UNIVERSITÀ DEGLI STUDI DI PADOVA
Dipartimento di Ingegneria dell'Informazione

Laurea in Ingegneria Elettronica



**UNIVERSITÀ
DEGLI STUDI
DI PADOVA**



**ELABORATO TRIENNALE
HIGH-LEVEL SYNTHESIS:
UN APPROCCIO INNOVATIVO
ALLA SINTESI DIGITALE**

Prof.: DANIELE VOGRIG

Laureanda: TULLIA FONTANA
1189588

ANNO ACCADEMICO 2020-2021

Indice

Elenco delle figure	iii
1 Introduzione	1
2 Il contesto	3
2.1 La sintesi ad alto livello nel modello di rappresentazione dell'hardware	3
2.2 I livelli di astrazione relativi all'HDL e ai linguaggi ad alto livello	6
2.3 Sintesi logica VS sintesi ad alto livello	7
2.4 Che cosa significa ottimizzare un progetto hardware?	9
2.5 La tecnologia	10
2.6 Le figure professionali	11
3 Storia: dagli anni '70 agli anni 2010	13
3.1 Ricerca VS Industria	14
3.1.1 Anni '70 - primi anni '80	14
3.1.2 Seconda metà anni '80 - primi anni '90	14
3.1.3 Seconda metà degli anni '90 - primi anni 2000	14
3.1.4 Primi anni 2000 - primi anni '10	15
4 Il flusso di progettazione della high-level synthesis	17
4.1 L'input del processo di sintesi	18
4.2 Compilazione delle specifiche	18
4.2.1 Per approfondire: le ottimizzazioni del compilatore	19
4.3 Allocation, scheduling e binding	19
4.4 La generazione dell'architettura RTL	21
4.5 L'output del processo di sintesi	22
4.6 Un esempio concreto del flusso di progettazione ad alto livello	22
5 I vantaggi dell'high-level synthesis	27
5.1 Esplorazione dello spazio di progettazione	27
5.2 Verifica	30
5.3 Riutilizzo dei progetti	33
5.4 Il ruolo chiave degli FPGA	34
6 Linguaggi	37
6.1 Qual è il linguaggio più adatto per l'input dell'high-level synthesis?	37
6.2 L'avvento del linguaggio C/C++	38
6.2.1 Estensioni e limitazioni	39
6.2.2 Chi può trarne vantaggio?	43
6.2.3 Quali sono i requisiti che un progettista deve avere per approcciarsi all'HLS? . .	44
6.3 L'impiego di altri linguaggi diversi da C/C++	46

7 Tool	49
7.1 Tool accademici VS tool commerciali	49
7.2 Quali sono i tool più diffusi?	51
7.2.1 Vivado HLS	51
7.2.2 Intel FPGA SDK FOR OpenCL	52
7.2.3 Intel HLS Compiler	52
7.2.4 Catapult HLS Platform	52
7.2.5 LegUp	53
7.2.6 Bambu	53
7.3 Quali fattori è necessario tenere in considerazione per la scelta del tool?	54
8 Sviluppi recenti	57
8.1 Applicazioni	57
8.2 State of art e possibili miglioramenti	58
9 Conclusioni	61
Bibliografia	65

Elenco delle figure

1.1	Il gap tra sviluppo tecnologico e produttività	2
2.1	Diagramma Y di Gajski e Kuhn	4
2.2	Flusso di progettazione dell'hardware (versione semplificata)	5
2.3	Sintesi RTL (a) e high-level synthesis (b) a confronto [10]	6
2.4	Livelli di astrazione relativi agli HDL e ai linguaggi ad alto livello [11]	7
2.5	Sintesi logica e sintesi ad alto livello [11]	8
2.6	Il processo di progettazione mediante sintesi ad alto livello (o sintesi comportamentale) e sintesi RTL a confronto [13]	9
2.7	Le figure professionali coinvolte nel flusso di progettazione dell'hardware [16]	11
3.1	Andamento del mercato dei tool di HLS dal 1994 al 2007 [20]	16
4.1	Step di progettazione nella sintesi ad alto livello [18]	17
4.2	Esempio di codice in C per un filtro FIR [21]	18
4.3	Esempio di architettura RTL [18]	21
4.4	Rappresentazione mediante DFG [22]	22
4.5	Rappresentazione mediante CDFG [22]	23
4.6	Modello (a) e implementazione (b) di una macchina a stati e del datapath [22]	23
4.7	Libreria RTL [22]	24
4.8	Possibili allocazioni per il DFG [22]	24
4.9	Tradeoff tra area e performance [22]	24
4.10	Esempio di schedule basato sull'allocazione E [22]	25
4.11	Storage binding: lifetime delle variabili (a), grafo di compatibilità (b), soluzioni possibili (c) [22]	26
5.1	Spazio di progettazione register-transfer level per delle particolari specifiche [8]	27
5.2	Classificazione delle tecniche di DSE [23]	29
5.3	Flussi di DSE: (a) synthesis based, (b) supervised learning based, (c) graph analysis based [23]	29
5.4	Framework di simulazione e verifica HLS [27]	32
5.5	Generazione automatica di un testbench RTL in AutoPilot, antenato dell'attuale Vivado HLS [27]	32
6.1	Esempio di codice con pragma per il loop unrolling e per il bilanciamento delle espressioni per implementare il parallelismo [39]	40
6.2	Flusso di progettazione di Vivado HLS [38]	41
6.3	Architettura del linguaggio SystemC [41]	42
6.4	Area e performance dei progetti RTL e HLS [24]	45
6.5	Produttività HLS e RTL [24]	45
6.6	Tempo massimo, minimo e medio per ogni fase del processo di sintesi RTL e con tool di HLS [24]	46

7.1	Tool commerciali attualmente disponibili [4]	50
7.2	Tool accademici attualmente disponibili [22]	51
7.3	Utilizzo dei tool di sintesi emerso in una ricerca del 2019 [24]	54

Glossario

ASIC Application Specific Integrated Circuit. [1](#)

CDFG Control and Data Flow Graph. [18](#)

CPU Central Processing Unit. [1](#)

DFG Data Flow Graph. [18](#)

DSE Design Space Exploration. [28](#)

DSP Digital Signal Processor. [1](#)

EDA Electronic Design Automation. [5](#)

ESL Electronic System-Level. [16](#)

FPGA Field Programmable Gate Array. [1](#)

GPU Graphics Processing Unit. [1](#)

HDL Hardware Description Language. [6](#)

HPC High Performance Computing. [10](#)

IEEE Institute of Electrical and Electronics Engineers. [43](#)

IR Internal Representation. [18](#)

NI National Instrument. [47](#)

RTL Register-Transfer Level. [2](#)

SLM System-Level Modeling. [7](#)

SoC System-on-a-chip. [1](#)

SR State Register. [21](#)

TSM Transaction-Level Modeling. [11](#)

VLSI Very Large Scale Integration. [1](#)

Capitolo 1

Introduzione

Introduzione generale del da' contesto: Necessità RTL per progettazione + Necessità Behav. per velocizzare

Come teorizzato dalla legge di Moore [1] e dallo scaling di Dennard [2], gli ultimi quattro decenni sono stati segnati dallo sviluppo inesorabile delle performance di calcolo nei sistemi digitali [3]. L'avanzamento tecnologico dell'industria del silicio ha permesso di ridurre la dimensione dei transistor, consentendo di integrare in uno stesso chip un numero sempre maggiore di dispositivi a semiconduttore. Più precisamente, dagli anni '60 ad oggi, il numero di transistor per SoC è cresciuto esponenzialmente grazie a un processo di miniaturizzazione e di VLSI (Very Large Scale Integration) che attualmente consente di produrre SoC con decine di milioni di transistor, proprio come era stato previsto da Moore. In sintesi, la miniaturizzazione dei transistor e la loro integrazione in SoC sempre più complessi si traduce in termini di performance in un aumento della velocità di calcolo, quindi della frequenza di clock dei processori; inoltre, allo scalare delle dimensioni, si assiste anche allo scaling di corrente e potenziale elettrico dei transistor, con il risultato che la densità di potenza dissipata rimane costante, in accordo con lo scaling di Dennard.

La legge di Moore e lo scaling di Dennard, di fatto osservazioni empiriche e non vere e proprie leggi, hanno determinato un'era di grande sviluppo tecnologico.

Tuttavia, a partire dagli anni 2000, si iniziò ad assistere al rallentamento di questo processo di miniaturizzazione dei transistor, con conseguente stabilizzazione della frequenza di clock dei processori [4]: oggi le dimensioni dei transistor sono dell'ordine dei 10 nanometri [3] e un'ulteriore miniaturizzazione indurrebbe degli effetti parassiti di natura quantistica che provocano consumi di potenza elevati, contravvenendo così allo scaling di Dennard. Per questo motivo, nonostante sia tutt'ora possibile miniaturizzare i transistor - in altri termini l'industria tecnologica è ancora in grado di realizzare transistor di dimensioni sempre più ridotte - sono sopraggiunti dei problemi di natura fisica: la dissipazione termica - che non può essere risolta nemmeno con sistemi di raffreddamento, che sarebbero causa di un significativo aumento dei costi dei chip - ha decretato non solo la fine dello scaling di Dennard, ma anche un rallentamento della legge di Moore.

Tali limiti fisici della tecnologia in silicio hanno dato la propulsione allo sviluppo di modi alternativi per migliorare le performance e l'efficienza energetica dei processori [5].

Una strada promettente per ovviare al problema è quello del calcolo eterogeneo: un gruppo di nodi di elaborazione (processing nodes) esegue un carico di lavoro in parallelo. Dati diversi tipi di nodi, come CPU multi-core, processori real-time, DSP, GPU e acceleratori su piattaforme Field Programmable Gate Array (FPGA) o Application Specific Integrated Circuit (ASIC), il carico di lavoro può essere partitionato in modo che ogni parte sia eseguita su un processore che sia adatto alle operazioni che deve effettuare e che soddisfi gli obiettivi di ottimizzazione delle performance. Tale metodo, quindi, mediante l'integrazione di processori software e hardware specifico creato ad hoc, consente di accelerare i processi con ridotto consumo energetico [4].

Il principale ostacolo nell'impiego di questo nuovo approccio basato sull'eterogeneità risiede nelle difficoltà che emergono nella fase di progettazione: la tecnologia si è sviluppata molto più velocemente rispetto ai metodi di sintesi che consentono di progettarla; di conseguenza la tecnologia prodotta è meno performante rispetto a quella che lo sviluppo tecnologico in sè potrebbe consentire.

La Figura 1.1, aggiornata al 2009, illustra proprio questo gap tra sviluppo tecnologico e produttività, dovuto appunto alle difficoltà che devono affrontare i progettisti nello sviluppo dell'hardware, non essendo supportati da adeguati strumenti e metodi di sintesi.

La metodologia di progettazione consolidatasi nei decenni passati prevede infatti che il progettista hardware scriva manualmente le specifiche del sistema digitale al livello RT (register-transfer).

Nel 2004, in uno studio [6] veniva riportato che un progetto hardware costituito da 1 milione di porte logiche richiedeva tipicamente 300.000 linee di codice Register-Transfer Level (RTL). Si tratta di un numero esorbitante, molto difficile da gestire anche per un progettista esperto, il cui lavoro può diventare molto frustrante e faticoso.

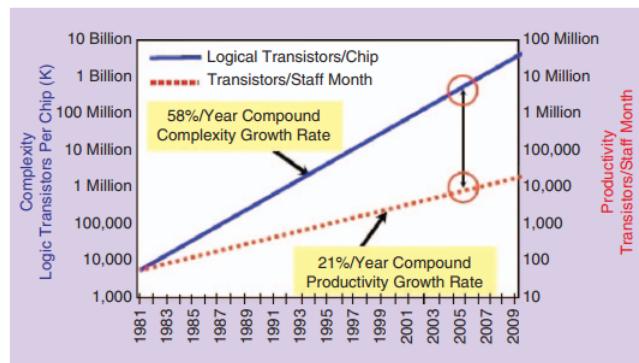


Figura 1.1: Il gap tra sviluppo tecnologico e produttività

Esiste un modo per rendere il progetto più semplice e leggibile e che consenta, al tempo stesso, di sfruttare al meglio il grande potenziale di calcolo offerto dalla tecnologia di oggi?

Spostare il flusso di progettazione a livelli di astrazione più elevati sembra essere la chiave per bypassare questa problematicità: l'high-level synthesis, detta anche sintesi comportamentale o algoritmica, consente di descrivere l'hardware che si intende progettare in termini di comportamento, nascondendo al progettista parte dei dettagli implementativi e lasciando al tool di sintesi il compito di sintetizzare l'architettura che meglio si adatta alle specifiche ad alto livello del progetto.

In questo modo, non solo la densità del codice può essere ridotta di 7X-10X, riuscendo a descrivere lo stesso progetto con 30k-40k righe di descrizione comportamentale e risolvendo così il problema della complessità della progettazione, ma è anche possibile andare incontro a numerosi altri vantaggi, che, complessivamente, conducono a una notevole riduzione del time-to-market e a un aumento della produttività, questioni molto care all'industria, soggetta alle pressioni e alle dinamiche di mercato.

Questo elaborato si propone di comprendere l'importanza della high-level synthesis come turning point nel campo della progettazione digitale.

Dopo una breve ricostruzione storica dell'evoluzione dei metodi di progettazione digitale dagli anni '70 ad oggi, verranno analizzate le fasi dell'high-level synthesis; seguirà poi un excursus sui numerosi vantaggi che tale sintesi porta con sé. Verranno inoltre presentati i linguaggi maggiormente diffusi impiegati nel processo di sintesi e i tool commerciali e accademici di punta degli ultimi anni. L'elaborato si concluderà con un rapido excursus che illustrerà le direzioni future nel campo della ricerca.

Overview

Capitolo 2

Il contesto

2.1 La sintesi ad alto livello nel modello di rappresentazione dell'hardware

L'high-level synthesis è uno degli step della progettazione di un circuito digitale. La sua collocazione nel flusso di progettazione dell'hardware può essere individuata mediante il diagramma Y di Gajski e Kuhn (Figura 2.1).

Tale grafico è stato proposto nel 1983 per illustrare le diverse prospettive nella progettazione hardware VLSI (very large scale integration) ed è stato successivamente perfezionato da Robert Walker e Donald Thomas nel 1985 [7].

Il diagramma Y, un modello di rappresentazione dell'hardware, consta di tre domini, il dominio comportamentale, il dominio strutturale e il dominio fisico. Ciascun dominio fornisce una diversa prospettiva sull'hardware che si intende progettare, nascondendo i dettagli agli altri domini, o per meglio dire astraeendoli dai dettagli; pertanto ogni dominio dispone solo del suo proprio punto di vista.

Il primo dominio di rappresentazione dell'hardware è il dominio comportamentale, che descrive le funzionalità di base del progetto e contiene componenti statici e dinamici. I componenti statici descrivono le operazioni, mentre la parte dinamica descrive la sequenza e il timing delle suddette operazioni.

C'è poi il dominio strutturale, che realizza un'implementazione astratta del progetto, rappresentandolo come un'interconnessione strutturale di un insieme di blocchi astratti. Esso consente di collegare il dominio comportamentale a quello il fisico.

Infine, quest'ultimo costituisce l'implementazione fisica del progetto, o meglio, la realizzazione in componenti fisici reali dei componenti strutturali astratti del dominio strutturale.

I domini di descrizione dell'hardware sono rappresentati tramite degli assi che convergono in uno stesso vertice, formando una Y. Lungo gli assi, sono rappresentati i livelli di astrazione: partendo dal vertice, man mano che si procede verso l'esterno, il livello di astrazione aumenta e i dettagli di progettazione vengono nascosti [7]. Analizziamo la gerarchia dei livelli di astrazione con un approccio top-down, con preciso riferimento al dominio strutturale e a quello comportamentale. Essa fu teorizzata da Bell e Newell nel 1971, che la applicarono specificamente ai computer, ma può essere utilizzata per descrivere qualunque dispositivo hardware [8].

Alla sommità c'è il livello in cui il progetto digitale che si vuole implementare è descritto in termini di CPU e memorie e prende il nome di system-level; qui il comportamento del sistema consiste in un insieme di specifiche sulle performance dell'hardware.

Il livello successivo è quello algoritmico, dove il progetto si configura come dei moduli hardware, più precisamente come un datapath e una macchina a stati finiti che operano contemporaneamente, anche se le operazioni interne possono essere sia sequenziali, sia concorrenti; in termini di comportamento, il progetto è descritto a un livello sintatticamente simile ai linguaggi di programmazione: espressioni algoritmiche complesse, strutture dati, procedure e variabili sono impiegate per la descrizione del comportamento al livello algoritmico. Alcune operazioni tipiche di questo livello di astrazione nel contesto

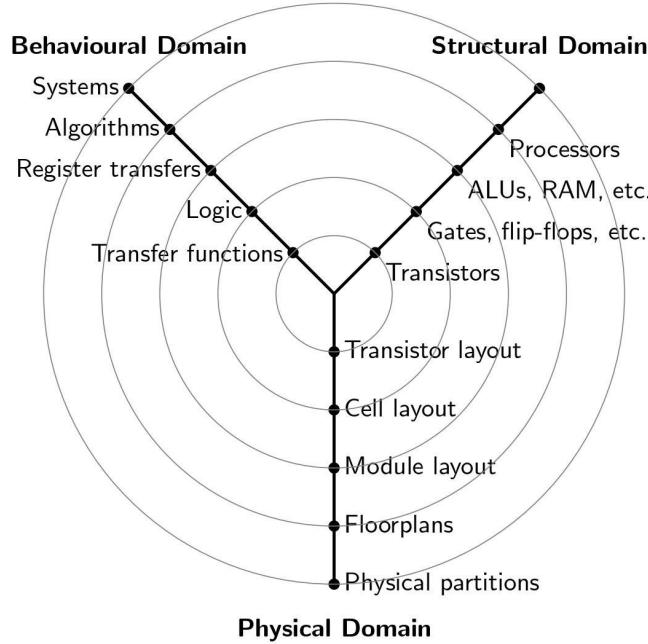


Figura 2.1: Diagramma Y di Gajski e Kuhn

del dominio comportamentale sono la decodifica e l'esecuzione delle istruzioni. Limitate informazioni generali relative al timing sono prerogativa di questo livello: è necessario specificare la temporizzazione per consentire l'interfacciamento e la coordinazione tra i vari moduli hardware.

Sotto il livello algoritmico c'è il livello register-transfer, in cui il sistema è visto come un insieme di storage element, come i registri, e blocchi funzionali interconnessi, tra cui ALU, adder, comparatori e multiplexer; a questo livello avvengono operazioni logiche e aritmetiche e trasferimenti tra registri. Dal momento che questo livello ha un livello di dettaglio superiore rispetto al precedente, deve fornire indicazioni più precise sul timing, specificando il comportamento cycle-by-cycle.

Segue poi il livello logico, in cui il sistema è descritto come una rete di porte, flip-flop e registri che operano in parallelo; mentre il comportamento è specificato da equazioni logiche.

Infine, troviamo il livello del circuito, che presenta una visione del sistema in termini di transistor di cui è composto; per specificarne il comportamento, si fa riferimento a potenziale elettrico e a correnti, espressi tramite equazioni differenziali [8].

Con preciso riferimento al diagramma Y, la progettazione dell'hardware consiste quindi nel processo di traduzione di una descrizione nel dominio comportamentale system-level a una descrizione nel dominio fisico circuit-level. Tale processo include una serie di operazioni di traduzione che consentono la costruzione di collegamenti tra domini - dal dominio comportamentale, a quello strutturale, a quello fisico - aggiungendo un numero sempre maggiore di dettagli per produrre una descrizione di basso livello.

Ciascuna delle operazioni di traduzione che avvengono nel flusso di progettazione prende il nome di sintesi. Quindi, in generale, la sintesi consiste in un processo di traduzione della descrizione di un progetto hardware da un livello di astrazione più alto a uno più basso oppure da un dominio a un altro (più precisamente dal comportamentale allo strutturale oppure dallo strutturale al fisico) [7]. Dal momento che la direzione del processo di sintesi è rivolta sempre più in profondità nella gerarchia dei domini e dei livelli di astrazione e sapendo che a un livello di astrazione più basso corrisponde un maggiore grado di dettaglio, la sintesi deve essere in grado di aggiungere dei dettagli implementativi alla descrizione di un progetto. Tale operazione non presenta una soluzione univoca: potrebbero esserci infatti diverse implementazioni strutturali, cioè diverse architetture, di un particolare comportamento, come anche diverse implementazioni fisiche di una particolare struttura. Per questo motivo si dice che la sintesi è un processo few-to-many [9].

Nella pratica, la progettazione dell'hardware è una procedura molto articolata, la cui descrizione esula

2.1. LA SINTESI AD ALTO LIVELLO NEL MODELLO DI RAPPRESENTAZIONE DELL'HARDWARE

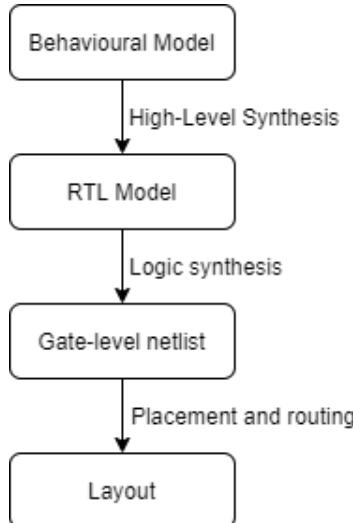


Figura 2.2: Flusso di progettazione dell'hardware (versione semplificata)

dallo scopo di questo breve elaborato; per questo motivo si è deciso di riassumerla nel modo seguente:

- progettazione elettronica a livello di sistema;
- progettazione RTL;
- progettazione fisica.

Per passare da una fase all'altra del flusso di progettazione dell'hardware, è necessario compiere un'operazione di sintesi che può essere effettuata da un progettista umano oppure automaticamente tramite dei software **EDA** (electronic design automation).

Gli EDA sollevano il progettista da un compito molto oneroso, ma dal momento che la sintesi non è una traduzione letterale questi software sono molto difficili da sviluppare: le scelte che dovrebbe compiere il progettista vengono delegate a degli algoritmi che devono essere in grado di vagliare le diverse possibilità, riuscendo a effettuare delle scelte almeno paragonabili a quelle che potrebbe eseguire a mano l'utente umano.

Nel corso dei decenni le varie operazioni di sintesi del flusso di progettazione sono state man mano automatizzate grazie allo sviluppo di numerosi tool, riuscendo a risalire la gerarchia dei livelli di astrazione, sollevando il progettista dal dover eseguire manualmente tali operazioni.

Da almeno tre decenni il settore della progettazione digitale vede la presenza di tool ben sviluppati che coprono tutte le fasi della sintesi a partire dalle specifiche RTL, passando a livelli di astrazione sempre più bassi, fino ad arrivare al place and route, che porta alla progettazione dei circuiti veri e propri, come si può vedere seguendo lo schema di Figura 2.2.

Fino a qualche decennio fa non era presente alcun tool che consentisse la sintesi a livelli superiori all'RTL: il passaggio dal livello di sistema - ovvero il livello più alto della gerarchia, a cui la progettazione ha inizio - al livello RT doveva quindi essere effettuato manualmente dai progettisti.

Tra il system-level e il register-transfer level c'è però un altro livello di astrazione, il livello algoritmico, quindi di fatto il gap di astrazione con cui i progettisti dovevano misurarsi era molto consistente.

Ed è qui che entra in gioco l'high-level synthesis. Essa si propone di accorciare il gap tra la fase preliminare della progettazione system-level e la generazione dell'RTL: grazie all'high-level synthesis il codice register-transfer level può essere generato automaticamente a partire dalla descrizione algorithmic-level del progetto, invece che scritto a mano dai progettisti hardware.

In questo modo, date le specifiche system-level del progetto, la fase manuale della progettazione si riduce all'elaborazione di specifiche algorithmic-level, come mostrato dalla Figura 2.3, dove vengono messe a confronto le fasi del flusso di progettazione manuale RTL e dell'HLS.

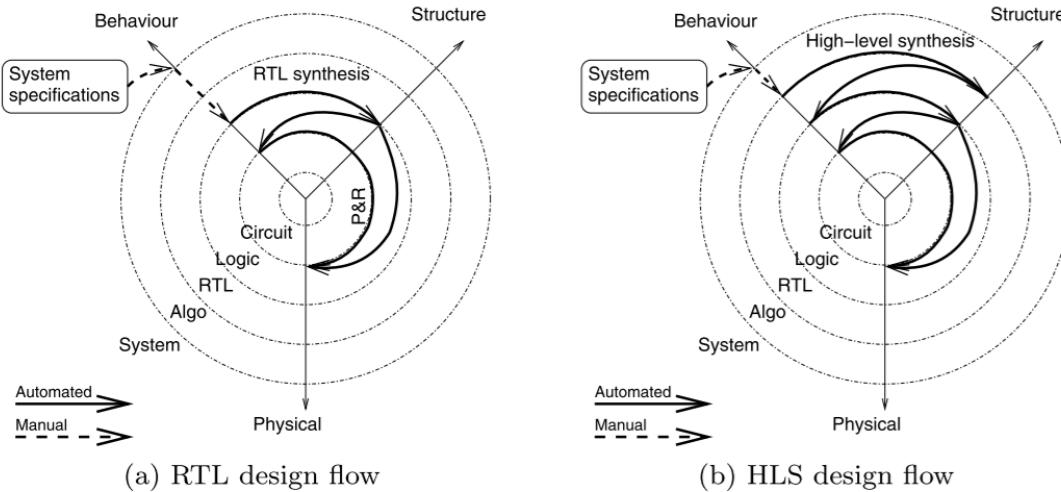


Figura 2.3: Sintesi RTL (a) e high-level synthesis (b) a confronto [10]

Sintetizzare automaticamente specifiche RTL a partire da una descrizione algoritmica tuttavia non è un'operazione banale: le differenze tra i due livelli di astrazione sono notevoli.

Come è stato sottolineato, la differenza tra i due livelli risiede nel livello di dettaglio con cui il progetto viene descritto: al livello register-transfer, le variabili nella descrizione vengono memorizzate in registri ben precisi del progetto e le dichiarazioni di assegnazione trovano corrispondenza diretta con le operazioni di trasferimento tra registri; inoltre il timing delle operazioni è completamente specificato. Al livello algoritmico, invece, viene definito solamente il comportamento del sistema in termini di input/output, con specifiche sul timing parziali o del tutto assenti.

2.2 I livelli di astrazione relativi all'HDL e ai linguaggi ad alto livello

Analizziamo più da vicino il processo di progettazione dell'hardware, facendo riferimento a un modello riconducibile all'Y graph, ma semplificato, in cui il passaggio da un dominio a quello adiacente è considerato alla stregua di un passaggio da un livello di astrazione a un altro.

Tale modello si compone di quattro livelli di astrazione, che sono, in ordine crescente: livello strutturale, livello RTL e livello comportamentale, caratterizzati dal fatto che possono essere descritti mediante **HDL** (Hardware Description Language) e infine alto livello, che si distingue dai precedenti in quanto può essere descritto appunto mediante linguaggi ad alto livello, come illustrato nella Figura 2.2. Gli **HDL**, ovvero i linguaggi di descrizione dell'hardware, sono dei linguaggi che consentono la definizione di architetture e di funzionalità tipiche dell'hardware, come la sincronizzazione e il parallelismo tra processi. I linguaggi HDL permettono di implementare un progetto hardware a tre diversi livelli di astrazione.

Il primo livello è quello strutturale, che prevede l'instanziazione, la configurazione e la connessione in modo esplicito di ciascun elemento hardware del progetto.

Più frequentemente, i progettisti lavorano al livello register-transfer level (RTL), un livello di astrazione che nasconde i dettagli a livello di tecnologia, ma prevede comunque uno stile di descrizione in cui i registri e le operazioni che avvengono tra essi devono essere specificate.

Infine le descrizioni comportamentali sono a un livello di astrazione più elevato rispetto all'RTL e rappresentano una descrizione algoritmica del circuito (indicano cioè come il circuito si comporta), invece che esprimere le singole operazioni tra registri, come avveniva ai livelli di astrazione inferiori. L'uso di HDL comportamentali, pertanto, affida al tool di sintesi la deduzione dell'hardware dalla descrizione; in questo modo il progettista concede al tool di sintesi un certo controllo sull'implementazione finale, ma con il vantaggio che i progetti possono essere creati più rapidamente.

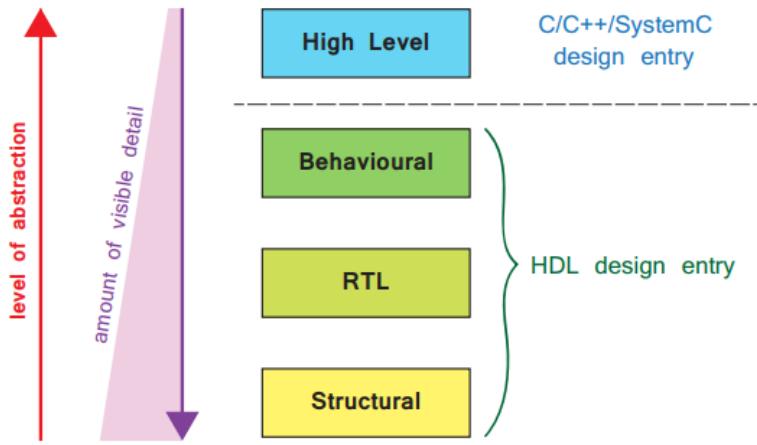


Figura 2.4: Livelli di astrazione relativi agli HDL e ai linguaggi ad alto livello [11]

Infine, al più alto livello di astrazione del modello proposto, vi è il metodo di progettazione ad alto livello. Tale metodo non prevede l'utilizzo di HDL, ma utilizza un insieme di linguaggi adatti ad esprimere i progetti a un livello di astrazione algoritmico, ovvero C, C++ e SystemC, con alcune restrizioni sull'uso di certi costrutti.

Questi linguaggi, infatti, sono spesso impiegati per sviluppare modelli a livello di sistema (SLM - System-Level Modeling) come primo passo di alto livello per la progettazione di un sistema [11]. Una volta che il progetto da implementare è stato descritto in termini di sistema, il progettista hardware ha il compito di elaborare tale descrizione ad alto livello per renderla sintetizzabile, cioè adatta a essere accettata in input da un tool di high-level synthesis in modo tale che il codice HDL possa essere sintetizzato. L'high-level synthesis, quindi, in virtù dei linguaggi da essa impiegati, si propone come ponte tra il mondo software e quello hardware.

Per riassumere, l'high-level synthesis, oggetto di questa breve trattazione, consiste nella conversione di codice scritto in C, C++ o SystemC di alto livello in una descrizione in HDL.

La sintesi logica, è il processo di analisi e di interpretazione di codice scritto in hardware description language (HDL) che consente di formare una netlist gate-level.

La sintesi di alto livello e la sintesi logica sono entrambe applicate (una dopo l'altra) nel processo di progettazione hardware, come mostrato nella Figura 2.5.

2.3 Sintesi logica VS sintesi ad alto livello

La sintesi logica è la modalità tradizionale per descrivere sistemi a larga scala di integrazione (VLSI) e i blocchi costituenti di proprietà intellettuale (IP) consolidatasi negli ultimi decenni.

Questa metodologia richiede una conoscenza molto approfondita dei circuiti digitali: per attuarla i progettisti devono specificare le funzionalità dell'hardware a un basso livello di astrazione, dove il comportamento cycle-by-cycle è completamente specificato. Questo procedimento richiede al progettista uno sforzo non indifferente, sempre maggiore a causa della complessità crescente dei sistemi digitali: mentre i tool RTL si sono sviluppati solo linearmente, la complessità dei sistemi VLSI è cresciuta esponenzialmente, rendendo così la realizzazione del progetto e il processo di verifica un collo di bottiglia per la produttività [12].

Quale può essere la soluzione a una tale impasse per il settore della sintesi digitale?

Il passaggio dal register-transfer level - proprio dei linguaggi di descrizione dell'hardware (HDL) - al livello immediatamente superiore, l'alto livello o il livello comportamentale, sembra essere la chiave delle problematiche in cui si è imbattuto il settore della sintesi digitale. Lavorare a un livello di astrazione più alto per progettare sistemi digitali, mediante una sintesi detta appunto high-level, consente ai progettisti

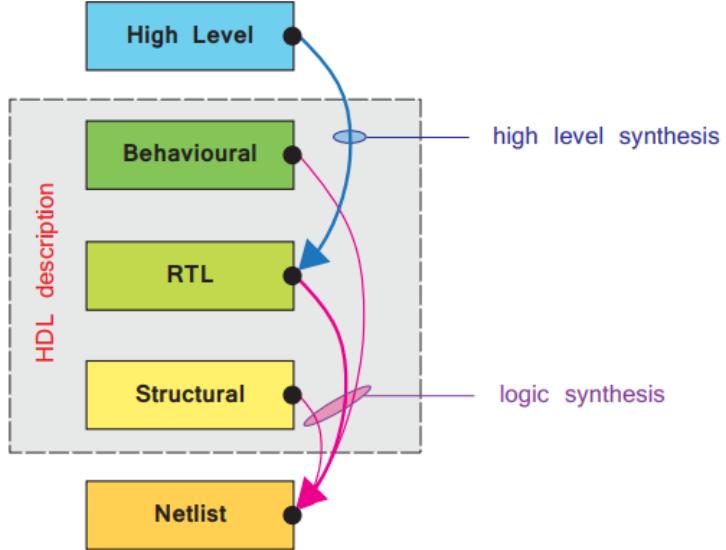


Figura 2.5: Sintesi logica e sintesi ad alto livello [11]

di specificare, modellizzare, verificare, sintetizzare, simulare ed effettuare il debug dei progetti in maniera più efficiente e in meno tempo.

Ma in che cosa consiste in termini più concreti la differenza tra questi due livelli di astrazione? In che modo fare uso di tool di high-level synthesis risulta essere più vantaggioso rispetto ai metodi di progettazione tradizionali?

La Figura 2.5 mostra il flusso di progettazione tramite high-level synthesis e sintesi logica applicate una dopo l'altra nel processo di progettazione dell'hardware. La differenza tra i due livelli di astrazione consiste nel fatto che ad alto livello non è ancora stata compiuta alcuna scelta in termini di architettura. Nella progettazione RTL, la ricerca dell'architettura e dei constraints ottimali per ottenere determinate prestazioni è una fase critica del flusso di progettazione e richiede una stretta collaborazione tra system engineer e progettista dell'hardware. L'implementazione dell'architettura è poi a carico del progettista, il quale non solo possiede una conoscenza approfondita degli aspetti architetturali, ma ha anche coscienza delle prestazioni di massima ottenibili con le varie architetture. Il progettista dell'hardware potrà quindi effettuare una scelta architetturale in linea di massima adatta ad ottenere le prestazioni richieste dal progetto.

Tuttavia, il progettista appurerà se la scelta fatta preliminarmente soddisfa o meno i requisiti del progetto solo dopo aver effettuato la verifica funzionale, al termine di una lunga fase di progettazione. Pertanto, fino a quando il progetto non sarà completato, il progettista è potenzialmente esposto alla possibilità di aver scelto un'architettura non del tutto ottimale. Se ciò dovesse verificarsi, potrebbe essere necessario rivalutare le specifiche system-level, per poi intraprendere nuove scelte architetturali in un processo iterativo, molto lungo e dispendioso o persino dannoso per l'azienda, vincolata da scadenze stringenti dettate dalle brevi finestre di mercato.

La Figura 2.6 illustra il flusso di progettazione con scrittura manuale del codice RTL, che potrebbe essere reiterata in caso di scelte architetturali non del tutto adeguate; al contrario, nel flusso di high-level synthesis la scelta dell'architettura è un processo "push-button". Grazie ai tool di high-level synthesis, infatti le decisioni architetturali sono compiute automaticamente dal tool di sintesi attraverso degli algoritmi, che idealmente dovrebbero essere in grado di valutare sistematicamente lo spazio di progettazione, esplorando così diverse architetture prima di selezionare quella che meglio si adatta alle specifiche del progetto. In questo modo, tutte le operazioni di norma eseguite manualmente dai progettisti hardware, possono essere effettuate automaticamente dai tool di high-level synthesis, non soltanto in maniera più rapida, ma anche in maniera ottimizzata.

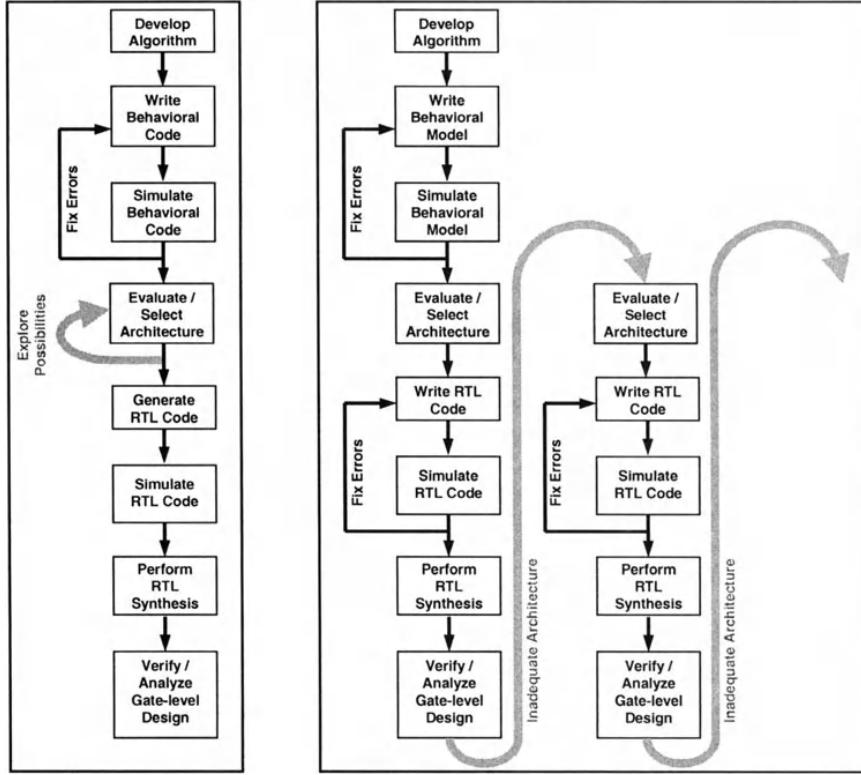


Figura 2.6: Il processo di progettazione mediante sintesi ad alto livello (o sintesi comportamentale) e sintesi RTL a confronto [13]

2.4 Che cosa significa ottimizzare un progetto hardware?

In che cosa consiste l'ottimizzazione che la sintesi ad alto livello permette di attuare in un progetto hardware?

I parametri fondamentali su cui si basa la progettazione dell'hardware sono:

- l'area, ovvero la quantità di hardware necessaria alla realizzazione delle funzionalità che il progetto deve soddisfare, che si traduce in un determinato costo delle risorse;
- la velocità, più precisamente il throughput, ovvero la velocità a cui il circuito può elaborare i dati;
- la latency, ovvero quanti cicli di clock sono necessari per produrre un output;
- la frequenza di clock, che può essere modificata in combinazione con altre scelte architetturali, come la gerarchia (per esempio è possibile adattare il periodo di clock alla velocità di trasferimento dei dati specifico in ciascun blocco del progetto);
- il consumo di potenza, cioè quanta energia consuma il sistema quando è operativo;
- requisiti di I/O, cioè quali sono le interfacce di cui il progetto è fornito;
- la memory bandwidth, ovvero il rate a cui i dati possono essere letti o archiviati in memoria mediante l'uso di un processore;
- la mappatura di interfacce, variabili locali o array nella memoria o la suddivisione in dei registri.

L'area, il throughput e la latenza costituiscono vincoli e obiettivi ben noti. Tuttavia, da dieci anni a questa parte, hanno iniziato a essere prese in considerazione caratteristiche, strettamente legate tra loro,

come il periodo di clock, il consumo energetico, la memory bandwidth e la mappatura nella memoria, rendendo così il problema della sintesi sempre più difficile da risolvere [14]. Area e throughput restano comunque i fattori più importanti: trovare un tradeoff tra area (costo delle risorse) e throughput è l'obiettivo principale della sintesi ad alto livello. Più precisamente, la HLS aiuta il progettista a realizzarne rapidamente le ottimizzazioni grazie all'esplorazione dello spazio di progettazione [11].

2.5 La tecnologia

Che genere di tecnologie consente di progettare l'high-level synthesis?

Per sviluppare hardware ottimizzato, in virtù della sua natura architecture-independent, l'high-level synthesis adotta un metodo che prevede l'integrazione di diversi componenti o nodi - che vanno dai processori software ad hardware application-specific - all'interno dello stesso sistema digitale, che deve essere partizionato in modo tale che ciascun nodo di elaborazione possa eseguire in parallelo task specifici per cui è stato appositamente sviluppato. Tale metodo prende il nome di heterogenous computing, cioè elaborazione eterogenea.

I nodi di elaborazione del calcolo eterogeneo includono CPU multi-core e di piattaforme di calcolo ibride con acceleratori, come GPU (Graphic Processing Unit), FPGA (Field Programmable Gate Array), ASIC (Application Specific Integrated Circuit) o DSP (Digital Signal Processor).

Per i domini di applicazione caratterizzati da esteso parallelismo la tecnologia più efficace risulta essere quella degli acceleratori, come FPGA o GPU. Gli FPGA sono delle piattaforme programmabili che mettono a disposizione numerose funzionalità logiche; le GPU sono delle unità specializzate in processi di elaborazione grafica.

Nei sistemi High Performance Computing (HPC), un FPGA può fungere da coprocessore configurabile per una CPU, dove l'FPGA esegue delle porzioni di progetto ad alta densità di calcolo. Gli FPGA offrono inoltre efficienza energetica e throughput elevati per applicazioni con ampio parallelismo e per la gestione di semplici e regolari tipi di dato, operazioni aritmetiche o strutture di controllo. Per applicazioni streaming, floating-point e applicazioni che richiedono un'elevata bandwidth della memoria, le GPU sono più indicate; sono infatti utilizzate in modo intensivo per numerose applicazioni scientifiche [15].

Per porzioni di applicazioni intrinsecamente seriali o con un ampio flusso di controllo invece, sono più adatte le CPU o ancora una volta gli FPGA.

Gli FPGA funzionano a frequenze dell'ordine dei MHz, mentre le CPU presentano frequenze che si aggirano attorno a qualche GHz. Tuttavia gli FPGA presentano prestazioni migliori rispetto alle CPU in quanto implementano ogni compito specifico su un circuito dedicato, sfruttando il più possibile il parallelismo, e offrono un'ampia bandwidth della memoria.

Gli FPGA sono la tecnologia target della sintesi platform-based, ovvero una metodologia di sintesi che prevede l'integrazione di diversi IP all'interno dello stesso sistema digitale, offrendo qualità del prodotto e al contempo permettendo di implementare il progetto rapidamente.

Un'altra tecnologia che l'high-level synthesis consente di implementare sono gli ASIC, circuiti integrati progettati ad hoc per svolgere una determinata funzione, similmente agli FPGA. Tuttavia sono molto più costosi rispetto a questi ultimi e non sono riconfigurabili. Per questo motivo sono tipicamente destinati a progetti che coprono ampi volumi di mercato.

Infine i DSP (Digital Signal Processor) sono dei processori sviluppati per la gestione di applicazioni real-time che richiedono l'elaborazione di dati in streaming.

Di fronte alle innumerevoli soluzioni offerte dalle diverse tecnologie presenti sul mercato, l'high-level synthesis si configura come lo strumento idoneo atto a gestire la progettazione di sistemi complessi, che integrano diversi componenti: la scelta della piattaforma di calcolo che più si adatta a una data applicazione, sulla base di considerazioni relative alle prestazioni e all'efficienza energetica, può essere eseguita automaticamente grazie all'HLS.

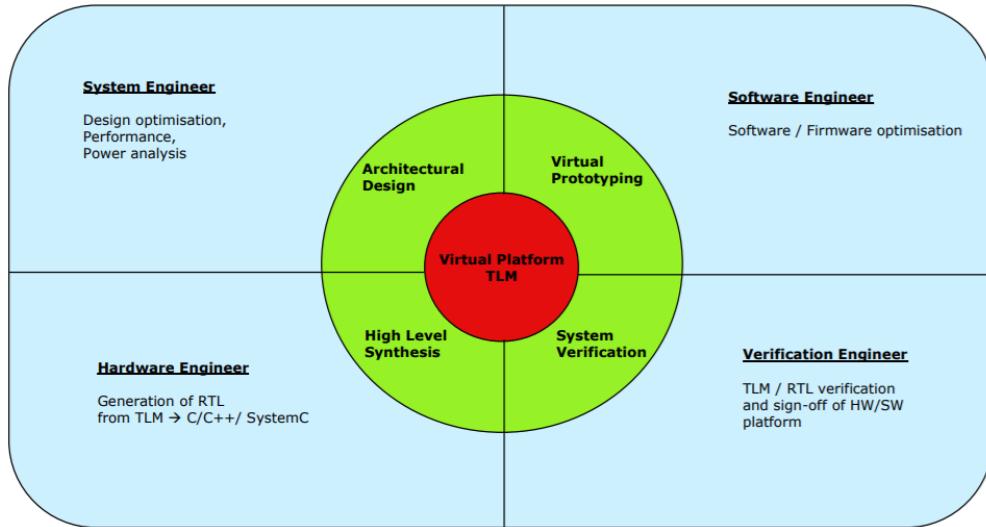


Figura 2.7: Le figure professionali coinvolte nel flusso di progettazione dell'hardware [16]

2.6 Le figure professionali

Quali sono le figure professionali che si occupano della sintesi di hardware? In che modo la high-level synthesis riesce ad agevolare il loro lavoro?

La Figura 2.7 [16] mostra la divisione dei compiti nel Transaction-Level Modeling (**TSM**), un approccio ad alto livello per lo sviluppo di sistemi digitali: software, system, hardware e verification engineers collaborano alla progettazione dell'hardware.

I software engineer si occupano della scrittura del codice a livello di sistema. Il loro obiettivo è che i tool di HLS producano automaticamente hardware relativamente veloce a partire dalla descrizione dell'algoritmo, integrando rapidamente il prodotto hardware in un sistema software esistente. La HLS agevola moltissimo questo processo, consentendo l'implementazione - più precisamente la prototipazione - su piattaforma FPGA.

I system engineers si occupano dell'ottimizzazione del progetto, effettuando delle analisi a priori relative alle performance e al consumo di potenza con l'obiettivo di definire le caratteristiche system-level che il progetto dovrà soddisfare.

Il compito dell'hardware engineer è quello di generare il codice RTL, a partire dagli algoritmi sviluppati dai software engineers e dalle analisi e dalle valutazioni effettuate dai system engineers.

I system e gli hardware engineers vengono notevolmente agevolati dalla high-level synthesis in quanto consente loro di esplorare rapidamente varie scelte architettoniche per progettare hardware di buona qualità. Le loro esperienze pregresse, specialmente per gli hardware engineers, coprono l'ambito della sintesi RTL e ciò permette loro di comprendere facilmente le funzionalità messe a disposizione dai tool di high-level synthesis, che tipicamente forniscono direttive per la mappatura di costrutti di alto livello in strutture hardware di basso livello e vari parametri legati al throughput e alla latenza per limitare il timing dei progetti.

Infine, i verification engineers si occupano della verifica del progetto ad ogni fase del flusso di progettazione. La verifica del codice, infatti, deve essere effettuata per ogni processo di sintesi: la verifica si declina in varie fasi che vanno dalla verifica funzionale del codice ad alto livello a quella del codice RTL, effettuate tramite simulazione, prototipazione FPGA, verifica formale e altre tecniche.

I tool di high-level synthesis, combinati ad altri tool appositamente ideati, forniscono un ambiente senza soluzione di continuità che facilita l'interazione tra le varie figure professionali che collaborano alla progettazione dell'hardware, rendendo sempre più labile il confine tra una professione e l'altra [17].

Capitolo 3

Storia: dagli anni '70 agli anni 2010

Quali fattori hanno dato la spinta a una graduale transizione della progettazione digitale verso livelli di astrazione sempre più elevati e in particolare dall'RTL all'alto livello?

In questo paragrafo ci si propone di effettuare un breve excursus storico dell'HLS, dalla sua prima comparsa nel campo della ricerca, alla sua graduale applicazione in ambito industriale, mettendola a confronto con la sintesi RTL.

L'high-level synthesis è ancor oggi considerata una nuova frontiera della progettazione digitale.

Fino alla fine degli anni '60 i progettisti digitali non erano in possesso di alcuno strumento atto a facilitare il processo di sintesi digitale: i circuiti integrati venivano progettati e ottimizzati a livello di transistor. I progettisti dovevano quindi posizionare a mano ogni singolo transistor [18].

Una tale progettazione richiedeva una conoscenza specifica dell'hardware e time-to-market molto dilatati.

E' proprio in questo periodo che la legge di Moore viene formulata: da questo momento in avanti la tecnologia aumenterà esponenzialmente la sua complessità, rendendo la progettazione digitale un compito sempre più difficile da gestire a un basso livello di astrazione. In questo scenario, nasce la necessità di sviluppare delle metodologie di sintesi digitale per far fronte alla sempre maggiore complessità delle applicazioni e per riuscire a sfruttare al meglio il crescente potenziale della tecnologia in silicio.

L'high-level synthesis inizia a prendere piede nel campo della ricerca a partire dagli anni '70, tuttavia la complessità di questa tecnica di sintesi, l'iniziale mancanza di dialogo tra industria e ricerca, le basse prestazioni dei primi tool messi a disposizione dai vendori e l'eterogeneità dei tool presenti sul mercato hanno decretato una latenza nel suo impiego nel mondo industriale, in particolare come valida alternativa ai linguaggi HDL per la progettazione di sistemi digitali.

Nel 2009, Grant Martin e Gary Smith proposero di suddividere le fasi di ricerca nel campo della sintesi ad alto livello in generazioni [19]:

- generazione 0 (anni '70),
- generazione 1 (anni '80 - primi anni '90),
- generazione 2 (metà degli anni '90 - primi anni 2000),
- generazione 3 (anni 2000 - primi anni '10)
- avviamento verso la generazione 4, ipotizzato dagli autori del paper.

Come vedremo, solo a partire dagli anni 2000, ricerca e industria inizieranno a muoversi di pari passo, rispondendo la prima alle esigenze della seconda.

3.1 Ricerca VS Industria

3.1.1 Anni '70 - primi anni '80

E' in questo periodo che iniziano le prime ricerche nel settore della sintesi ad alto livello. I risultati ottenuti dai primi gruppi di ricerca nel campo dell'HLS della cosiddetta generazione 0 furono dirompenti: le ricerche si focalizzarono su come descrivere le specifiche di progetto, sulla simulazione e sulla sintesi, sia a livello register-transfer, sia a livello algoritmico. Tali ricerche non ebbero tuttavia alcun impatto nel campo industriale, dove invece comparvero strumenti di simulazione a livello di porte logiche e a livello di ciclo.

3.1.2 Seconda metà anni '80 - primi anni '90

Ogni anno i vendori di ASIC producevano tecnologie sempre più veloci e più economiche. Si trattò infatti dell'era dell'avvento degli FPGA. I vendori stavano rilasciando sul mercato nuove tecnologie a una velocità tale per cui per i progettisti iniziò a diventare molto difficile stare al passo con l'avanzamento tecnologico. Urgeva quindi una nuova metodologia di progettazione che fosse indipendente dalla tecnologia in sè.

E' così che i risultati delle ricerche dei decenni precedenti relativi alla sintesi a livello register-transfer – una strategia appunto technology-independent - si tradussero in dei veri e propri tool di sintesi RTL, che iniziarono a essere largamente impiegati dall'industria; la maggior parte dei progettisti si dedicò quindi allo studio di questi nuovi strumenti di sintesi RTL. I nuovi tool richiedevano in input linguaggi di descrizione dell'hardware (HDL) come il Verilog e il VHDL: grazie a questi nuovi strumenti, la progettazione non avveniva più a livello di porte logiche, ma consentiva di lavorare a un livello di astrazione più elevato, permettendo di gestire più agilmente progetti sempre più complessi e finestre di mercato sempre più brevi. Inizialmente i progetti non presentavano risultati buoni quanto quelli a livello di porte logiche, ma in compenso i nuovi tool consentivano di progettare e di effettuare la verifica dei progetti più velocemente, con anche un notevole abbassamento dei costi di riprogettazione nel caso in cui l'architettura progettata non avesse rispecchiato le specifiche iniziali di progetto.

Mentre la sintesi digitale mediante linguaggi HDL si diffonde nell'industria, l'high-level synthesis non viene considerata: non ha ancora raggiunto uno sviluppo tale da potersi proporre come una valida alternativa nel settore industriale. Presenta infatti diverse lacune. Tra queste vi era la tipologia dei linguaggi in input dei primi tool di sintesi digitale ad alto livello: adottare linguaggi nuovi e oscuri, come ad esempio Silange, per cimentarsi in una tecnica di progettazione sconosciuta e poco documentata non si presentava come una soluzione appetibile per i progettisti. Un'altra motivazione dello scarso successo della sintesi ad alto livello era legata alla qualità inadeguata dei risultati, caratterizzati da architetture semplici, allocazione costosa e scheduling primitivo. Inoltre, il dominio di specializzazione di questi nuovi tool (quello dei DSP) risultava essere già obsoleto rispetto alla nuova direzione della tecnologia, dominata dagli ASIC, basati sulla logica di controllo, piuttosto che sul flusso dei dati e sul signal processing.

3.1.3 Seconda metà degli anni '90 - primi anni 2000

Ben presto il successo dei tool di sintesi RTL inizia ad affievolirsi. La sintesi RTL ha permesso di aumentare ulteriormente la complessità dei progetti, non soltanto in termini di dimensione dei progetti stessi, ma anche a livello di complessità degli algoritmi. Il mondo della progettazione è cambiato radicalmente: quello del progettista non è più un lavoro individuale, ma per uno stesso progetto è necessario che lavorino in sinergia anche dieci progettisti contemporaneamente. Inoltre, la sintesi RTL presuppone una scelta a priori dell'architettura da implementare. Tuttavia, non sempre la scelta architettonica effettuata preliminarmente si rivela essere efficace, perciò, se le verifiche mostrano che l'architettura prescelta non soddisfa appieno i requisiti di partenza, è necessario ri-progettare l'architettura daccapo, con pesanti ripercussioni sul time-to-market. E' così che inizia a diffondersi tra i progettisti un senso di impotenza

dovuto all'impossibilità di indagare e valutare attentamente a priori diverse opzioni architetturali [13]. Nasce quindi l'esigenza di esplorare più architetture prima della scrittura del codice RTL. In altri termini, c'è bisogno di un tool o di una metodologia che permetta ai progettisti di catturare le idee del progetto in maniera indipendente dall'architettura, proprio come il linguaggio RTL aveva catturato l'essenza del progetto in modo indipendente dalla tecnologia.

E' così che l'high-level synthesis, finora relegata al mondo della ricerca, inizia a farsi strada nel campo dell'industria. E' in questo periodo, infatti, che i maggiori produttori di EDA (electronic design automation), quali Synopsys, Cadence e Mentor Graphic, iniziarono a rendere disponibili i primi tool di sintesi digitale ad alto livello. Tali tool tuttavia non tarderanno a manifestare le loro criticità: le generazione 2 dei tool di HLS si rivelerà essere un vero e proprio fallimento. In primo luogo, alcuni dei nuovi tool di HLS, invece che porsi come complemento della sintesi RTL, creando un flusso che usasse sia la sintesi HLS che quella RTL, restituivano un output direttamente al gate-level. In secondo luogo, anche le scelte effettuate in merito ai linguaggi in input furono errate: tali tool richiedevano in input descrizioni comportamentali in HDL, entrando direttamente in competizione con i tool di sintesi RTL pre-esistenti e non permettendo ai progettisti software di accostarsi a questo nuovo strumento di sintesi. Inoltre, i risultati erano di scarsa qualità, molto variabili e imprevedibili, con pochissimi fattori di controllo, e la verifica dei risultati molto ardua. Per di più, i tempi di simulazione erano lunghi tanto quanto la sintesi RTL poiché le specifiche in input richiedevano linguaggi HDL e la simulazione HDL era usata nel processo di progettazione.

Inoltre, la sintesi avveniva a compartimenti stagni, senza alcuna ottimizzazione tra i vari blocchi, e non utilizzava in modo efficace la memoria e i registri per salvare i risultati intermedi. I nuovi tool di sintesi non riuscivano a discernere tra data-flow e controllo, con risultati scarsi per la logica di branch, tipica del controllo; al contrario, la sintesi rivolta al dataflow e al signal processing produceva già buoni risultati, ma era destinata a una ristretta fetta di mercato.

In definitiva, anche questa generazione di tool fallì clamorosamente: tutte le caratteristiche riportate dai tool invece che agevolare i progettisti software - destinatari dei tool di sintesi, prima ancora dei progettisti hardware - sembravano remare contro alle loro esigenze.

3.1.4 Primi anni 2000 - primi anni '10

E' a partire dagli anni 2000 che l'high-level synthesis inizia a diffondersi sempre di più in ambito industriale: i tool di HLS offerti da diversi vendori iniziano a trovare maggiore consenso tra i progettisti. Tra i principali si ricordano Mentor Catapult C Synthesis, Forte Cynthesizer, Celoxica Agility, Bluespec, Synfora PICO Express ed Extreme, ChipVision PowerPot, NEC CyberWork-Bench, AutoESL AutoPilot, Xilinx AccelDSP e SystemGenerator, Esterel EDA Technologies Esterel Studio, Synopsys Synplicity Synplify DSP e il compilatore Cadence C-to-Silicon (C2S).

Alcuni di questi tool erano stati progettati ad hoc per progetti di ASIC, altri erano stati sviluppati appositamente per FPGA e altri ancora avevano come target entrambe le tecnologie ASIC e FPGA.

La maggior parte di questi tool impiega variazioni di C, C++ e SystemC come input (ad esempio Celoxica utilizza un dialetto speciale di C chiamato Handel-C), ma non è una scelta adottata universalmente. Ad esempio, Bluespec fa uso di costrutti in SystemVerilog e offre anche un meccanismo di input in SystemC. Alcuni dei tool orientati alla tecnologia DSP sono guidati da software come MathWorks Matlab e Simulink, anche se potenzialmente potrebbero usare C come formato intermedio.

La generazione 3 ha raggiunto un successo ragguardevole, specialmente in Giappone e in Europa. La ragione di questo successo è da ricercarsi nella scelta di linguaggi in input derivati da C, che hanno agevolato i progettisti algoritmici e di sistema, senza imporre loro lo studio di nuovi costrutti di linguaggi HDL, a loro poco familiari, o di altri linguaggi inusuali come era accaduto per le generazioni precedenti. Un altro motivo della popolarità conquistata dai tool della generazione 3 risiede nel fatto che la maggior parte dei tool è specializzata nella sintesi di hardware per ben precisi domini di applicazione, quelli in cui ci si aspetta di avere una maggior probabilità di successo, che vanno dal dominio del dataflow ai DSP (digital signal processor).

In generale la qualità dei risultati è comunque notevolmente migliorata, in particolare grazie alle ot-

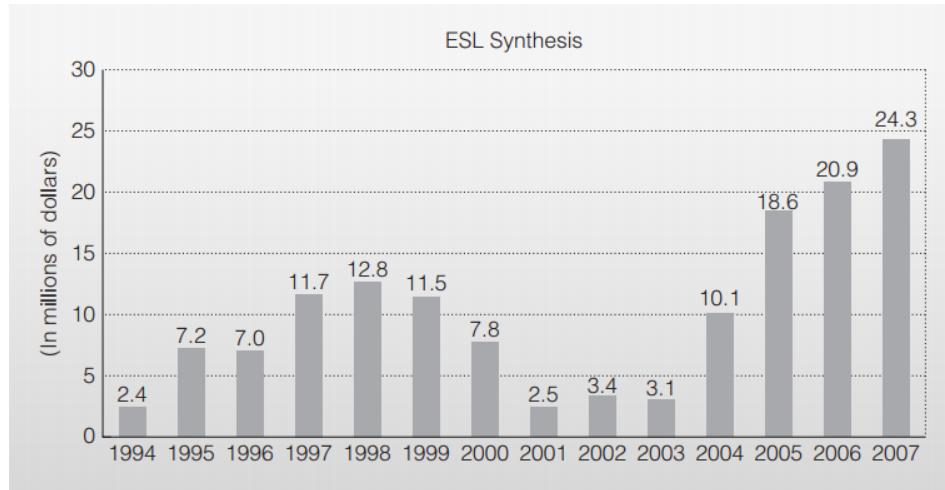


Figura 3.1: Andamento del mercato dei tool di HLS dal 1994 al 2007 [20]

timizzazioni basate su compilatore rese possibili dall’adozione di linguaggi provenienti dall’ambiente software, ampliati con costrutti orientati all’hardware.

Inoltre, a partire dalla generazione 3, gli FPGA hanno cominciato a ricoprire un ruolo di maggior rilievo rispetto ad altre tecnologie come gli ASIC: non sono soltanto in virtù della possibilità di riprogrammarli e per il costo più contenuto, ma anche perchè risultano essere più adatti alla rapida conversione di un algoritmo in hardware mediante tool di HLS [19].

Nell’ambito dell’esplorazione architetturale, i tool di HLS forniscono un metodo rapido per modellare funzionalità di diverse architetture e in tal modo derivare un limite superiore sulla rispettiva area di silicio impiegata per il progetto, sulle performance e sulla potenza. Questo permette ai progettisti di modellare diverse architetture, per poi scegliere la migliore tra esse e modificarla manualmente in codice RTL per ottenere efficiente tecnologia in silicio, riuscendo a soddisfare in maniera sempre più efficiente le richieste che avevano dato la spinta allo sviluppo di queste nuove tecniche di sintesi.

Come si può osservare dalla Figura 3.1, in seguito a un andamento altalenante, che rispecchia l’evoluzione delle generazioni di tool qui discusse, è stato proprio a partire dalla generazione 3 che il mercato della sintesi **ESL** (Electronic System-Level) ha iniziato a crescere. Nel 2008 aveva raggiunto \$ 29.5 milioni in totale.

Capitolo 4

Il flusso di progettazione della high-level synthesis

In questo capitolo si descriverà brevemente il flusso di progettazione dell'high-level synthesis. Il metodo è stato teorizzato dai primi gruppi di ricerca negli anni '80 ed è tutt'ora impiegato in termini concettuali nei tool di sintesi, a meno di piccole integrazioni o varianti. I vari task previsti dal metodo sono oggi eseguiti da algoritmi sempre più sofisticati, che hanno permesso all'high-level synthesis di evolversi, ottenendo una qualità dei risultati sempre migliore.

Nei tool di HLS, la descrizione dell'input del sistema che si intende progettare può essere effettuata mediante linguaggi ad alto livello, come C/C++. L'output generato dai tool è generalmente una register-transfer level netlist dell'hardware, ovvero una lista dei componenti hardware necessari alla realizzazione del progetto scritta in hardware description language (HDL).

Il flusso di progettazione, mostrato in Figura 4.1, può essere diviso in due fasi: frontend e backend.

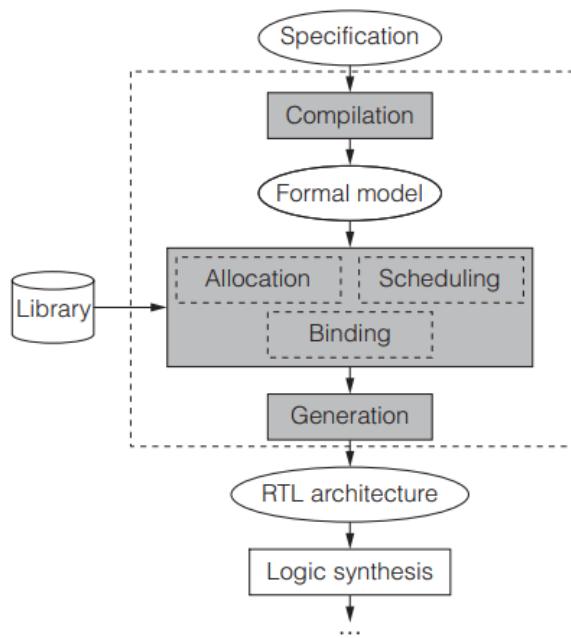


Figura 4.1: Step di progettazione nella sintesi ad alto livello [18]

La fase di frontend comprende la definizione delle specifiche, la compilazione e l'elaborazione di un modello formale; la fase di backend prevede tre passaggi interdipendenti, denominati allocation, scheduling e binding, che consentono di generare l'architettura RTL, output della high-level synthesis, attingendo a

una libreria contenente componenti descritti in RTL.

Proseguendo nel flusso di progettazione dell'hardware, l'output dei tool di HLS verrà poi fornito come input a un tool di sintesi logica per generare una netlist a livello di gate, che sarà a sua volta trasformata attraverso un tool di back-end per l'implementazione su piattaforme hardware, come FPGA o ASIC.

4.1 L'input del processo di sintesi

La descrizione in input consiste in una descrizione algoritmica del comportamento del sistema, che si presenta quindi in maniera del tutto simile a un algoritmo scritto con i tradizionali linguaggi di programmazione come C/C++, con la differenza che è impiegato per descrivere dell'hardware. Tale descrizione non contiene tuttavia informazioni strutturali come la tipologia di componenti o le loro interconnessioni, né fornisce informazioni riguardo alla struttura del circuito, come il numero di stadi della pipeline o informazioni sul clock. Nel flusso di progettazione dei tool di HLS vi è inoltre la possibilità di inserire direttive e pragma (gestite dal precompilatore) a carattere hardware-oriented e di inserire dei vincoli di progettazione che definiscono delle caratteristiche (come la latency o la frequenza di clock) che il progetto dovrà soddisfare. Un semplice codice d'esempio scritto in linguaggio C viene fornito in Figura 4.2.

```
void fir_filter (int *input, int coeff[NUM_TAPS], int *output)
{
    int delay_line[NUM_TAPS];
    int accumulate = 0;
    int i;
    delay_line[0] = *input ;
    for(i = NUM_TAPS-1; i >= 1; i--){
        delay_line[i] = delay_line[i-1];
    }
    // Calculate FIR value
    for(k = NUM_TAPS-1;k >= 0; k--){
        accumulate = accumulate +coeff[k] * delay_line[k];
    }
    *output = accumulate >> 7;
}
```

Figura 4.2: Esempio di codice in C per un filtro FIR [21]

4.2 Compilazione delle specifiche

L'high-level synthesis comincia con la compilazione delle specifiche funzionali. Questo primo step trasforma l'input in una rappresentazione formale interna (IR - internal representation) e include alcune ottimizzazioni del codice, come l'eliminazione del deadcode e di false dipendenze tra i dati, il ripiegamento del codice e la trasformazione dei loop. Il modello formale prodotto dalla compilazione è un data flow graph (DFG), ovvero un grafo che consiste in una serie di nodi, in cui ogni nodo rappresenta un'operazione. Se due nodi sono connessi significa che tra di essi sussiste una dipendenza a livello di dati: un arco tra due nodi può rappresentare input, output o variabili temporanee. In virtù di questa dipendenza, le due operazioni rappresentate dai nodi dovranno essere eseguite in sequenza. Nei sistemi in cui sono presenti loop oppure se la sequenza di controllo è basata su condizioni esterne, la compilazione produrrà un Control and Data Flow Graph (CDFG), un grafo diretto, che completa il DFG precedente con dei nodi di controllo, che rappresentano costrutti di controllo come loop e branch. In alcuni casi è possibile rimuovere dal modello le dipendenze di controllo delle specifiche iniziali al momento della compilazione e ottenere un DFG. Per farlo i loop sono completamente srotolati attraverso la conversione in blocchi non iterativi e i blocchi condizionali sono implementati tramite dei multiplexer. Il DFG risultante esibisce esplicitamente un parallelismo intrinseco delle specifiche. Tuttavia, le trasformazioni richieste possono portare a una rappresentazione formale molto pesante che richiede un considerevole spazio in memoria durante la sintesi. In più questa rappresentazione non supporta i loop unbounded né dichiarazioni di controllo non-statiche come *goto*. Questo limita l'uso del DFG puro a poche applica-

zioni.

I CDFG sono dunque più espressivi dei DFG: non solo esibiscono le dipendenze che sussistono tra i dati all'interno dei blocchi di base, ma mostrano anche il flusso di controllo tra i blocchi. Tuttavia il parallelismo è esplicito solo all'interno dei blocchi di base e sono richieste analisi addizionali e trasformazioni per rendere esplicito il parallelismo che potrebbe esistere tra i vari blocchi. Queste trasformazioni includono lo srotolamento dei loop, il loop pipelining e il loop merging. Queste tecniche permettono di rivelare il parallelismo tra i loop e tra le iterazioni dei loop e consentono di ottimizzare le latenze, il throughput, la dimensione e il numero di accessi in memoria.

4.2.1 Per approfondire: le ottimizzazioni del compilatore

Molti tool di HLS si avvalgono di framework di compilatori software nel loro front-end, che consentono di effettuare ottimizzazioni standard del compilatore, ma anche alcune ottimizzazioni specifiche per l'high-level synthesis, come la bitwidth optimization.

Le ottimizzazioni messe a disposizione dai compilatori sono molteplici, alcune delle quali agiscono a livello del codice sorgente, altre a livello di intermediate representation (IR); l'ordine in cui i vari step di ottimizzazione vengono eseguiti incide sulla qualità dell'RTL generato. Il problema di ordinare i passaggi di ottimizzazione del compilatore, noto anche come phase-ordering problem, è stata un'area di ricerca attiva negli ultimi decenni.

In questo elaborato ci limiteremo ad elencare alcune delle ottimizzazioni più comuni indispensabili per il processo di high-level synthesis.

Tali ottimizzazioni sono finalizzate a ottenere una riduzione della complessità del codice, massimizzando la località dei dati ed esponendo il parallelismo. Le trasformazioni tipicamente impiegate per la sintesi di hardware, alcune delle quali già menzionate, comprendono:

- ottimizzazioni del codice come eliminazione del deadcode e del codice ridondante;
- semplificazione aritmetica delle espressioni;
- ottimizzazione della bitwidth, una trasformazione che mira a ridurre il numero di bit richiesti dagli operatori nel datapath; è molto importante perché ha un impatto su tutti i requisiti non funzionali (ad esempio, prestazioni, area e potenza) di un progetto, senza alterarne il comportamento;
- analisi delle dipendenze tra i dati per scoprire il parallelismo tra gli accessi a puntatori e agli array;
- ottimizzazioni della memoria come riutilizzo della memoria, scalarizzazione e partizionamento di array per ridurre il numero di accessi alla memoria e migliorare la memory bandwidth;
- loop transformation per esporre il parallelismo a livello di loop; ad esempio, il loop unrolling rende visibile il parallelismo esistente tra iterazioni successive del loop stesso; il loop pipelining fa in modo che un'iterazione del loop possa iniziare prima del completamento della precedente; il loop merging permette di unire due loop sequenzialmente adiacenti in un unico loop con lo stesso comportamento;
- ottimizzazioni delle funzioni come ad esempio l'inlining, che sostituisce la chiamata a una funzione con il corpo della funzione stessa.

4.3 Allocation, scheduling e binding

A partire dalla IR ottenuta nella fase di compilazione delle specifiche ad alto livello e da una libreria RTL dei componenti, verranno eseguite le operazioni di allocation, scheduling e binding. E' necessario precisare che questi task sono interdipendenti e per ottenere risultati ottimali dovrebbero essere effettuati simultaneamente. Vista l'enorme difficoltà computazionale di un tale processo, queste operazioni sono generalmente eseguite in sequenza, che può variare a seconda dei vincoli di progettazione e delle

direttive impostate dall’utente. L’ordine in cui queste vengono eseguite può infatti avere un impatto significativo sulla qualità del progetto. In ambito accademico, sono molti gli studi condotti in merito alla possibilità di valutare diversi flussi di progetto in maniera automatica, in altri termini per esplorare lo spazio di progettazione; i recenti avanzamenti in tale campo verranno presentati nel Capitolo 4. Le operazioni di allocation, scheduling e binding porteranno alla formazione di un datapath e di un’unità di controllo.

Allocazione delle risorse hardware

L’allocazione è il processo in cui vengono stabilite le risorse hardware (ad esempio unità funzionali, memoria, componenti di connessione) necessarie all’implementazione del progetto. Durante l’allocazione vengono inoltre definiti uno schema relativo al clock, la gerarchia della memoria e l’uso di pipeline.

Alcuni componenti aggiuntivi possono essere allocati anche durante le operazioni di scheduling e binding. Per esempio, i componenti connettivi (come i bus oppure le connessioni point-to-point tra i componenti) possono essere aggiunti prima oppure dopo lo scheduling e il binding.

L’obiettivo dell’allocazione è di realizzare un trade-off tra il costo del progetto e le performance. I componenti che verranno impiegati nel progetto sono selezionati dalla libreria RTL; quest’ultima contiene informazioni relative alle caratteristiche dei componenti (come l’area, il delay, la potenza dissipata), in modo tale da poter effettuare una stima della qualità del progetto sulla base dei componenti prescelti.

Scheduling

Lo scheduling consiste nell’assegnazione delle operazioni da eseguire e degli accessi in memoria ai control step. Un control step è l’unità fondamentale nei sistemi sincroni e corrisponde a un ciclo di clock. Tutte le operazioni presenti nel codice sorgente devono essere schedulate in ben precisi cicli di clock. A seconda del componente funzionale in cui l’operazione è mappata, essa può essere eseguita all’interno di un solo ciclo di clock o svolgersi su più cicli. Le operazioni possono essere concatenate (l’output di un’operazione alimenta direttamente l’input di un’altra operazione) oppure possono essere schedurate per essere eseguite in parallelo a patto che non ci siano dipendenze a livello di dati tra le operazioni e che ci siano risorse disponibili a sufficienza nello stesso momento. Lo scheduling deve riuscire a soddisfare i vincoli di progetto legati al numero dei control step, al delay, alla potenza dissipata e alle risorse hardware impiegate.

Se l’utente ha completamente specificato tutte le risorse disponibili e la lunghezza del ciclo di clock durante l’allocazione, l’obiettivo dell’algoritmo di scheduling è quello di produrre un progetto con le migliori performance possibili o con il minor numero di cicli. Lo scheduling deve cioè massimizzare l’uso delle risorse allocate. Questo approccio prende il nome di resource-constrained scheduling.

Se invece le specifiche di progetto impongono dei vincoli ben precisi in termini di performance, l’obiettivo dell’algoritmo di scheduling è quello di realizzare un progetto con il minor costo possibile o con il minor numero di unità funzionali. Tale approccio prende il nome di time-constrained scheduling.

Binding

L’operazione di binding assegna le operazioni e gli accessi in memoria schedulati in ben precisi cicli di clock a un’unità hardware disponibile. Una risorsa come un’unità funzionale, di storage o di interconnessione, può essere condivisa per svolgere diverse operazioni, accessi ai dati o per trasferimenti di dati se tali operazioni sono mutuamente esclusive. Due operazioni sono mutuamente esclusive se non saranno mai eseguite simultaneamente; in virtù di ciò possono essere assegnate alla stessa unità hardware.

L’allocazione garantisce che ci siano risorse a sufficienza per implementare il progetto e lo scheduling utilizza tali risorse per assegnare le operazioni ai cicli di clock. Ma i componenti effettivi da utilizzare per ogni operazione non sono ancora stati assegnati ed è in questo che consiste l’operazione di binding: data un’operazione, se ci sono diverse unità che assolvono a una stessa funzione, l’algoritmo di binding deve ottimizzare la selezione della risorsa per svolgere quella determinata operazione.

A seconda della tipologia delle unità che devono essere assegnate, il binding si declina in tre fasi

strettamente legate tra loro:

- lo storage binding, in cui le variabili vengono assegnate alle unità di storage, come registri, register file e unità di memoria;
- il functional-unit binding, che assegna ciascuna operazione all'interno di uno step di controllo a un'unità funzionale;
- l'interconnection binding, che assegna un'unità di interconnessione come un multiplexer o un bus per ogni trasferimento di dati tra porte, unità funzionali e unità di storage.

4.4 La generazione dell'architettura RTL

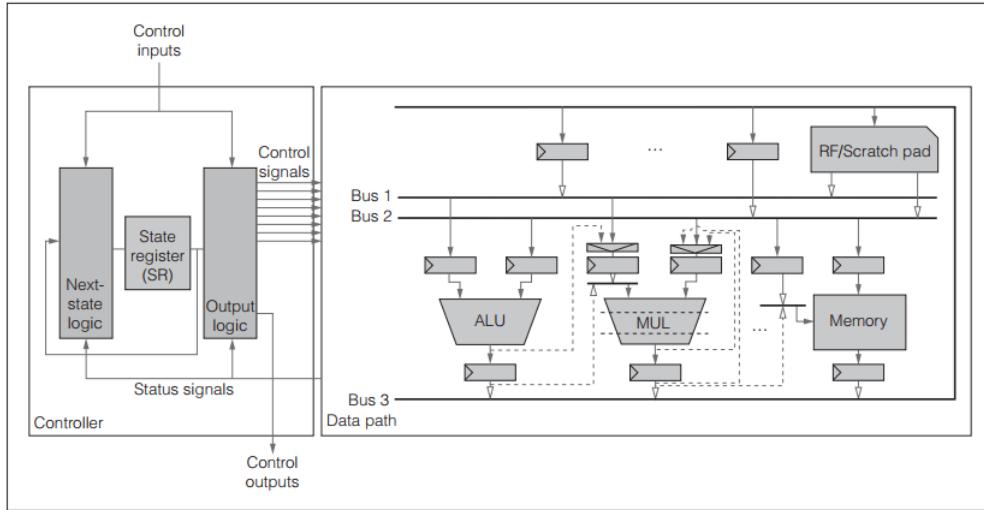


Figura 4.3: Esempio di architettura RTL [18]

Tutte le decisioni prese in merito al progetto durante le fasi di allocazione, scheduling e binding concorrono alla determinazione dell'architettura RTL [18]. L'architettura RTL è implementata da un insieme di componenti register-transfer e generalmente è costituita da un controller e da un data path. Quest'ultimo consiste in un insieme di elementi di memoria (registri, register file e memorie), un insieme di unità funzionali (ALU, moltiplicatori, shifter e altre funzioni) ed elementi di interconnessione (tristate driver, multiplexer e bus). Tutti questi componenti register-transfer possono essere allocati in diverse quantità e tipologie e connessi arbitrariamente lungo il bus. Ogni componente può eseguire operazioni in uno o più cicli di clock, può fare uso di pipeline e può avere registri di input o di output.

In generale, il progetto digitale si interfaccia con il mondo esterno per trasferire dati e segnali di controllo (ad esempio per la sincronizzazione e per protocolli di interfaccia handshaking) mediante porte primarie di input e di output. Il datapath riceve in ingresso gli input provenienti dall'esterno e segnali di controllo dal controller; in uscita fornisce gli output e invia segnali di stato al controller.

Il controller è una macchina a stati finiti che gestisce il flusso dei dati nel data path impostando i valori dei segnali di controllo, che possono essere ad esempio gli input selezionati delle unità funzionali, dei registri e dei multiplexer. Oltre ai segnali di controllo rivolti al datapath, il controller fornisce in uscita anche degli output di controllo. Gli input del controller si dividono in input primari, ovvero quelli specificamente del controller, provenienti dall'esterno, e in segnali di stato, provenienti dai componenti del data path.

Il controller è costituito da un registro di stato (SR), dalla logica dello stato futuro (next-state logic) e dalla logica di output (output logic). Il registro SR memorizza lo stato corrente della macchina a stati

finiti. La logica del next-state calcola lo stato futuro che sarà caricato nel registro SR, mentre la logica di output genera i segnali e gli output di controllo [18]

4.5 L'output del processo di sintesi

In seguito, l'architettura RTL viene scritta sottoforma di register-transfer level netlist: un netlister genera la struttura finale RTL, che consiste appunto in una netlist dei componenti RTL e un modello di simulazione di ogni componente. Una netlist è utilizzata per passare i dati di progetto da un tool a un altro. L'output dei tool di HLS viene infatti fornito come input a un tool di sintesi logica per generare una netlist gate-level, che sarà in seguito data in input a un tool di back-end per l'implementazione su hardware.

4.6 Un esempio concreto del flusso di progettazione ad alto livello

Sia $Y = \max((A >> 1) + (B - (C >> 3)), B)$ l'espressione che si vuole implementare in hardware; essa contiene operazioni quali addizioni, sottrazioni, shift a destra (indicato con la dicitura shr nelle Figure) e ricerca del massimo.

Attraverso questo semplice esempio si percorreranno le fasi della sintesi appena discusse. Tale esempio, tratto da un paper del 1994 [22], è ancora attuale nel descrivere in termini assolutamente generali gli step della sintesi ed è stato scelto perché consente di toccare con mano e di avvicinarsi in modo più concreto alle dinamiche, ma soprattutto alle difficoltà in cui si imbatte il progettista nello sviluppo di un progetto. L'high-level synthesis, effettuando automaticamente le operazioni che verranno ripercorse di seguito, si configura come una risorsa importantissima, capace di agevolare incommensurabilmente il lavoro dei progettisti.

Front-end: dal codice sorgente alla IR

Per prima cosa l'espressione deve essere compilata e quindi trasformata in una rappresentazione intermedia (IR), come quella mostrata in Figura 4.4.

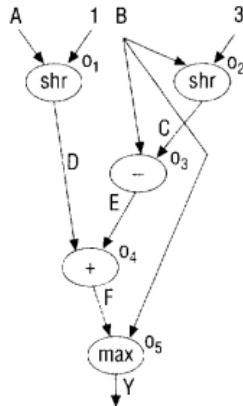


Figura 4.4: Rappresentazione mediante DFG [22]

Nel codice sorgente ad alto livello vengono inserite anche delle condizioni al calcolo di Y : Y viene calcolato solo se un input esterno X è asserito e se il Timer non è uguale a 0. La rappresentazione deve quindi essere completata mediante un CDFG.

Nel CDFG ottenuto (Figura 4.5) è presente un nodo di attesa che controlla l'asserimento della condizione esterna X ; un altro nodo if-begin controlla che il Timer non sia uguale a 0 e un nodo if-end rappresenta la fine dello statement di branch.

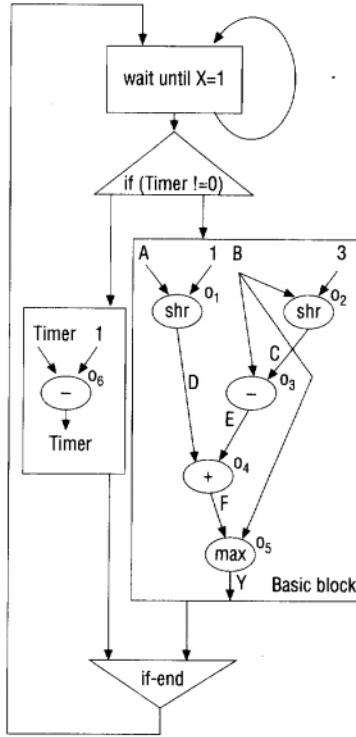


Figura 4.5: Rappresentazione mediante CDFG [22]

Back-end: allocation, scheduling e binding

La rappresentazione interna viene poi convertita in una macchina a stati finiti (FSM), che implementa i costrutti di controllo, e in un datapath, che esegue le operazioni tra le variabili.

Il modello della macchina a stati finiti e del datapath e la sua implementazione sono mostrati in Figura 4.6.

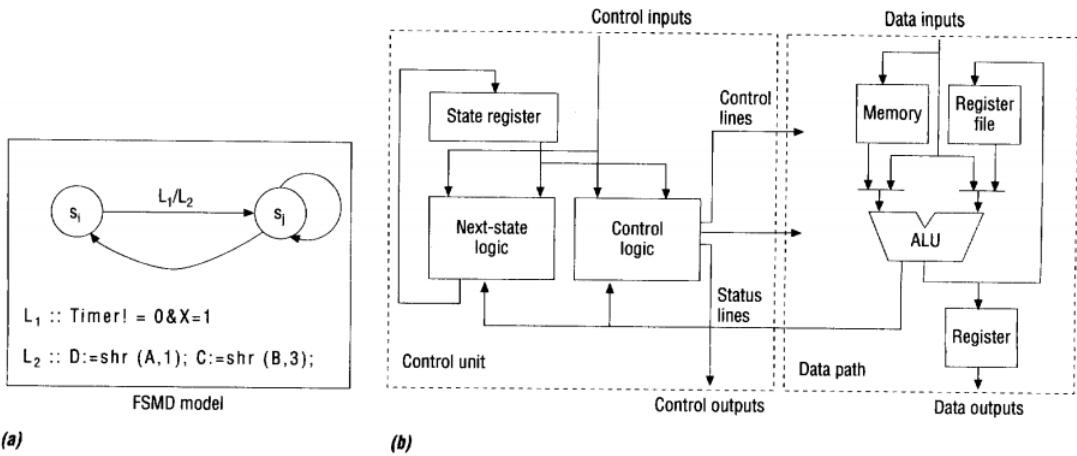


Figura 4.6: Modello (a) e implementazione (b) di una macchina a stati e del datapath [22]

Tale implementazione è solo una delle possibili implementazioni, che sarà caratterizzata da ben precise caratteristiche in termini di performance e di area di progetto.

L'architettura generata è costituita da componenti hardware che vengono attinti da una libreria RTL (Figura 4.7). Nel caso in esame, si dispone di una libreria RTL contenente 3 componenti: due diverse implementazioni di una ALU, l'unità logico aritmetica, chiamate ALU-F e ALU-S, che provvederanno

Components (operations)	Delay (ns)	Area (μm^2)
ALU-F (+/-/shr)	20	600
ALU-S (+/-/shr)	70	400
MAX (max)	80	800

Figura 4.7: Libreria RTL [22]

alle operazioni di addizione, sottrazione e shift e un componente chiamato MAX, capace di trovare il massimo tra due operandi. La ALU-F è molto veloce, capace di eseguire le operazioni in 20 ns; la ALU-S richiede 70 ns per completare l'operazione, ma richiede un area minore ed è più economica. Utilizzando questa libreria, ci sono diverse allocazioni possibili delle unità funzionali per il DFG della Figura 4.4. Per scegliere l'implementazione, è necessario fare delle valutazioni in termini di area e

Allocation	No. of ALU-F	No. of ALU-S	No. of MAX	Area (μm^2)
A	0	1	1	1,200
B	1	0	1	1,400
C	0	2	1	1,600
D	1	1	1	1,800
E	2	0	1	2,000

Figura 4.8: Possibili allocazioni per il DFG [22]

performance. Una semplice stima dell'area per ogni scelta di allocazione è data dalla somma delle aree di ogni singolo componente della libreria. E' possibile inoltre stimare la performance per ogni allocazione come mostrato in Figura 4.9.

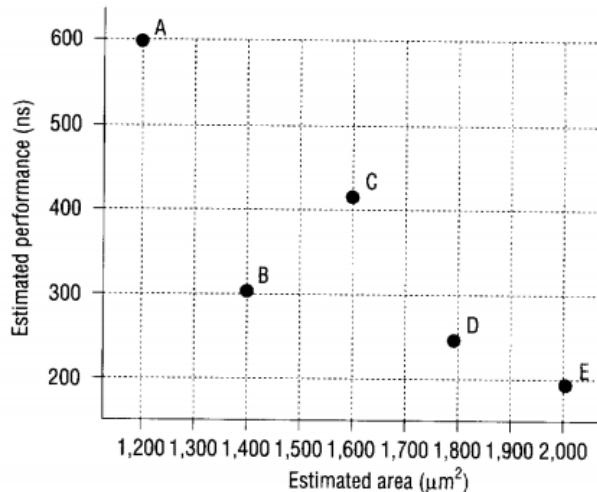


Figura 4.9: Tradeoff tra area e performance [22]

La Figura 4.9 mostra che l'allocazione A consuma la minor area, ma risulta la peggiore in termini di performance. L'allocazione E invece è quella che produce la miglior performance, ma è anche il progetto più costoso. La scelta tra una delle cinque soluzioni di allocazione proposte per la sintesi dipende dai vincoli e dai requisiti del progetto.

Simili valutazioni dovranno essere effettuate anche per l'allocazione delle unità di storage e per quelle di interconnessione, determinando così il numero ottimale di risorse di ciascuna tipologia.

Nella fase di allocazione, oltre alla selezione delle risorse che verranno messe a disposizione, viene anche scelta la durata del ciclo di clock. Nel caso in esame viene selezionata l'opzione E (Figura 4.8), con due ALU veloci (ALU-F) e un'unità di MAX, con performance stimate a 200 ns; il ciclo di clock viene

fissato a 50 ns.

Si passa quindi all'operazione di scheduling, che deve massimizzare l'utilizzo delle risorse allocate, realizzando un progetto con le migliori performance (o con il minor numero di cicli di clock), secondo un approccio resource-constrained.

La Figura 4.10 mostra la schedule finale. Come si evince dalla Figura, l'algoritmo ha potuto schedulare entrambe le operazioni di shift (o_1 e o_2) nel primo step di controllo in quanto non dipendono da operazioni precedenti e perché ci sono 2 ALU disponibili. Quando l'algoritmo ha schedulato i nodi nel secondo step di controllo, ha determinato che, sia la sottrazione o_3 , sia l'addizione o_4 potevano essere schedurate nello stesso ciclo di clock dal momento che ci sono 2 ALU disponibili. Infine, il nodo MAX non è stato schedulato fino a quando il suo predecessore (operazione o_4) non ha calcolato il risultato e quindi esso è stato inserito nello stato successivo. Le operazioni o_3 e o_4 , pur essendo tra loro dipendenti,

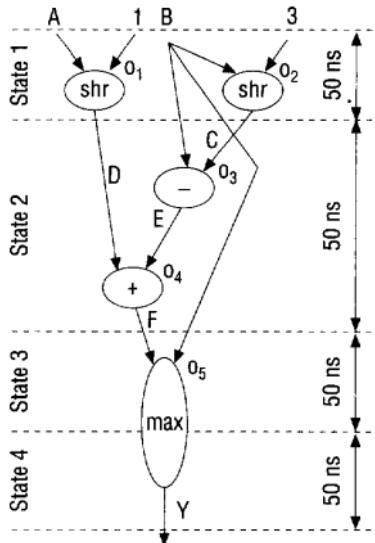


Figura 4.10: Esempio di schedule basato sull'allocazione E [22]

sono state schedulate nello stesso stato. Tale processo di scheduling di due nodi dipendenti è chiamato chaining; esso è possibile solo se è disponibile un numero sufficiente di componenti e se i loro delay durano meno di un ciclo di clock. Nel nostro esempio, le due ALU allocate hanno un delay di 20 ns; dal momento che sono entrambe disponibili, è possibile concatenare l'addizione e la sottrazione in un unico periodo di clock di 50 ns. Se invece il delay di un componente supera un ciclo di clock, lo scheduler deve permettere l'impiego di diversi stati per completare le operazioni, proprio come avviene per il componente MAX. Quest'ultimo infatti ha un delay di 80 ns, mentre il periodo di clock è di soli 50 ns. Quindi, l'operazione o_5 richiede due cicli di clock per essere completata. Questo tipo di scheduling, in cui le operazioni richiedono più di un ciclo di clock per essere completate, prende il nome di multicycling.

Si prosegue quindi con la fase di binding. Come visto in precedenza, il binding si divide in storage binding, functional-unit binding e interconnection binding. Per semplicità si illustrerà brevemente solo la fase di storage binding, dove ciascuna variabile viene assegnata a un dato registro. Per farlo, è necessario partizionare tutte le variabili descritte in insiemi compatibili. Un insieme di variabili è compatibile se tutte le variabili nell'insieme non sono attive nello stesso momento. Per determinare gli insiemi compatibili, è necessario determinare il tempo di vita (lifetime) delle variabili, come mostrato in figura 4.11.a. Dal tempo di vita, viene creato un grafo di compatibilità, in cui ogni nodo rappresenta una variabile e un arco connette due variabili con tempi di vita mutuamente esclusivi. Nel nostro esempio, le variabili C e D sono scritte nello stato 1 e lette nello stato 2, la variabile E è scritta e letta nello stato 2 e la variabile F è scritta nello stato 2 e letta nello stato 3 e 4. La Figura 4.11.b mostra il grafo di compatibilità di queste variabili. Poi, il grafo di compatibilità deve essere partizionato in sottografi completamente connessi, cioè sottografi contenenti nodi diversi in cui ogni nodo è connesso a tutti i suoi vicini. Un sottografo indica quindi un insieme di nodi mutuamente esclusivi che possono essere assegnati alla stessa risorsa.

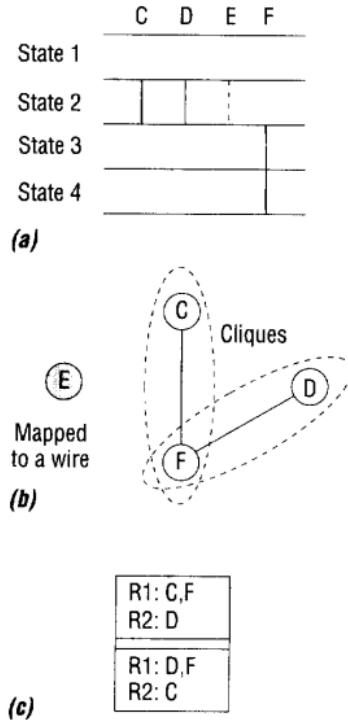


Figura 4.11: Storage binding: lifetime delle variabili (a), grafo di compatibilità (b), soluzioni possibili (c) [22]

Nel nostro esempio, sono proposte due possibili soluzioni per partizionare il grafo di compatibilità, che sono mostrate in Figura 4.11.c. Entrambe le soluzioni richiedono due registri per memorizzare le quattro variabili. La prima soluzione usa il registro R1 per salvare le variabili C ed F e il registro R2 per la variabile D. La variabile E non richiede la memorizzazione perché non è attiva su un confine tra stati ed è implementata con una connessione cablata (wire connection). Una volta effettuate anche le operazioni di functional-unit binding e interconnection binding, la sintesi è conclusa.

Le operazioni di ottimizzazione illustrate vengono eseguite automaticamente dai tool di high-level synthesis mediante degli algoritmi eseguiti sequenzialmente. Quindi lo scopo dei tool di sintesi ad alto livello è fare in modo che queste stime, qui effettuate a mano, vengano svolte in maniera automatica, aiutando così il progettista a fare la scelta più adatta mediante un'agile esplorazione dello spazio di progettazione.

Capitolo 5

I vantaggi dell'high-level synthesis

L'high-level synthesis apporta al processo di progettazione dell'hardware molti vantaggi, che si affiancano al processo di sintesi in sè. Analizziamone alcuni in dettaglio.

5.1 Esplorazione dello spazio di progettazione

L'esplorazione dello spazio di progettazione consiste nella scelta dell'architettura più adatta a realizzare un dato progetto hardware, effettuando un trade-off tra area e performance.

Plotando l'area sul tempo di elaborazione per delle particolari specifiche si ottiene un grafico come quello in Figura 5.1, che rappresenta quello che prende il nome di spazio di progettazione a livello register-transfer [8].

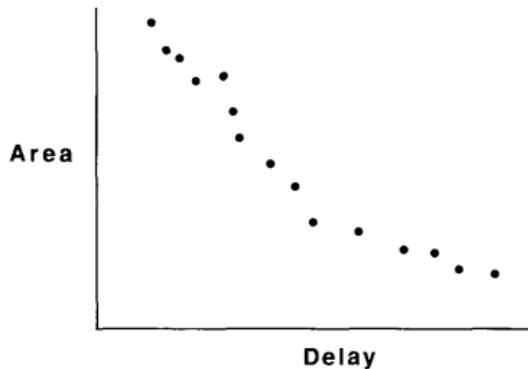


Figura 5.1: Spazio di progettazione register-transfer level per delle particolari specifiche [8]

La sintesi ad alto livello, essendo una metodologia di sintesi indipendente dall'architettura, si propone di delegare ai tool di sintesi le diverse scelte architettoniche, in modo tale che vengano vagliate da degli algoritmi, invece che eseguite a mano [23].

I tool di HLS contengono varie direttive che possono guidare la sintesi hardware per generare progetti più efficienti. Queste direttive consentono di implementare pipeline e operazioni come il loop unrolling o il partizionamento di array. Poiché la maggior parte degli algoritmi contiene numerosi loop e array di dati, trovare delle impostazioni delle direttive che portino a buone soluzioni può essere un compito arduo.

Attualmente diversi tool di HLS consentono di esplorare lo spazio di progettazione intervenendo sull'ordine delle fasi di ottimizzazione del compilatore comportamentale. Tra questi vi sono Vivado HLS e LegUp. Tali tool impiegano come front-end compilatori software general-purpose, che utilizzano tecniche tradizionali per ottimizzare l'intermediate representation (IR) del programma. Quindi la qualità delle ottimizzazioni del compilatore nella fase di front-end si ripercuote direttamente sui circuiti generati

dall'HLS.

Trovare la sequenza ottimale delle fasi di ottimizzazione è un problema computazionale molto complesso - si tratta infatti di un problema NP-difficile, ovvero un nondeterministic polynomial-time hard problem, la cui spiegazione esula dallo scopo di questo elaborato - e valutare in maniera esaustiva tutte le sequenze dei numerosi parametri non sarebbe possibile [23].

I maggiori tool di HLS di oggi non consentono all'utente di effettuare automaticamente l'esplorazione dello spazio di progettazione, ma la ricerca sta lavorando in questa direzione [24].

L'esplorazione automatica dello spazio di progettazione (design space exploration - **DSE**) nell'ambito della sintesi ad alto livello consiste in un problema di ottimizzazione multi-obiettivo in quanto il suo scopo è quello di minimizzare un insieme di parametri di progetto in conflitto tra loro. Questi includono tipicamente l'area, le performance e la potenza. Quindi l'obiettivo della DSE nel contesto dell'high-level synthesis non è quello di trovare una singola soluzione, ma un insieme di progetti, chiamati tipicamente progetti o micro-architetture Pareto-ottimali, cioè - semplificando - i progetti con la minor area e la minore latenza. Dal momento che lo spazio di esplorazione è molto vasto, nella pratica non è possibile realizzare l'ottimizzazione di Pareto e quindi, la curva ottenuta è spesso un tradeoff. L'obiettivo principale dell'esplorazione è quello di generare automaticamente un insieme di parametri chiamati knob e invocare poi il tool di HLS per minimizzare una funzione data. Il tool HLS prende gli knob generati dall'esplorazione, la descrizione comportamentale da sintetizzare e le librerie RTL e genera il nuovo codice RTL con caratteristiche di progetto uniche in termini di area, latenza, delay e potenza.

I primi lavori sull'HLS DSE - su cui ancora si basano i tool di HLS odierni - facevano uso di algoritmi di allocation, scheduling e binding differenti oppure eseguivano gli algoritmi secondo ordini differenti per ottenere diverse microarchitetture. Il principale svantaggio di questa metodologia è che assumono che il progettista abbia accesso al processo HLS, cosa che nella pratica non avviene.

La maggior parte dei lavori di ricerca attuali assume che il processo di HLS sia un black box e impostano gli knob prima che il processo di HLS sia eseguito. Solo in un secondo momento invocano il tool e sulla base della qualità dei risultati riportata, generano un nuovo insieme di knob.

Ci sono tre famiglie principali di knob nei maggiori tool commerciali: attributi locali sottoforma di pragma inseriti direttamente nel codice sorgente, opzioni di sintesi globale che influenzano l'intera descrizione comportamentale e il numero di FU (functional unit) [23].

1. *Pragma*: sono delle direttive di sintesi ampiamente utilizzate dalla maggior parte degli strumenti HLS commerciali. I pragma vengono inseriti sottoforma di commenti direttamente in corrispondenza della descrizione comportamentale. Il vantaggio principale è che i pragma consentono di avere un controllo individuale sui principali costrutti della descrizione comportamentale che hanno il più alto impatto sulla micro-architettura finale, ovvero loop, array e funzioni. Attraverso i pragma, ad esempio, è possibile decidere se effettuare o meno lo srotolamento dei loop. Gli array possono essere mappati su registri o memorie di diversi tipi e le porte e le funzioni possono essere inline o no. Esplorare tutte le diverse combinazioni è pressocchè impossibile per progetti con un numero elevato di loop, array e funzioni. L'impostazione manuale dei pragma porta a un'unica microarchitettura, ma non è detto che si tratti di una soluzione ottimale.
2. *Opzioni di sintesi globale*: sono normalmente specificate in uno script di sintesi e applicate all'intera descrizione comportamentale da sintetizzare. Molte di queste opzioni sono analoghe alle direttive della sintesi locale (relative quindi alla sintesi di loop, array e funzioni), ma con la differenza che vengono applicate a tutti i loop, gli array e le funzioni presenti nella descrizione comportamentale, senza la possibilità di specificare opzioni diverse per le singole operazioni. Tra le altre opzioni che si possono settare troviamo la tipologia di scheduling oppure vincoli legati al clock e alla pipeline.
3. *Functional Unit (FU)*: influenzano il parallelismo che può essere estratto dalla descrizione comportamentale. Attraverso una tecnica di condivisione delle risorse è possibile ridurre l'area di progetto: una singola FU viene riutilizzata tra diverse operazioni computazionali nella descrizione comportamentale. Spesso i tool commerciali adottano tecniche resource-constrained, dando

inizio al processo di sintesi con la fase di allocation delle risorse, che determina un vincolo in termini di FU, che può comunque essere modificato.

Classificazione dei metodi di DSE

La Figura 5.2 mostra una classificazione delle tecniche DSE proposte, che si dividono in synthesis based, supervised learning e model based.

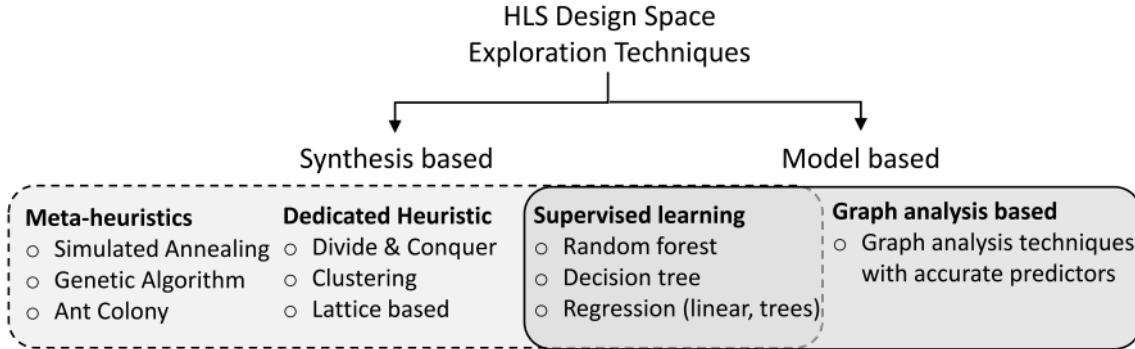


Figura 5.2: Classificazione delle tecniche di DSE [23]

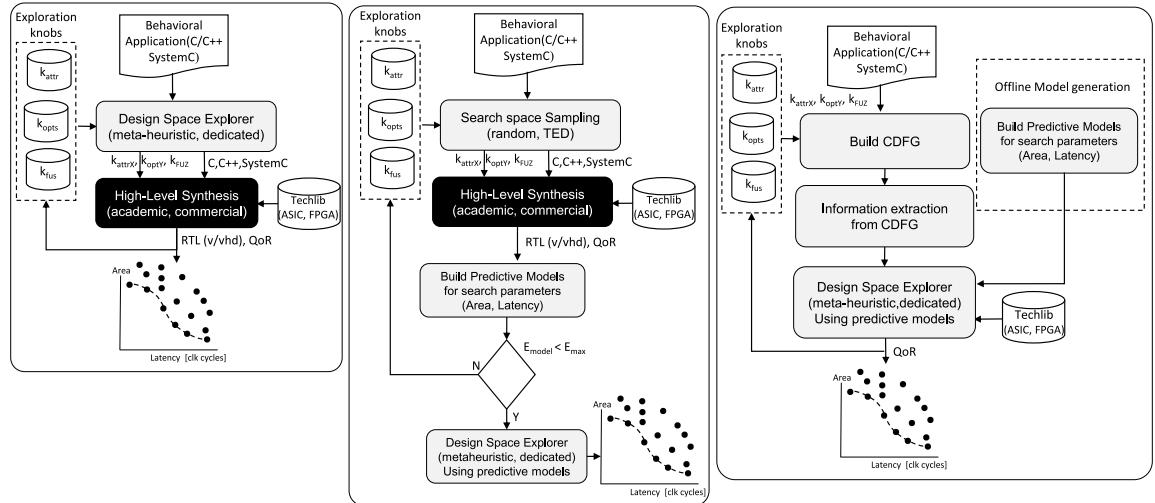


Figura 5.3: Flussi di DSE: (a) synthesis based, (b) supervised learning based, (c) graph analysis based [23]

Metodi synthesis based

I metodi basati sulla sintesi generano delle nuove impostazioni degli knob e invocano il tool di HLS per valutarne l'effetto sulla QoR. Questa categoria può essere ulteriormente suddivisa in metaeuristiche - di cui fanno parte, ad esempio, gli algoritmi genetici - ed euristica dedicata, su cui il campo della ricerca si sta concentrando maggiormente.

La Fig. 5.3(a) mostra un tipico flusso di DSE basato sulla sintesi. Per prima cosa tale metodo di esplorazione analizza la descrizione comportamentale da esplorare ed estrae alcune informazioni, come le operazioni presenti nel codice che possono essere controllate tramite pragma. Quindi genera una combinazione unica dei diversi knob di esplorazione e con tale combinazione invoca il tool di HLS. A seconda della qualità dei risultati ottenuta dalla HLS, il processo viene reiterato, cambiando di volta in volta le

impostazioni degli knob. Il processo termina quando determinati criteri impostati dall'utente vengono soddisfatti.

Metodi supervised learning based

Questa categoria - metodi di apprendimento supervisionato - è una combinazione dei metodi basati sulla sintesi e di quelli basati sul modello. La Fig. 5.3(b) mostra il flusso tipico dell'esplorazione con apprendimento supervisionato. Inizialmente viene effettuata un'analisi della descrizione comportamentale da esplorare e - analogamente ai metodi basati sulla sintesi - vengono estratte le operazioni che possono essere controllate tramite pragma.

Dal momento che la parte più dispendiosa del processo di DSE consiste nell'invocazione dei tool di HLS, queste tecniche usano un metodo di apprendimento supervisionato per generare modelli predittivi. Per generare questi modelli lo spazio di ricerca viene campionato; ciò consente di sintetizzare un certo numero di progetti e di costruire continuamente nuovi modelli predittivi finché l'errore tra il modello e i progetti sintetizzati si trova al di sotto di una certa soglia massima definita dall'utente. Non appena l'errore è minore, l'esplorazione continua utilizzando i modelli predittivi in combinazione con una ricerca euristica, un metodo non rigoroso che consente di ottenere un risultato che dovrà poi essere convalidato, per trovare i progetti ottimali. Questo approccio riesce quindi a ovviare al problema di dover eseguire l'intero processo di HLS per ogni nuova impostazione degli knob, come invece avviene per i metodi synthesis based.

Metodi model based

L'ultima categoria che è stata recentemente introdotta si basa su tecniche di analisi di grafi statici.

Come si può riscontrare dalla Figura 5.3(c), tale metodo non richiede nemmeno l'invocazione del tool di HLS. Questi metodi si basano sulla costruzione di modelli predittivi e sul loro utilizzo attraverso tecniche di analisi del control and data flow graph (CDFG). In primo luogo, viene effettuata un'analisi della descrizione comportamentale da esplorare e poi viene costruito un CDFG, da cui verranno estratte informazioni, ad esempio relative ai loop e al numero di iterazioni.

Sulla base di queste informazioni e dei modelli predittivi, viene fatta una stima dell'area, della latenza, ecc. del dato CDFG. Questi metodi realizzano delle trasformazioni del CDFG che imitano le trasformazioni determinate dai pragma dell'HLS e continuano poi l'esplorazione.

5.2 Verifica

La progettazione è solo un aspetto dell'architettura dei sistemi. Un determinato progetto infatti ha anche bisogno di essere verificato.

In generale, la verifica di un progetto consiste nel dimostrare che il sistema sintetizzato presenta il comportamento richiesto dalle specifiche, effettuandone quindi una verifica funzionale. Questo controllo deve essere effettuato tramite una simulazione o altre tecniche sia sulle specifiche iniziali, sia sul progetto finale con lo stesso insieme di input, verificando infine che gli output corrispondano, dimostrandone così l'equivalenza formale [8].

L'industria stima che più del 50% del tempo richiesto per la progettazione e lo sviluppo di chip è attualmente speso in verifiche [21]; per questo motivo, anche nell'ambito della verifica è nata l'esigenza di trovare delle tecniche per renderla più rapida e ottimizzarla. Effettuare un passaggio a livelli di astrazione più elevati è ancora una volta la soluzione che viene in soccorso ai progettisti.

Grazie all'high-level synthesis, i progettisti non devono aspettare che i modelli RTL diventino disponibili per poter condurre la verifica, ma possono invece sviluppare, eseguire il debug e verificare un progetto dal punto di vista funzionale in una fase preliminare della progettazione.

Come è cambiata la procedura di verifica nel passaggio dalla progettazione RTL a quella ad alto livello? In un primo momento le stesse tecniche adottate per la verifica e il debug del codice RTL sono state trasferite al codice ad alto livello: oggi infatti anche codice scritto in linguaggi ad alto livello come C/C++ può essere simulato, emulato e prototipato mediante piattaforma FPGA proprio come avviene

per il codice RTL.

La tecnica più diffusa ereditata dal codice RTL è la simulazione: grazie a un testbench è possibile fornire in ingresso al DUT (device under test) dei vettori di test. I risultati del testbench potranno essere valutati in termini di performance e copertura del codice. Valutare la copertura del codice significa valutarne il grado di esecuzione e quindi la qualità della simulazione; maggiore è la percentuale di copertura del codice, minore sarà la probabilità che siano presenti dei bug software.

Un'altra tecnica di verifica funzionale, retaggio della verifica RTL, in genere eseguita in seguito alla simulazione, è l'emulazione, che consiste nell'esecuzione del codice sorgente su un dispositivo hardware diverso da quello di destinazione.

L'ultima tecnica acquisita dalla verifica dell'RTL è la prototipazione su FPGA, che a differenza della simulazione via software, consente di accelerare l'algoritmo da provare alla velocità per cui è stato pensato. Questa possibilità è particolarmente utile per sviluppare e validare complesse decisioni di progetto e ha inoltre il vantaggio di essere più economica rispetto all'emulazione.

Il passaggio ad alti livelli di astrazione consente di ridurre notevolmente lo sforzo di verifica:

1. in primo luogo è più facile tracciare, identificare e correggere i bug a livelli di astrazione più elevati con descrizioni di progetto più compatte e leggibili;
2. in secondo luogo la simulazione ad alto livello è tipicamente molti ordini di grandezza più veloce della simulazione RTL, consentendo test più completi e una maggiore copertura;
3. infine, nonostante le simulazioni delle specifiche in C++/SystemC non esentino dalla verifica del codice RTL, sono comunque vantaggiose in quanto determinano una riduzione dei cicli di verifica e di debug che di norma vengono effettuati sull'RTL generato manualmente.

Da un sondaggio tra gli utenti dell'HLS, è emerso che grazie ad essa il tempo di verifica è stato più che dimezzato [25]. La simulazione delle specifiche in C++/SystemC ha dunque molti vantaggi, ma, come già accennato, non esime dal compiere la verifica dell'RTL, un'operazione molto lunga e onerosa, che richiede la scrittura manuale di un testbench RTL.

L'high-level synthesis potrebbe inoltre sollevare un altro problema nella fase di verifica, sottolineato in [26], quello della ridondanza del codice. Le ridondanze possono essere presenti nelle specifiche del codice sorgente, ma anche introdotte inavvertitamente dal tool di HLS durante la generazione dell'RTL. Gli sviluppatori di tool dovrebbero cercare di eliminare la tendenza a generare logica ridondante. La ridondanza logica infatti introduce punti che potrebbero non essere coperti con alcun vettore di test, abbassando così la copertura RTL. E' compito dei verification engineer individuare quali punti non sono coperti ed eventualmente riscrivere l'RTL.

Rispetto alla copertura line/branch utilizzata nei tradizionali linguaggi di programmazione, la copertura RTL richiede dei controlli aggiuntivi per esercitare le funzioni logiche. Ogni termine nell'espressione viene posto alternativamente a true o a false, mentre i valori dei termini rimanenti sono tenuti fissi in modo tale da individuare come un determinato termine influenza l'output dell'espressione. In generale, tali operazioni richiedono molto tempo ed energia, risultando spesso tediose.

Quali strategie si possono adottare per ovviare a questi inconvenienti e sfruttare al meglio la rapidità della verifica funzionale ad alto livello, facilitando anche la verifica del codice RTL generato ed individuando le ridondanze?

La ricerca ha provveduto a elaborare delle nuove tecniche che si prefiggono lo scopo di agevolare la verifica del codice RTL attraverso la co-simulazione, capace di combinare l'operazione di simulazione dell'input e dell'output dell'HLS, e la verifica formale, volta a dimostrare l'equivalenza tra il codice ad alto livello e il codice RTL e a permettere di individuare la ridondanza logica.

Per comprendere le nuove strategie di verifica proposte dalla ricerca, oggi già adottate da alcuni produttori di EDA, ma ancora in via di sviluppo, analizziamo nel dettaglio il processo di verifica messo in atto nei tool di HLS.

La Figura 5.4 mostra un tipico framework di simulazione e verifica dei tool HLS basati sul linguaggio C/C++. A partire da specifiche ad alto livello, scritte generalmente in C/C++, i progettisti impiegano

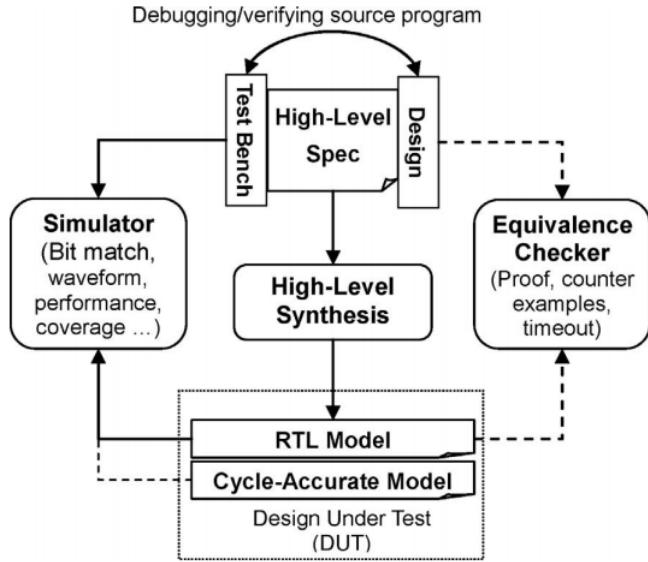


Figura 5.4: Framework di simulazione e verifica HLS [27]

tool di programmazione e di debug del software per garantire che il progetto sia sufficientemente testato e verificato su un testbench adeguatamente costruito. Una volta che la descrizione dell'input per HLS è stata verificata dal punto di vista funzionale, i progettisti possono procedere con la sintesi vera e propria, generando una o più versioni di codice RTL per esplorare i compromessi in termini di prestazioni, area e potenza sotto diversi vincoli di progetto. A questo punto anche l'RTL dovrebbe essere verificato per completare il processo di verifica. Invece che eseguire la verifica RTL con le tecniche tradizionali, è possibile confermare la correttezza dell'RTL generato grazie a delle nuove tecniche recentemente introdotte. Tali tecniche sono la co-simulazione automatica e il controllo formale di equivalenza.

Co-simulazione automatica

Come accennato, la simulazione è ancora la tecnica prevalente per controllare se l'RTL risultante è conforme alle specifiche di alto livello. Per ridurre lo sforzo speso per la simulazione RTL, le ultime tecnologie HLS hanno apportato importanti miglioramenti sulla co-simulazione automatica consentendo il diretto riutilizzo del framework di test originale in C/C++ per verificare la correttezza dell'RTL sintetizzato. La Figura 5.5 mostra come nel tool AutoPilot, antenato dell'attuale Vivado HLS, di cui

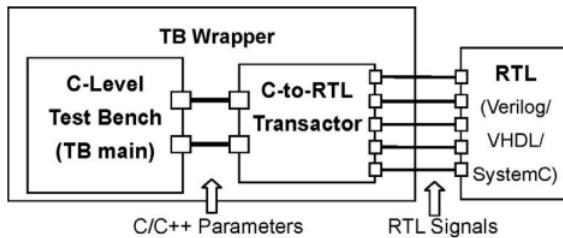


Figura 5.5: Generazione automatica di un testbench RTL in AutoPilot, antenato dell'attuale Vivado HLS [27]

si parlerà in seguito, il testbench comportamentale in C viene sottoposto a delle trasformazioni per ottenere automaticamente un testbench RTL. Attraverso la sintesi, i costrutti di interfaccia di alto livello (ad esempio parametri e variabili globali) vengono convertiti in segnali a livello di pin in RTL. Inoltre, grazie ai livelli di astrazione forniti da SystemC, una libreria di C++ di cui si discuterà più avanti, viene creato un involucro che comprende il testbench in C di partenza e le interfacce e i dati sintetizzati. A tale involucro viene inoltre aggiunta della logica di controllo per gestire la comunicazione tra il modulo

che viene testato e il DUT (device under test) in RTL, creando così il testbench RTL. In definitiva, la co-simulazione automatica permette ai progettisti di risparmiare moltissimo tempo, sollevandoli dalla dispendiosa creazione manuale di un testbench RTL.

Il framework di co-simulazione automatica può anche fornire prestazioni aggiuntive e analisi della copertura del codice sull'output RTL. Inoltre molti tool HLS generano anche modelli alternativi cycle-accurate (tipicamente in SystemC) del progetto sintetizzato, che possono essere simulati più rapidamente di codice in HDL [27].

Controllo dell'equivalenza

Il controllo formale dell'equivalenza consiste nel dimostrare l'equivalenza di due specifiche, input e output di un'operazione di sintesi.

Mentre i tool di controllo dell'equivalenza formale per confronti RTL-to-RTL e RTL-to-gate sono in produzione da anni, il controllo high-level-to-RTL è ancora una tecnologia in evoluzione.

Tuttavia negli ultimi anni, sono stati fatti progressi promettenti sul controllo dell'equivalenza C-RTL, soprattutto dall'industria.

Nel controllo formale dell'equivalenza RTL-to-gate viene dimostrato che lo stato futuro calcolato dalla logica a livello di gate e dalla logica RTL sono equivalenti. Le verifica formale applicata alla HLS si basa su principi simili di corrispondenza dello stato, dimostrando che i next-state dell'RTL generato e del codice in C++/SystemC sono equivalenti.

Tra le tecniche proposte per effettuare il controllo di equivalenza, molto promettente è il metodo sequenziale [28]. Questa tecnica di verifica formale dell'RTL generato rispetto al codice sorgente in C++ è un'area di sviluppo attivo nella ricerca: essa consentirebbe di ridurre alcuni dei requisiti di verifica basati sulla simulazione concentrandosi sulla verifica della correttezza dell'integrazione dei blocchi e dei sottosistemi nel SoC.

Attualmente le tradizionali metodologie di verifica dell'RTL generato dalla HLS - dalla simulazione, all'emulazione, alla prototipazione su FPGA - sono ancora molto diffuse, e il controllo formale di equivalenza, sebbene sia stato implementato sottoforma di tool da produttori di EDA come Synopsys e Calypto, copre solamente un ruolo di supporto nel flusso di verifica per l'HLS [27].

5.3 Riutilizzo dei progetti

Il livello di astrazione più elevato è quello che permette più facilmente di riutilizzare una specifica. Grazie alla high-level synthesis, un progetto può essere riutilizzato per ottenere delle diverse performance o per essere adattato a una diversa tecnologia.

Per effettuare questa operazione, talvolta è necessario apportare delle modifiche al codice sorgente. Ad esempio se la tecnologia impiegata nel nuovo progetto presenta degli obiettivi differenti in termini di performance, potrebbe essere necessario implementare una diversa architettura della memoria. Questo è proprio ciò che avviene nel passaggio da ASIC a FPGA e viceversa. I progettisti ASIC infatti necessitano di avere molta flessibilità e controllo durante la generazione del loro codice RTL. Per questo motivo, i tool commerciali di high-level synthesis forniscono un'ampia varietà di opzioni di sintesi in termini di esplorazione dello spazio di progettazione. Non è questo il caso di progetti che hanno come target una piattaforma FPGA, dove lo spazio di progettazione è molto più circoscritto e i parametri da ottimizzare messi a disposizione dai tool di sintesi sono in numero minore rispetto a quelli per gli ASIC [23]. Per ovviare a questo problema, in un paper del 2019 [29] è stato proposto un framework di conversazione basato sul machine learning che converte i risultati dell'esplorazione per passare da una tecnologia target a un'altra differente automaticamente.

Altre volte il codice di un progetto può essere riutilizzato per realizzare una versione migliorata del progetto di partenza, ad esempio includendo funzionalità aggiuntive.

Un'altra strategia per riutilizzare delle porzioni di un progetto consiste nella creazione di IP (intellectual property) comportamentali; il linguaggio ad alto livello (in particolare il linguaggio C++) è ideale per la

creazione di IP, che vengono definiti attraverso dei parametri, che possono essere utilizzati per selezionare un dato algoritmo o un'architettura di memoria.

A differenza degli IP RTL che hanno una microarchitettura definita e protocolli di interfaccia, gli IP comportamentali possono essere riadattati a diverse tecnologie di implementazione o requisiti di sistema, ponendosi quindi come un'importante risorsa per aumentare la produttività.

Attualmente, la maggior parte degli IP HLS viene prodotta dagli utenti per il loro uso interno all'azienda; alcuni di essi sono forniti come parte del tool di HLS come progetti di riferimento per facilitarne l'adozione. Ai fini della commercializzazione di tali IP HLS, sarebbe opportuno poterli crittografare in modo tale da proteggere l'investimento e poterlo sviluppare, fornendo tuttavia informazioni sufficienti sul prodotto affinché l'utente possa sceglierlo con criterio e impiegarlo al meglio nei propri progetti [25]. Per poter riutilizzare anche i vecchi progetti RTL nel flusso di progettazione dell'high-level synthesis, nel campo della ricerca si sta diffondendo l'idea di sviluppare un compilatore capace di convertire il codice RTL in C, con l'obiettivo di raccogliere l'eredità dei vecchi progetti, convertendoli e rendendoli riutilizzabili. Esistono già alcuni compilatori che assolvono questa funzione, ma sono principalmente finalizzati alla creazione di modelli di simulazione rapida [30].

5.4 Il ruolo chiave degli FPGA

In aggiunta ai vantaggi dell'adozione di metodologie di sintesi ad alto livello, analizziamo i punti di forza dell'utilizzo dell'high-level synthesis specificamente per FPGA, sempre più utilizzati, ancor più degli ASIC, su cui i primi tool di HLS si erano focalizzati.

- Gli FPGA consentono ai progettisti di effettuare velocemente la progettazione; essendo dispositivi programmabili, possono essere ri-progettati se necessario, senza costi di manifattura aggiuntivi. Permettono inoltre di effettuare simulazioni in-system con un buon livello di copertura del codice. Si tratta di un grande vantaggio, soprattutto per quanto concerne le verifiche formali, che risultano essere particolarmente ostiche nel campo degli ASIC. In questo settore infatti i costi per la manifattura nelle tecnologie dei circuiti integrati (IC) sono elevatissimi. A causa dei consistenti investimenti effettuati dall'industria per lo sviluppo di ASIC, i progettisti sono sottoposti a molte pressioni. Inoltre i tool di verifica per l'HLS non sono ancora maturi e la copertura di simulazione può essere limitata per progetti SoC con milioni di gate. Questo costituisce un ostacolo non trascurabile per l'adozione dell'HLS negli ASIC, a vantaggio delle piattaforme FPGA.

- Gli FPGA si compongono di blocchi logici riconfigurabili, che possono essere sviluppati mediante HLS, e di un'infrastruttura di sistema, costituita da linee di interconnessione e blocchi di I/O che possono essere implementati con degli IP (intellectual property) predefiniti. In generale gli IP sono delle unità che svolgono funzioni specifiche, implementando funzioni aritmetiche, memorie, processori e bus embedded. Ad esempio, interfacce di I/O o interfacce di comunicazione tra varie risorse del dispositivo possono essere brevettate come IP. Ciascuno di questi IP è implementato a livello RTL e viene messo a disposizione del progettista come un core encapsulato.

Durante la progettazione di piattaforme FPGA, le caratteristiche di questi blocchi predefiniti devono essere conciliate con i vincoli di progetto della piattaforma che si intende implementare. Grazie alla presenza di tali blocchi, ampiamente testati e ottimizzati in quanto IP, lo spazio di progettazione è circoscritto e il progetto può essere sviluppato velocemente, ottenendo risultati di buona qualità. Questa metodologia di progettazione, per cui gli FPGA sono particolarmente indicati, che consiste nell'integrazione di diversi blocchi - predefiniti e non - prende il nome di sintesi platform-based [31].

- Gli FPGA sono una risorsa importante per progetti che presentano un time-to-market critico, per evitare tempi troppo dilatati nella progettazione dei chip e nei cicli di manifattura. I progettisti potrebbero scendere a compromessi in termini di performance, potenza e costi, accorciando però notevolmente il tempo necessario alla progettazione oppure, con ulteriori sforzi, ottenere una QoR paragonabile all'RTL scritto manualmente.

- Gli FPGA sono richiesti per realizzare la compilazione e la sintesi di hardware per operazioni di calcolo accelerato e riconfigurabile. Recenti avanzamenti nel campo degli FPGA, infatti, hanno permesso alle piattaforme di calcolo riconfigurabile di accelerare molte applicazioni di calcolo ad alta performance, come processing di immagini e video, analisi finanziarie, bioinformatica e applicazioni di calcolo scientifico.

Alla luce di quanto sopradetto, un numero sempre più elevato di progetti realizzati mediante FPGA sono prodotti tramite tool di HLS.

Molti tool, oltre a generare l'RTL, creano anche dei report di sintesi che stimano l'utilizzo delle risorse FPGA, così come il timing, la latenza e il throughput del progetto sintetizzato [27]. I report includono una ripartizione delle prestazioni e metriche dell'area per singoli moduli, funzioni e loop nel codice sorgente. Ciò consente agli utenti di identificare rapidamente aree specifiche per il miglioramento della QoR e quindi regolare le direttive di sintesi o perfezionare di conseguenza la progettazione del codice sorgente. Infine, i file HDL generati e i vincoli di progettazione vengono dati in ingresso a tool di sintesi logica per l'implementazione su hardware. Esistono delle vere e proprie toolchain ISE (integrated synthesis environment) e degli EDK (Embedded Development Kit) che forniscono degli ambienti di sviluppo per poter trasformare l'implementazione RTL in un'implementazione completa appositamente per FPGA sotto forma di bitstream per la programmazione della piattaforma FPGA di destinazione.

Capitolo 6

Linguaggi

6.1 Qual è il linguaggio più adatto per l'input dell'high-level synthesis?

La scelta del linguaggio è stata una questione piuttosto spinosa nei decenni passati nel campo della sintesi ad alto livello.

Uno dei primi tool pionieri della HLS, Carnegie-Mellon University design automation (CMU-DA), è stato sviluppato dai ricercatori della Carnegie Mellon University negli anni '70. In questo tool, il progetto veniva specificato mediante il linguaggio instruction set processor specification (ISPS) [32], un computer description language, che supportava numerosi domini di applicazione e vari livelli di progetto, fornendo delle estensioni per specificare informazioni dipendenti dall'applicazione. Da un lato ISPS poteva essere visto come un linguaggio di programmazione, ma lo scopo della notazione era quello di descrivere il computer e altri sistemi digitali, non necessariamente generali algoritmi di calcolo. Con il tool CMU-DA inizia quindi a profilarsi l'idea di un tool che accetti in ingresso un linguaggio con caratteristiche affini ai linguaggi di programmazione, ma con delle estensioni specificamente hardware. Successivamente, negli anni '80 e nei primi anni '90, sono stati stati sviluppati diversi tool di HLS simili, principalmente nel campo della ricerca. Tali tool impiegavano dei linguaggi ideati ad hoc per definire le specifiche di progetto.

La maggior parte dei linguaggi adottati da tali tool erano linguaggi di descrizione dell'hardware (HDL), orientati alla simulazione e alla documentazione dell'hardware. Oltre a ISPS, si ricordano ADLIB/SABLE, ESIM e VHSIC.

HardwareC [33] del tool Hercules faceva eccezione: pur trattandosi anch'esso di un linguaggio di descrizione dell'hardware, aveva tuttavia la peculiarità di essere stato progettato appositamente per la sintesi. Come suggerisce il nome, si basava sulla sintassi del linguaggio di programmazione C. Tuttavia presentava una propria semantica hardware e differiva dal linguaggio C sotto molti aspetti, fornendo funzionalità aggiuntive per aumentare la potenza espressiva del linguaggio e per facilitare la descrizione dell'hardware. Ad esempio i progetti potevano essere descritti in HardwareC sia come sequenza di operazioni, sia come interconnessione strutturale di componenti, fornendo quindi supporto sia per una semantica procedurale, sia dichiarativa; presentava inoltre meccanismi incorporati per supportare i vincoli di progettazione e le specifiche di interfaccia.

Un altro linguaggio degno di nota fu Silage, impiegato nel tool Cathedral/Cathedral-II, specificamente progettato per la sintesi di hardware per l'elaborazione di segnali digitali. Presentava inoltre supporto integrato per tipi di dato personalizzati e consentiva facili trasformazioni.

Come discusso nel Capitolo 3, negli anni '90 si assiste all'affermazione e alla diffusione della sintesi RTL grazie allo sviluppo di numerosi tool; parallelamente la qualità dei tool di HLS volge verso un miglioramento e le tecniche di sintesi ad alto livello iniziano a espandersi dalla ricerca all'industria. I principali vendori di EDA iniziano a proporre i loro primi tool di HLS commerciali, capaci di generare implementazioni RTL a partire da linguaggi HDL comportamentali, completamente o solo parzialmente temporizzati, come quelli sopra menzionati. Tuttavia questi nuovi tool non riuscirono a porsi come valida alternativa all'RTL. Presentavano infatti delle criticità: nonostante alcuni tentativi di rendere il

linguaggio dei tool di HLS più simile a linguaggi di programmazione ben noti, come nel caso di HardwareC, l'HDL espresso in termini comportamentali come linguaggio in input era riuscito ad alzare solo parzialmente il livello di astrazione, richiedendo notevoli sforzi nell'apprendimento dei nuovi linguaggi, sia ai progettisti software, sia a quelli hardware, vanificando così i tentativi di abbattimento del gap tra hardware e software che la sintesi ad alto livello si proponeva di attuare.

Alla luce delle problematiche emerse con i linguaggi adottati dalle prime generazioni di tool di HLS, tra gli sviluppatori di tool iniziò a crescere la consapevolezza che per potersi affermare come scelta ottimale, il linguaggio in input a un tool di HLS doveva soddisfare i seguenti requisiti:

- doveva essere un linguaggio già utilizzato nella stesura di algoritmi, per andare incontro alle conoscenze dei software engineers e accorciare la distanza tra il mondo software e il mondo hardware;
- doveva supportare un meccanismo di astrazione: se il linguaggio presenta un solo livello di astrazione infatti, non può essere applicato a una grande varietà di progetti hardware;
- infine doveva supportare specificamente un livello di astrazione hardware, fornendo un collegamento tra il codice sorgente ad alto livello e le implementazioni di basso livello; in altri termini doveva supportare una rappresentazione che mappasse direttamente in RTL e provvedesse a una gerarchia dei livelli di astrazione fino all'RTL.

Quest'ultimo requisito richiede la presenza di elementi ben precisi: una gerarchia, la bit-accuracy, la possibilità di esprimere la concorrenza e informazioni relative al timing.

In assenza di questi ingredienti, un linguaggio non può rappresentare tutti gli aspetti necessari alla progettazione hardware [34].

6.2 L'avvento del linguaggio C/C++

Il linguaggio C/C++ sembrò essere il linguaggio più adatto ad assolvere questa funzione; si trattava infatti di un linguaggio ben noto agli ingegneri del software e in virtù di ciò avrebbe potuto agevolare la co-progettazione dell'hardware e del software di sistemi digitali.

Tuttavia esso non soddisfaceva tutti i requisiti richiesti a un linguaggio per la sintesi di hardware e in più presentava alcune caratteristiche che sembravano non essere compatibili con lo scopo che si proponeva di realizzare:

- in primo luogo non era dotato di costrutti specifici legati al timing, alla gerarchia dell'hardware, alle porte di interfaccia, alla definizione dell'esatta lunghezza in bit di un segnale, al parallelismo, ... [18].
- presentava inoltre costrutti complessi, come i puntatori, funzionalità come gestione della memoria dinamica, ricorsioni e polimorfismo, che non avevano delle controparti in hardware, portando quindi a grandi difficoltà implementative [35].

Tra la fine degli anni '90 e i primi anni 2000, per ovviare alle mancanze e per risolvere le problematicità del linguaggio C, vennero introdotte rispettivamente estensioni e restrizioni del linguaggio C per renderlo più conforme alla sintesi di hardware.

Le scelte non furono univoche: ogni produttore di EDA (electronic design automation) adottò soluzioni differenti, ma nessuna di esse riuscì a emergere sulle altre come era successo per il VHDL e il Verilog nell'ambito della progettazione register-transfer level [36].

Tuttavia, nel panorama variegato dei tool che si svilupparono in questo periodo, si possono individuare due correnti: l'approccio library-based e quello language-based.

Per illustrarne la differenza, si analizzeranno brevemente SystemC e SpecC, presi come riferimento rispettivamente per l'approccio library-based e per quello language-based.

SystemC è una libreria standard di classi C++ che fornisce funzionalità di modellazione dell'hardware, grazie a cui è possibile scambiare liberamente degli IP in C++.

Gli approcci basati su libreria, prevedono la trasformazione del codice in C/C++ in un linguaggio di descrizione dell'hardware; questa caratteristica consente di gettare un ponte tra software e hardware engineers, che grazie a queste librerie di descrizione dell'hardware possono "parlare" lo stesso linguaggio. Inizialmente c'erano alcune lacune semantiche, legate cioè al significato delle istruzioni del linguaggio: la mappatura delle specifiche comportamentali in C++, espresse in forma algoritmica, nell'architettura del sistema non era un compito semplice e, in un momento in cui i tool di sintesi non avevano ancora raggiunto una adeguata maturazione, sembrava che la sola aggiunta di una libreria non sarebbe bastata a esprimere la complessità dei dispositivi hardware.

Per questo motivo nello stesso periodo si è delineata un'altra strategia che non si limitava allo sviluppo di librerie, ma proponeva delle estensioni del linguaggio che facevano uso di una semantica nuova, realizzando nuovi veri e propri linguaggi C-based.

Tra questi vi era appunto SpecC, un system description language, superset di C. Lo SpecC Technology Open Consortium (STOC), sostenuto dalle aziende di elettronica del Giappone, venne fondato nel 1999 per promuovere l'adozione di SpecC come linguaggio di specifica per la progettazione ad alto livello.

Analogamente all'approccio basato su libreria, quello basato su linguaggio adottato da SpecC, prende piede dallo stesso intento di passare dai linguaggi HDL ai linguaggi C/C++ per la progettazione ad alto livello. Tuttavia SpecC adotta un approccio diverso: per lo sviluppo di tale linguaggio, in primo luogo sono stati identificati i requisiti per la progettazione dei SoC e in base ad essi sono stati ideati dei costrutti creati ad hoc con una semantica ben definita che estendono il linguaggio C. Quando una specifica in SpecC viene simulata i nuovi costrutti vengono espansi in una serie di chiamate API (application program interface) di simulazione, che sono approssimativamente equivalenti all'API definita dalla libreria open source SystemC, con la differenza che l'utente interagisce con i costrutti senza avere accesso alla libreria, che è nascosta [37].

Come SpecC, si sono sviluppati una pletora di dialetti diversi, come ImpulseC, Handel-C, BachC, SiliconC, più simili ad hardware description language che non a linguaggi di programmazione.

Molti di questi sono caduti in disuso, con poche eccezioni (come Handel-C, ancora in uso in ambito accademico), a vantaggio dei tool che supportano il puro linguaggio C/C++, con l'ausilio della libreria SystemC e con la possibilità di inserire delle direttive per guidare il processo di high-level synthesis. Di questi ultimi si discuterà nel prossimo paragrafo.

6.2.1 Estensioni e limitazioni

Di seguito, verranno presentate brevemente alcune delle estensioni e delle limitazioni del linguaggio C/C++ che sono state adottate nei tool maggiormente diffusi oggi. In merito alla gestione di certi costrutti e tipi di dato (ad esempio i puntatori), ciascun tool ha optato per soluzioni differenti. Nella difficoltà di impostare un discorso generale, è stato scelto un tool in particolare, Vivado HLS, molte delle cui soluzioni vengono impiegate anche da altri tool di HLS.

La ragione di questa scelta è da ricercarsi nel fatto che, da dieci anni a questa parte, Vivado HLS - un tempo AutoPilot - può essere considerato emblematicamente come lo stato dell'arte dei tool di high-level synthesis [27].

Il sottoinsieme di costrutti messi a disposizione dai primi tool di sintesi che avevano adottato il linguaggio C/C++ era molto limitato: in genere includeva i tipi di dato nativi integer (char, short e int), array a 1 dimensione, if-then-else, cicli for. Tale copertura del linguaggio è tutt'altro che sufficiente per consentire la progettazione di sistemi complessi.

Vivado HLS, di cui si discuterà ampiamente nel Capitolo 7, accetta tre voci di progettazione standard: C, C++ e SystemC.

Esso fornisce dei solidi strumenti di sintesi (da librerie a pragma e direttive) che consentono di gestire in modo efficiente diversi aspetti del linguaggio C/C++ per la sintesi di diversi tipi di dato (sia nativi, sia hardware-oriented), di puntatori, della memoria, del controllo, di loop, della gerarchia modulare (per funzioni, classi e moduli concorrenti) e sintesi di interfacce (per parametri e variabili globali).

```

VoteDpr = 0;
count   = pDprCount[0];
for(d = 1; d < DprRange; d++) {
#pragma AP unroll complete
#pragma AP expression_balance
    if(pDprCount[d] > count) {
        count   = pDprCount[d];
        VoteDpr = d;
    }
}

```

Figura 6.1: Esempio di codice con pragma per il loop unrolling e per il bilanciamento delle espressioni per implementare il parallelismo [39]

In particolare, i progettisti possono controllare completamente la precisione dei dati nelle specifiche in C/C++. Vivado HLS ad esempio supporta tipi di dato floating-point a mezza precisione, a precisione singola e doppia.

Ha inoltre la capacità di simulare e sintetizzare tipi di dati a precisione arbitraria, sia interi (ap_int), sia a virgola fissa (ap_fixed). I tipi di dati a virgola fissa a precisione arbitraria (ap_fixed) supportano le operazioni algoritmiche più comuni.

Mediane queste estensioni, i progettisti possono valutare un tradeoff tra accuratezza e costi, modificando la risoluzione e la posizione del punto fisso.

Come molti altri tool di HLS diffusi oggi, Vivado HLS supporta anche la libreria SystemC. I progettisti pertanto possono utilizzare tipi di dato bit-accurate in SystemC (sc_int/sc_uint, sc_bigint/sc_biguint, e sc_fixed/sc_ufixed) per definire le precisioni dei dati. I tipi di dato in SystemC permettono anche di creare progetti gerarchici paralleli con processi concorrenti in esecuzione all'interno di più moduli [27]. In Figura 6.2, viene mostrato il flusso di progettazione di Vivado HLS. Vivado HLS presenta vari input: accanto alla funzione in C/C++/SystemC, input primario del tool Vivado HLS, vi è inoltre la possibilità di specificare vincoli (constraints) e direttive (directives) [38]. In Vivado HLS, i vincoli sono obbligatori e includono il periodo, l'incertezza del clock e la tecnologia FPGA che si intende implementare.

Le direttive sono invece opzionali e dirigono il processo di sintesi per implementare un comportamento o un'ottimizzazione specifici. Esse consentono di dare indicazioni specificamente legate all'hardware (interfacce di comunicazione, chiamate di funzioni, cicli for, array di dati e una sezione di codice tra parentesi graffe).

In Vivado HLS, le direttive possono essere inserite sottoforma di #pragma direttamente nel codice sorgente (si veda l'esempio in Figura 6.1) oppure come comando in Tcl (Tool command language) in uno script in formato .Tcl [38].

Script differenti - e quindi diverse ottimizzazioni - possono essere utilizzati sullo stesso codice sorgente, consentendo così un'agevole esplorazione dello spazio di progettazione. I pragma invece sono incorporati nel codice sorgente e vengono applicati automaticamente a ogni soluzione quando viene risintetizzata. Possono quindi essere realizzati su un unico file, definendo così degli IP che possono essere utilizzati da terzi. Le direttive sottoforma di script, invece, devono essere sempre allegate al codice sorgente per poter ricreare lo stesso progetto.

In generale, pragma e direttive guidano il processo di high-level synthesis, realizzando delle ottimizzazioni volte a ridurre la latenza, a migliorare il throughput e a ridurre l'area e l'utilizzo delle risorse [39].

Dal momento che le direttive sono gestite dal precompilatore, il vantaggio di questo approccio è che il programma in input può essere compilato utilizzando il linguaggio C/C++ senza modifiche, in modo che un tale programma o un suo modulo possa essere facilmente gestito tra software e hardware e la co-simulazione di hardware e software possa essere eseguita senza la riscrittura del codice. Oggi l'infrastruttura di compilazione utilizzata dalla maggior parte dei tool è LLVM [4], un framework open source, che include un ambiente di progettazione completo e permette di generare codice intermedio target-independent a partire dalla descrizione comportamentale in linguaggi ad alto livello, semplificando il processo di HLS.

Il termine target-independent fa riferimento al fatto che la rappresentazione intermedia (IR) basata su

LLVM è in un formato indipendente dal linguaggio. In altre parole, il codice può essere ottimizzato senza considerare il linguaggio di partenza. Di conseguenza, lo stesso insieme di analisi e ottimizzazioni effettuate su questa rappresentazione può essere condiviso e sfruttato da molti frontend di diversi linguaggi. Inoltre, a differenza di altri compilatori C/C++ convenzionali, che sono tipicamente progettati per ottimizzare i tipi di dato nativi di C/C++ (ovvero, char, short, int e long), le procedure di compilazione e trasformazione di LLVM e di Vivado HLS sono completamente bit-accurate. Questo è un vantaggio significativo per la sintesi di hardware in quanto la ridondanza e il parallelismo bit-level possono essere ben ottimizzati e ben sfruttati.

Passando invece alle restrizioni poste al linguaggio C, ci sono varie operazioni il cui uso, ad oggi, non è ancora consentito nei principali tool di HLS.

Tra queste c'è l'allocazione dinamica della memoria: la tecnologia su cui i progetti vengono implementati ha un insieme fissato di risorse e la creazione e la liberazione dinamica delle risorse di memoria non sono consentite.

In Vivado HLS, come in molti altri tool, sono presenti inoltre altre limitazioni legate ai puntatori e alle funzioni ricorsive [38].

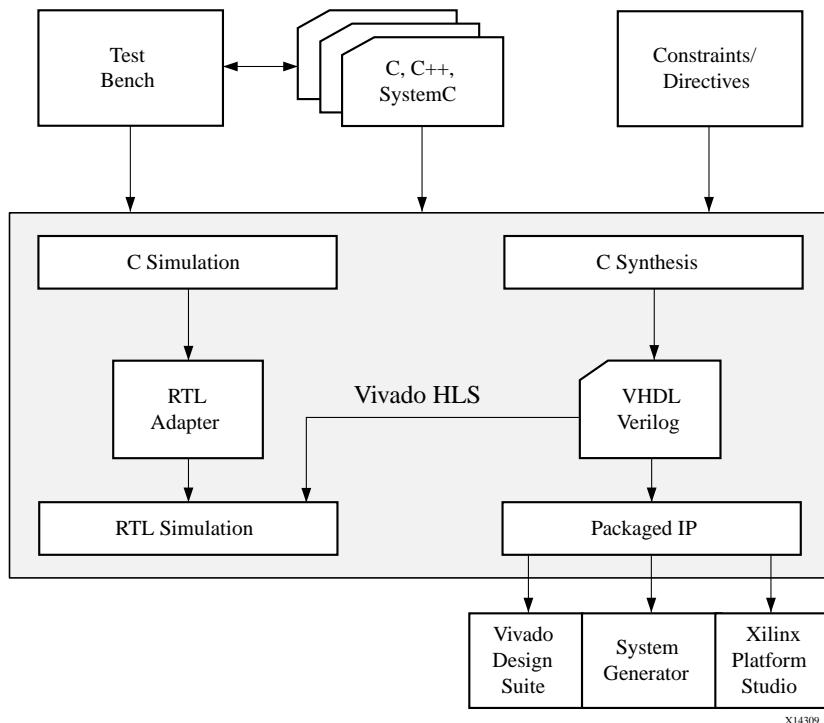


Figura 6.2: Flusso di progettazione di Vivado HLS [38]

Per approfondire: alcune estensioni hardware-oriented nel dettaglio

Le gerarchia

La gerarchia fornisce un modo per dividere il progetto in più parti e per gestire la concorrenza in maniera più esplicita, con l'intento di ridurre il tempo di esecuzione e di ottenere una migliore qualità di risultati, aiutando il progettista a interfacciarsi con un progetto di notevoli dimensioni.

Per ciascun blocco della gerarchia è possibile creare un numero di implementazioni alternative con interfacce diverse e successivamente utilizzarle come parte di un progetto più grande. Per ottenere una stima accurata del timing dell'implementazione di un blocco vengono utilizzati i tool di sintesi RTL.

La sintesi di interfacce

Il modo in cui la funzione in C++ comunica con il mondo esterno viene convertito in una descrizione

RTL attraverso la sintesi dell’interfaccia. Durante la sintesi, è possibile selezionare la migliore interfaccia per trasferire i dati al rate necessario a soddisfare una determinata performance.

L’importanza della bit-accuracy nella progettazione dell’hardware

I progetti hardware sono bit-accurate, ciò significa che ciascuna variabile è dimensionata in base a dei ben precisi requisiti in lunghezza (in bit). A differenza di C/C++, dove ogni variabile intera ha una lunghezza di 32 bit, nell’hardware le variabili presentano delle dimensioni arbitrarie. Affinchè la HLS possa supportare questa possibilità, il linguaggio C/C++ deve essere esteso e fornire all’utente la possibilità di creare dei tipi di dato che consentano di scegliere la lunghezza delle variabili. Quindi le variabili floating-point hanno bisogno di essere sostituite da variabili fixed-point o floating-point con una ben precisa bit-accuracy, quella minima per realizzare dell’hardware efficiente e al contempo capace di preservare le performance numeriche dell’algoritmo. Ad esempio, il pacchetto AC Datatype in C++ [40] fornisce integer, fixed-point, floating-point e tipi di dato complessi per facilitare il perfezionamento numerico mentre viene mantenuto il livello di astrazione della descrizione.

Il vantaggio dei tipi di dato a precisione arbitraria è che consentono di aggiornare il codice in C affinchè si faccia uso di variabili con la più piccola bit-width. Se viene apportata una modifica in termini di bit-width, viene nuovamente eseguita una simulazione per verificare che la funzionalità del progetto sia rimasta identica. Tale verifica è molto più veloce di una verifica del perfezionamento numerico in delle specifiche RTL create a mano [25]. La più piccola bit-width si traduce in operatori hardware che saranno appunto più piccoli e più veloci. Ciò permetterà di allocare più logica nella tecnologia prescelta e di farla eseguire a frequenze di clock più elevate.

Per approfondire: lo standard SystemC (IEEE 1666)

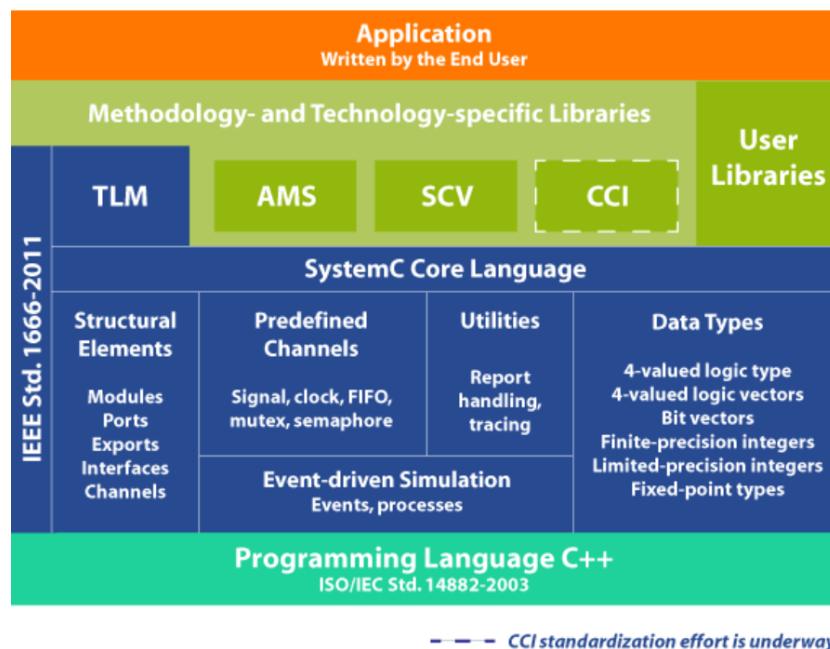


Figura 6.3: Architettura del linguaggio SystemC [41]

Negli ultimi anni, SystemC ha avuto sempre maggiore prominenza, al punto tale da essere standardizzato e da essere accettato in input da numerosi tool, affiancando i linguaggi C/C++.

Analizziamo più da vicino alcune peculiarità di SystemC, cercando di comprenderne i vantaggi che hanno portato alla sua diffusione.

Da una collaborazione tra gli utenti HLS e gli sviluppatori di tool, con il supporto di Accelera System

Initiative, SystemC è diventato un vero e proprio standard IEEE (Institute of Electrical and Electronics Engineers) per la sintesi, approvato nel 2011, esteso poi nel 2016 [41]. Per questo motivo oggi i più diffusi tool di HLS supportano C++ e SystemC. SystemC, sebbene venga spesso considerato alla stregua di un linguaggio, è di fatto un'estensione di C++, ovvero una libreria di classi specifica, che consente di scrivere codice in C++, che raccoglie l'eredità dei linguaggi HDL, mettendo a disposizione costrutti orientati all'implementazione dell'hardware, che non fanno parte dello standard C++.

Lo standard di sintesi di Accellera [41] definisce tutti i costrutti C++ e SystemC che dovrebbero essere supportati dai tool di HLS. Ai fornitori di tool è comunque consentito estendere lo standard con il supporto per elementi al di fuori dei costrutti di base messi a disposizione.

Di seguito verrà fornito un elenco delle principali funzionalità aggiuntive fornite dallo standard SystemC:

- fornisce supporto per l'uso dei puntatori e funzionalità correlate, come gli array e funzioni virtuali molto utili alla progettazione dell'hardware;
- fornisce classi in C++ che aggiungono costrutti di modellazione dell'hardware, permettendo di definire chiaramente l'architettura mediante moduli e porte per specificare la struttura e i thread (ovvero i processi) e tramite segnali ed eventi per specificare la concorrenza; consente inoltre di realizzare un prototipo virtuale dell'hardware per effettuare l'analisi delle prestazioni e lo sviluppo del software.
- fornisce tipi di dato bit-accurate integer e fixed-point e vettori di bit e logici.
- permette di specificare la gerarchia dell'hardware (in altri termini, la concorrenza a livello di blocco) a seconda della struttura delle chiamate alle funzioni in C++; in altri termini, la gerarchia può essere dedotta dalla struttura secondo cui avviene la chiamata delle funzioni in C++.
- permette di definire esplicitamente le interfacce;
- i dettagli pin-accurate e cycle-accurate (ovvero con una precisione rispettivamente a livello di pin e di ciclo) dei protocolli sono incapsulati in modo tale che la lettura o la scrittura su un'interfaccia appaiano come una chiamata di funzione nel thread del processo, che è in ogni caso privo di informazioni sul timing. Incapsulare i protocolli aiuta a mantenere il livello di astrazione del progetto isolando il comportamento a livelli di astrazione più bassi in un insieme di interfacce di libreria [25].

SystemC sta riscuotendo molto successo nell'industria: anche se il flusso di HLS in C/C++ presenta già buoni risultati per l'implementazione di una singola funzione, il flusso di HLS di SystemC sembra essere più adatto per la costruzione di moduli e di sistemi complessi. In Figura 6.3 è rappresentata l'architettura di SystemC: essa si erge al di sopra del livello di astrazione del linguaggio C++, fornendo la possibilità di descrivere elementi strutturali, canali predefiniti, tipi di dato e di effettuare simulazioni event-driven. Dato che SystemC è una libreria di C++, di fatto la scelta tra SystemC e C++ non riguarda il linguaggio, ma piuttosto l'astrazione, ovvero la necessità e la misura in cui vengono utilizzate le funzionalità di modellazione dell'hardware che SystemC mette a disposizione.

6.2.2 Chi può trarne vantaggio?

La scelta di linguaggi ad alto livello come C/C++ per descrivere progetti hardware risulta essere vantaggiosa non soltanto per gli ingegneri progettisti di hardware - che hanno la possibilità di interfacciarsi con progetti di dimensioni notevolmente ridotte e con codici più compatti e leggibili - ma anche per i software e i system engineers.

In virtù della scelta di linguaggi ad alto livello, infatti, il lato hardware è reso più accessibile anche alle figure professionali che non hanno esperienza in tale campo. Mentre la progettazione RTL richiedeva

la conoscenza specifica dei componenti hardware e di hardware description language come il VHDL e il Verilog, ora i nuovi tool di HLS consentono di realizzare la maggior parte dei dettagli implementativi specifici dell'hardware mediante l'uso di linguaggi più familiari come C/C++, colmando così il divario tra la progettazione system-level e la progettazione RTL.

Grazie all'high-level synthesis quindi la progettazione dell'hardware può avere inizio direttamente dal codice scritto dall'algorithm designer, facendo in modo che il progetto non debba essere re-implementato dai progettisti hardware in un linguaggio diverso [10].

Oltre a essere un vantaggio per i software e i system engineers, si tratta anche di uno sbocco sempre più necessario in virtù della direzione intrapresa nel campo della progettazione digitale: i sistemi digitali si compongono di hardware e software embedded e la co-progettazione hardware/software richiede appunto una combinazione di conoscenze hardware e software per essere effettuata in maniera efficace. Utilizzare un linguaggio ad alto livello per descrivere l'hardware risulta essere una scelta ottimale per andare incontro a questa nuova frontiera della progettazione digitale.

L'HLS si pone quindi come un vero e proprio ponte tra il lato software e il lato hardware. E' importante però precisare che, per ottenere risultati ottimali, è necessario avere esperienza nel campo dell'hardware per l'impiego della HLS [42], [24].

6.2.3 Quali sono i requisiti che un progettista deve avere per approcciarsi all'HLS?

L'ingegnere che si approccia all'HLS deve comunque avere una conoscenza di base dei principi della progettazione digitale. A seconda del background, l'uso efficace dell'HLS presenta diverse curve di apprendimento.

Utilizzare un linguaggio come SystemC è assolutamente nelle corde dei software engineers, abituati a programmare in C++ o comunque in linguaggi object-oriented: agli occhi di un ingegnere del software, SystemC è uno di tanti framework event-driven con cui sono abituati a interfacciarsi, dove i thread vengono eseguiti contemporaneamente nella simulazione (in un tempo virtuale) e nell'hardware sintetizzato (in tempo reale).

Tuttavia una conoscenza esclusiva del mondo software, rivolta principalmente al funzionamento dei compilatori e alla generazione di codice, non è sufficiente ad approcciarsi alla progettazione di hardware: è importante riuscire a comprendere i parametri da cui dipendono le performance dell'hardware, dalla frequenza di clock, alle pipeline, all'initiation interval; tali conoscenze consentono di individuare le dipendenze presenti nei cicli e la contesa delle risorse. E' necessario inoltre avere una certa consapevolezza delle risorse dei dispositivi FPGA e della libreria degli ASIC: operazioni, array e altri costrutti in C++ sono mappati in tali risorse, generando l'architettura RTL, output del processo di sintesi. Questo significa che c'è bisogno di avere una conoscenza di base dei diversi elementi di memoria (RAM, ROM e memorie distribuite) e blocchi moltiplicatori su chip. Un'abilità come quella di stabilire una prima corrispondenza tra tali risorse e i costrutti in C/C++ aiuta non solo a dare le giuste direttive, ma anche nell'ordine più efficace [21]

E' inoltre importante capire i costrutti in C/C++ che non sono supportati da questi tool poichè non c'è modo di implementare tali costrutti in hardware. Per esempio, le già citate allocazione dinamica della memoria e le funzioni ricorsive non possono essere utilizzate durante la sintesi ad alto livello.

Un'indagine del 2019 [24] si propone proprio di testare l'attitudine dell'HLS ad avvicinare alla progettazione digitale persone poco avvezze al modo hardware. Nel caso di studio in esame, è stato selezionato un gruppo di 6 persone con un background prevalentemente software a cui è stata sottoposta l'implementazione di un progetto sia in HDL, sia in C/C++ tramite il tool HLS Catapult-C di Mentor Graphics. Il progetto consisteva nell'implementazione di un ben noto algoritmo per il calcolo della trasformata discreta del coseno, componente destinato a un encoder di codifica video ad alta efficienza (HEVC). Il loro bagaglio di esperienza comprendeva una moderata conoscenza della programmazione software - che si traduce in circa 1k-100k righe di codice in C o C++ elaborate in media in 15 mesi per lavoro o per hobby - e un'esperienza molto limitata in ambito hardware, con una media di 1k righe di codice in Verilog o VHDL nell'arco di 3 mesi. Uno dei partecipanti aveva seguito un tutorial di introduzione all'HLS, senza

HLS						
Person #	Area (LUTs)	Freq. (MHz)	Speed*	Speed/Area**	Started	
1	1 860	214	258	139		
2	3 161	101	588	186	x	
3	2 814	211	675	240		
4	2 273	167	972	427	x	
5	2 768	137	797	288		
6	2 463	211	750	305		
Avg.	2 557	174	673	264		
RTL						
1	4 000	145	197	49	x	
2	2 068	108	192	93		
3	1 292	308	458	355	x	
4	4 431	148	499	113		
5	2 066	137	122	59	x	
6	2 722	149	121	44	x	
Avg.	2 763	166	265	119		
HLS/RTL ratio						
1	0.47x	1.48x	1.31x	2.81x		
2	1.53x	0.94x	3.06x	2.00x		
3	2.18x	0.69x	1.47x	0.68x		
4	0.51x	1.13x	1.95x	3.80x		
5	1.34x	1.00x	6.55x	4.89x		
6	0.90x	1.42x	6.22x	6.87x		
Avg.	0.93x	1.05x	2.54x	2.22x		

*Mtcoeff/s **Mtcoeff/s/kLUTs

Figura 6.4: Area e performance dei progetti RTL e HLS [24]

Person #	HLS		RTL		HLS/RTL Quality Ratio
	Hours	Quality*/Hours	Hours	Quality*/Hours	
1	2	80	4	14	5.9x
2	4	44	9	11	4.1x
3	12	19	21	17	1.1x
4	9	47	18	6	7.6x
5	5	60	9	6	9.4x
6	20	15	26	2	9.0x
Avg.	9	44	14	9	6.2x

*Mtcoeffs/kLUTs

Figura 6.5: Produttività HLS e RTL [24]

però averla mai sperimentata. Prima di sottoporli al test, è stato chiesto loro di documentarsi sulla HLS e di svolgere dei semplici esercizi sia in HDL, sia mediante il tool di HLS.

La Figura 6.4 mostra i risultati ottenuti da ciascuno dei 6 partecipanti all'esperimento in termini di area e performance nei progetti RTL e HLS. In linea generale, l'area e la frequenza di clock dei progetti RTL e di quelli ottenuti tramite tool di HLS sono equiparabili. Anche se l'RTL ha comunque ottenuto il primato sia in termini di area, raggiungendo il minor impiego di risorse hardware, sia riuscendo a ottenere la massima frequenza di clock.

Tuttavia la velocità di esecuzione delle operazioni - espressa in milioni di coefficienti di trasformata al secondo usando i coefficienti di output, le latenze, il throughput e la frequenza - dei progetti ottenuti con l'HLS sono 2.5 volte più elevate dei progetti sviluppati con RTL manuale. Ciò è da imputarsi al fatto che nell'RTL manuale nessun utente è riuscito a fare uso di pipeline, determinando una riduzione del throughput; al contrario, nel progetto HLS tutte le persone sono riuscite a fare uso di pipeline e di loop unrolling, ottenendo così throughput molto più elevati rispetto all'RTL.

La Figura 6.5 riporta i valori di produttività per i due approcci HLS e RTL. Il tool di HLS ha permesso una migliore produttività, in media fino 6.0 volte quella dell'RTL.

Il tempo impiegato dai partecipanti nello sviluppo dei progetti è stato diviso in 5 categorie: progetta-

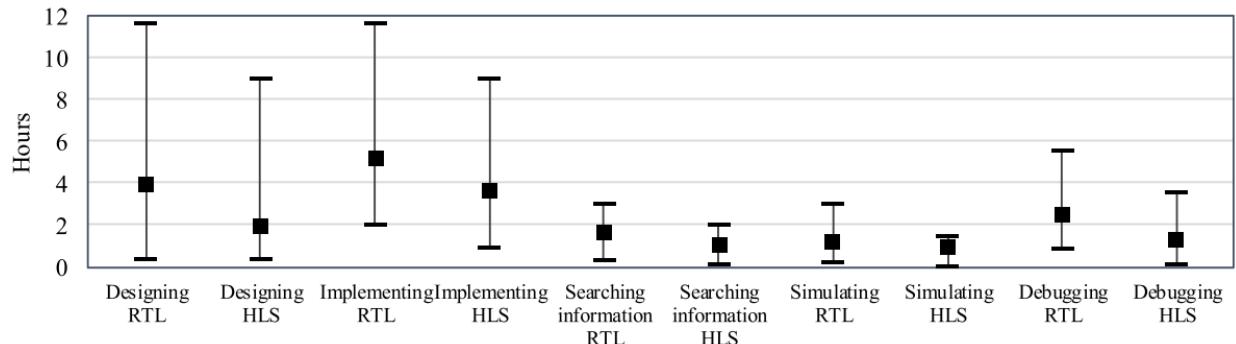


Figura 6.6: Tempo massimo, minimo e medio per ogni fase del processo di sintesi RTL e con tool di HLS [24]

zione, implementazione, formazione, simulazione e debugging. Come mostrato dalla Figura 6.6, mediamente, in ciascuna delle categorie, le persone hanno impiegato meno tempo lavorando con l'HLS. Il tempo massimo, medio e minimo impiegato complessivamente nei progetti è stato rispettivamente 37.7, 15.1 e 3.7 ore per l'RTL e 25.0, 10.1 e 1.6 ore con il tool di HLS.

In definitiva, questo studio - nonostante la ridotta dimensione del gruppo testato - mostra che è più facile adottare la HLS rispetto alla sintesi RTL e ottenere risultati migliori più velocemente per chi ha un background da progettista software.

Se per un software engineer è possibile colmare questo gap in modo rapido attraverso lo studio di esempi concreti che permettano di comprendere come funziona l'hardware, quali sono invece le competenze che devono essere acquisite da un progettista hardware per approcciarsi all'high-level synthesis?

Gli ingegneri hardware sono già in possesso di tutti i requisiti richiesti per utilizzare la HLS: generalmente conoscono il linguaggio C e hanno già abbastanza familiarità con linguaggi object-oriented dovendo fare uso di HDL come il VHDL e il Verilog. La sfida principale per queste figure professionali è probabilmente quella di esprimere le funzionalità a un livello di astrazione più elevato e affidare il processo di sintesi al compilatore. In altri termini, la difficoltà - più mentale che tecnica - sta proprio nell'abbandonare le familiari tecniche di sintesi RTL con descrizione del progetto ciclo per ciclo, per accostarsi alla nuova HLS.

Se per i software engineers la programmazione dell'hardware è qualcosa di nuovo e capace di destare la loro curiosità, per gli hardware engineers talvolta la HLS non è vista egualmente di buon grado: si presenta infatti come una sorta di "violazione" dei tradizionali flussi RTL a loro familiari.

Per questo motivo, l'introduzione della sintesi ad alto livello nella fase di progettazione dell'hardware a cura dei progettisti digitali è un processo graduale: certamente un hardware engineer non sarà disposto ad abbandonare di punto in bianco la sintesi RTL, ma è lecito aspettarsi che abbia la tendenza a integrare le due tecniche di sintesi, mettendo a confronto il codice RTL ottimizzato a mano con i risultati sintetizzati dalla HLS.

Ciò che i progettisti hardware possono riscontrare nell'attuale stato dell'arte dell'HLS è che essa non è ancora infallibile, quindi il progetto sintetizzato automaticamente mediante tool di HLS potrebbe presentare delle suboptimalità. Nonostante ciò, in virtù di tutti i vantaggi offerti dall'HLS, essa è una grande risorsa e - se usata sapientemente - è superiore alla progettazione RTL manuale, i cui risultati non sono in ogni caso esenti da suboptimalità [17].

6.3 L'impiego di altri linguaggi diversi da C/C++

Attualmente, la maggior parte dei tool di HLS commerciali supportano un linguaggio basato su C; tuttavia, nel 2010 esistevano diversi tool - molti dei quali sono stati abbandonati - che accettavano in input altri linguaggi [42]. Tra questi c'erano il MATLAB Compiler for Heterogeneous computing systems (MATCH), che accettava in input programmi in MATLAB, il MyHDL, un pacchetto Python open source,

il Sea Cucumber (SC), che supportava il linguaggio Java e molti altri.

Alcuni invece continuano ad essere utilizzati, come ad esempio KiwiC [43]. Si tratta di un compilatore basato su C#, che accetta gli output dei compilatori .NET o Mono C#, scritto in un linguaggio intermedio, e genera Verilog RTL.

Esistono anche altri compilatori HLS che utilizzano linguaggi diversi. Per esempio, Esterel è un linguaggio di programmazione sincrono progettato per programmare sistemi reattivi (sistemi che reagiscono continuamente al loro ambiente). Un altro esempio è il compilatore BlueSpec che funziona in base a Bluespec System Verilog – un linguaggio utilizzato nella progettazione di sistemi elettronici.

Oltre ai tool che accettano in input linguaggi di programmazione diversi da C, ci sono anche tool di HLS schematics-based per l'implementazione di FPGA, come ad esempio LabVIEW e MATLAB [15], [23].

Il modulo LabVIEW FPGA della National Instruments (NI) estende la piattaforma grafica di sviluppo LabVIEW per implementare degli FPGA su hardware I/O riconfigurabile della NI, riuscendo a rappresentare il parallelismo e il flusso dei dati.

HDL Coder invece genera codice Verilog e VHDL trasmissibile e sintetizzabile a partire da funzioni MATLAB, modelli Simulink e diagrammi Stateflow. Il codice HDL generato può essere utilizzato per la programmazione di FPGA o per la prototipazione e la progettazione di ASIC.

Anche le piattaforme di elaborazione parallela e i modelli di programmazione per le GPU sono un versante molto attivo della ricerca.

CUDA (Compute Unified Device Architecture) e OpenCL (Open Computing Language) sono modelli di programmazione parallela, nati inizialmente per la progettazione di GPU, ma che recentemente hanno espanso il loro dominio di applicazione anche ad altre tecnologie.

OpenCL (Open Computing Language) è un framework open source di programmazione parallela; esso consente di scrivere programmi che verranno eseguiti su piattaforme eterogenee costituite da CPU, GPU, DSP, FPGA e altri processori o acceleratori hardware.

E' possibile implementare codice OpenCL su tool di Intel, Xilinx e molte altre.

Capitolo 7

Tool

7.1 Tool accademici VS tool commerciali

Negli ultimi decenni, c'è stato molto fermento nel campo dell'high-level synthesis: sono stati sviluppati numerosi tool, sia in ambito accademico, sia in ambito commerciale. Ciascuno di questi tool accetta un input un linguaggio differente (talvolta supportandone più di uno), realizza diverse ottimizzazioni interne e produce risultati di diversa qualità, anche a fronte della stessa descrizione dell'input [42].

Xilinx Vivado HLS, Intel HLS Compiler, Intel FPGA SDK per OpenCL, Mentor Catapult e Cadence Stratus sono dei tool commerciali molto diffusi. Tra i tool accademici open-source spiccano LegUp, realizzato dall'Università di Toronto, Bambu, prodotto dal Politecnico di Milano, Dwarv, sviluppato alla Delft University of Technology, ROCCC della University of California, GAUT dell'Université Bretagne Sud e KiwiC, realizzato della University of Cambridge Computer Laboratory [4].

La maggior parte di questi tool supportano sia C/C++, sia SystemC.

I tool accademici open-source presentano molti vantaggi: forniscono gratuitamente supporto per risolvere bug e per sviluppi futuri mediante dei forum, contrariamente ai tool commerciali, il cui supporto è molto costoso.

Consentono inoltre di fissare numerosi obiettivi di ottimizzazione, tra cui l'area, il consumo di potenza, la velocità, la testabilità, ... mentre spesso i tool commerciali supportano solo alcune ottimizzazioni. In questo modo il progettista può sviluppare il suo proprio tool di HLS focalizzandosi sulle ottimizzazioni richieste dal suo progetto. Il codice sorgente del tool è infatti messo a disposizione del progettista, che può modificarlo per renderlo adatto a un'applicazione specifica [44].

Al contrario, per quanto concerne i tool commerciali, è molto difficile reperire i dettagli di progettazione. Fanno eccezione alcuni tool commerciali realizzati in collaborazione con la ricerca universitaria (Vivado HLS), che sono più documentati di quelli realizzati interamente nell'industria (come Intel HLS Compiler).

Dal momento che non esistono delle scadenze in termini di tempo per lo sviluppo dei tool accademici, gli sviluppatori sono meno soggetti alle pressioni cui invece è sottoposto il settore industriale; ciò consente di evitare errori che potrebbero essere commessi a causa dei rigidi vincoli di tempo a cui gli sviluppatori di tool commerciali devono sottostare.

D'altra parte, proprio a causa dell'assenza di scadenze, lo sviluppo e il supporto in alcuni tool di HLS accademici non sono sempre garantiti: trattandosi di progetti open source, esiste un'assistenza solo nella misura in cui c'è del personale che sta lavorando attivamente al progetto in un dato momento. La discontinuità probabilmente è dovuta dal fatto che tra gli sviluppatori possono esserci studenti che, una volta terminato il loro percorso di studi, smettono di lavorare al progetto. La formazione di nuovi sviluppatori richiede tempo, quindi ci sono periodi più o meno attivi nello sviluppo di un tool open-source [44]. Nei tool commerciali, invece, il supporto all'utenza deve essere sempre garantito. Questo servizio è compreso nel prezzo di acquisto della licenza. In molti tool commerciali quest'ultima non è soggetta a una scadenza; lo è invece il supporto: al termine di un periodo stabilito dalla casa produttrice del tool, l'assistenza e la possibilità di scaricare gli aggiornamenti non sono più concessi. Ciò dimostra l'importanza

tanza di avere un supporto costante da parte del produttore del tool.

Un'altra differenza tra i tool open source e quelli commerciali consiste nel fatto che la maggior parte dei tool HLS open source genera solo l'architettura RTL o codici intermedi, ma non la netlist finale. Quindi, sono necessari ulteriori tool per implementare il progetto in hardware. Un elevato numero di vendori di tool invece spesso integra l'high-level synthesis con altri tool offerti dalla compagnia. Per esempio, Cadence CtoS è integrato con il compilatore Cadence RTL per effettuare una stima dell'area utilizzata e considerazioni relative al timing durante l'operazione di scheduling. Sia il tool di Synopsys che quello di Cadence generano automaticamente testbench per le loro piattaforme di verifica [45]. Xilinx vende i suoi tool per la progettazione digitale in una vera e propria suite, che comprende il tool HLS e di sintesi logica, strumenti di implementazione, di verifica e altri, coprendo così ogni fase del processo di progettazione.

In generale, la maggior parte dei tool HLS ha come target gli FPGA più utilizzati (come quelli prodotti da Xilinx, da Intel o da Altera). Ci sono anche altri fornitori di FPGA come Lattice, Actel, Quick Logic, Atmel, Achronix, come si può riscontrare nelle Figure 7.1 e 7.2.

Tali Figure mostrano rispettivamente tool commerciali e tool accademici che erano disponibili nel 2020, tratti da un'indagine condotta specificamente su tool e toolchain di high-level synthesis [4].

Come si evince dalle Figure, la maggior parte dei tool accetta in ingresso il linguaggio C/C++ e SystemC ed è integrata in una suite software.

Tool	Provider	Intro.	Update	Pedigree	Platform	Input	Output	Design tool	Domain	Supported features								
										PR	P2P	P2F	FxP	FP	AP	RF	DM	AN
Vivado HLS [23]	Xilinx	2008	2019	xPilot (UCLA) → AutoPilot (AutoESL) → Vivado HLS (Xilinx)	Xilinx FPGAs	C, C++, System C, OpenCL API	Verilog, VHDL	Included in Vivado Design Suite	All	●	●	○	●	●	●	○	○	●
Intel FPGA SDK for OpenCL [24]	Intel (Altera)	2013	2019	Intel acquired Altera in 2015	Heterogeneous platforms with Intel CPUs and FPGAs	OpenCL kernels	Verilog	Stand-alone	All	●	○	○	●	●	●	○	○	●
Intel HLS Compiler [35]	Intel (Altera)	2017	2019	Intel acquired Altera in 2015	Intel FPGAs	C++	Verilog	Included in Intel Quartus Prime Design	All	●	○	○	●	●	●	●	○	●
Catapult HLS [36]	Mentor Graphics	2004	NI	Catapult-C (Mentor Graphics) → Catapult-C (Calypso, 2011) → Catapult HLS (Mentor Graphics, 2015)	All FPGAs and ASICs	C++	Verilog, VHDL	NI	All	●	○	○	●	●	●	●	○	●
Stratus HLS [32]	Cadence	2015	NI	C synthesizer (Forte) + C-to-Silicon Compiler (Cadence) → Stratus HLS (Cadence)	All FPGAs and ASICs	C++, SystemC	Verilog	NI	All	●	○	○	●	●	●	●	○	●
CyberWorkBench [43]	NEC	2011	NI	Based on Cyber [44]	Xilinx and Altera FPGAs	BDL	Verilog, VHDL	Stand-alone	All	●	○	○	●	●	●	○	○	●
C-to-Hardware [45]	Altium	2008	2019		Xilinx, Intel and Microsemi FPGAs	C	Verilog, VHDL	Included in Altium Designer	All	●	○	○	●	●	○	○	○	●
Synphony HLS [46]	Synopsys	2003	2017	Based on PICO [47]	All FPGAs and ASICs	C, C++, Simulink models	Verilog, VHDL	Used as a blockset in Simulink	DSP	○	○	○	●	○	○	○	○	○
HDL Coder and Verifier [48]	Mathworks	2012	2019		Xilinx, Intel and Microsemi FPGAs	Matlab functions and Simulink models	Verilog, VHDL	Used as an add-on in MATLAB and Simulink	All	○	○	○	●	○	○	○	○	○

NI = No Information found

PR = Pointer, P2P = Pointer-to-pointer, P2F = Pointer-to-function, FP = Floating point, FxP = Fixed-point, AP = Arbitrary bit precision

RF = Recursive function, DM = Dynamic memory allocation, AN = Annotations required

● = Supported, ○ = Not supported, ●○ = Supported with restrictions

Figura 7.1: Tool commerciali attualmente disponibili [4]

Tool	Provider	Intro.	Update	Pedigree	Platform	Input	Output	Design tool	Domain	Supported features							
										PR	P2P	P2F	FxP	FP	AP	RF	DM
LegUp [28], [49]	University of Toronto and LegUp Computing	2011	2020		Intel FPGAs and limited support to Xilinx FPGAs	C	Verilog	Stand-alone	All	●	○	○	○	●	○	○	○
Bambu [26]	Politecnico di Milano	2012	2017		All FPGAs	C	Verilog	Compatible with Xilinx Vivado Design Suite, Altera Quartus and Lattice Diamond	All	●	○	○	○	●	○	○	○
ROCCC [37], [38]	UC Riverside and Jacquard Computing	2002	2013	Inspired by SA-C compiler	FPGAs	A subset of C	VHDL	Stand-alone	Streaming applications	○	○	○	○	●	○	○	○
GAUT [39], [40]	Université de Bretagne-Sud	2008	2017		Xilinx and Intel FPGAs	C, C++	VHDL	Compatible with Vivado Design Suite and Synopsys Design Compiler	DSP	○	○	○	●	○	●	○	○
Kiwi Compiler [41], [42]	University of Cambridge	2008	2017		Xilinx and Altera FPGAs	.NET bytecode	Verilog	Stand-alone	All	●	●	○	○	●	○	●	●
Handel-C Compiler [50], [51]	Oxford University Computing Laboratory	1996	NI		Xilinx and Altera FPGAs	Handel-C	Verilog, VHDL	Used in DK Design Suite [52] from Mentor Graphics	All	○	○	○	●	○	○	○	○
DWARV [27]	Delft University of Technology	2007	2014		FPGAs	C	VHDL	Included in Delft Workbench [53]	All	●	○	○	●	●	○	○	●

NI = No Information found

PR = Pointer, P2P = Pointer-to-pointer, P2F = Pointer-to-function, FP = Floating point, FxP = Fixed-point, AP = Arbitrary bit precision

RF = Recursive function, DM = Dynamic memory allocation, AN = Annotations required

● = Supported, ○ = Not supported, ●○ = Supported with restrictions

Figura 7.2: Tool accademici attualmente disponibili [22]

7.2 Quali sono i tool più diffusi?

In questo paragrafo sarà effettuata una breve presentazione di alcuni dei tool e dei compilatori più diffusi; tra quelli accademici sono stati selezionati LegUp e Bambu, tra quelli commerciali Vivado HLS, Intel FPGA SDK FOR OpenCL, Intel HLS Compiler e Catapult HLS Platform.

E' importante fare una distinzione tra tool e compilatore: il compilatore genera in output un object file, indipendente dal linguaggio, che verrà poi eseguito su un sistema operativo per creare una netlist, mentre il tool restituisce direttamente il file della netlist RTL che sarà usato per l'implementazione hardware del sistema digitale [44].

7.2.1 Vivado HLS

Vivado HLS [38] è un tool commerciale fornito da Xilinx, sviluppato ad hoc per le piattaforme FPGA. Esso deriva da xPilot, sviluppato negli anni 2000 dall'università della California, sede di Los Angeles, successivamente commercializzato in AutoPilot e infine acquisito da Xilinx nel 2013. Vivado HLS accetta in input codice in C sintetizzabile, C++ e SystemC e OpenCL, effettua avanzate trasformazioni del codice platform-based e ottimizzazioni di sintesi, generando così una descrizione RTL in VHDL e in Verilog.

Il codice in input viene compilato mediante l'infrastruttura di compilazione LLVM (Low Level Virtual Machine); il codice HLL viene così trasformato in una IR (intermediate representation), successivamente sottoposta a una serie di operazioni di compilazione standard, che comprendono l'eliminazione del dead-code, la constant propagation, loop unrolling e ottimizzazioni specifiche dell'hardware come l'ottimizzazione della bit-width per ridurre la complessità del codice e la ridondanza e ottimizzazioni relative alla locazione dei dati e al parallelismo.

Basandosi sui vincoli specificati dall'utente e sull'IR modificata, Vivado HLS effettua ottimizzazioni interconnesse durante le fasi di scheduling delle operazioni e del binding delle risorse. Per ottimizzare

le prestazioni, l'utilizzo delle risorse, la comunicazione dei dati tra la CPU e l'hardware progettati, i processi di compilazione e di sintesi vengono guidati da direttive e pragma fornite da Vivado HLS. Inoltre, il tool mette a disposizione librerie e API (application program interface) ottimizzate per l'hardware come supporto ai progettisti. Infine, l'IR viene sintetizzato in implementazioni RTL per Xilinx FPGA. L'RTL generato può essere salvato in una libreria IP per un uso successivo.

Vivado HLS consente di effettuare la co-simulazione hardware/software, mediante dei wrapper (cioè degli involucri) di testbench in SystemC che permettono ai progettisti di impiegare il framework di test originale in C/C++/SystemC/OpenCL per verificare la correttezza dell'output RTL [27]. Crea inoltre degli script di simulazione per simulatori RTL, presenti all'interno della stessa suite di cui Vivado HLS fa parte. Questo tool HLS infatti è incluso in Vivado Design Suite e può essere utilizzato in abbinamento a tool di sviluppo di sistemi eterogenei forniti da Xilinx. Xilinx ha recentemente rilasciato Vitis Unified Software Platform, che è chiamato Vitis HLS, con alcune funzionalità aggiuntive [4].

7.2.2 Intel FPGA SDK FOR OpenCL

Open Computing Language (OpenCL) è un framework open source per la programmazione di applicazioni parallele per piattaforme eterogenee inclusi processori, GPU e FPGA. SDK Intel FPGA per OpenCL [46] è appunto un SDK (Software Development Kit) per la programmazione multi-core. Introdotto da Altera nel 2013 e successivamente acquisito da Intel nel 2015, questo compilatore genera circuiti hardware che fanno uso di pipeline che possono essere implementati sugli FPGA di Intel, quindi non esegue thread paralleli di costose funzioni su più core. Il tool trasforma OpenCL in Verilog e in librerie runtime per la parte del sistema in esecuzione sul processore. Utilizza il compilatore LLVM-Clang per analizzare i costrutti OpenCL e produce un IR (intermediate representation). L'IR passa attraverso una serie di ottimizzazioni, inclusa l'eliminazione di branch, la fusione di loop e la vettorizzazione automatica. L'utente ha la possibilità di gestire delle ottimizzazioni inserendo dei commenti per il loop unrolling, l'impiego di pipeline e per lo streaming di dati. Il compilatore applica automaticamente queste ottimizzazioni e traduce l'IR ottimizzato in Verilog.

7.2.3 Intel HLS Compiler

Il compilatore Intel HLS [47] consente di tradurre una descrizione dell'hardware in C++ in una descrizione RTL per gli FPGA Intel. Oltre alle ottimizzazioni standard del compilatore, Intel HLS Compiler, basandosi sulle annotazioni fornite dall'utente, esegue attività di ottimizzazione come loop unrolling e pipelining. L'utente ha la possibilità di esplorare architetture hardware incluse interfacce, parallelismo, memorie, datapath e loop utilizzando annotazioni e attributi specifici. Similmente a Vivado HLS, anche il compilatore Intel HLS facilita la generazione di IP riutilizzabili che possono essere integrati in un dato sistema e riduce i tempi di sviluppo degli FPGA. Nella Intel Quartus Prime Design Software Suite di cui questo tool HSL fa parte, è presente anche un tool di verifica dell'RTL generato.

7.2.4 Catapult HLS Platform

La piattaforma Catapult HLS [48] è stata inizialmente sviluppata da Mentor Graphics con il nome Catapult-C e successivamente è stata acquisita da Calypto Design Systems. È stato riacquistato da Mentor Graphics nel 2015, a sua volta acquisito da Siemens nel 2017. Catapult HLS mette a disposizione degli sviluppatori un subset di C++ e SystemC per generare una descrizione ottimizzata in Verilog o in VHDL per FPGA ed ASIC. La maggior parte dei costrutti in C++ sono supportati da Catapult: classi, struct, array e puntatori ad oggetti allocati in modo statico; inoltre è possibile includere interi di lunghezza arbitraria, fixed-point, floating-point e tipi di dato complessi utilizzando i tipi di dato Algorithmic-C [40].

Mediante l'uso di attributi e annotazioni, gli sviluppatori possono definire nelle specifiche vincoli legati al parallelismo, al throughput e alla configurazione della memoria; anche se la progettazione avviene ad alti livelli di astrazione, è comunque necessaria una comprensione approfondita dell'hardware per

ottenere un buon risultato. Durante la sintesi dell’hardware, Catapult esegue una serie di ottimizzazioni tra cui il loop unrolling, il loop merging e il pipelining. Catapult consente di generare automaticamente un’infrastruttura di simulazione per verificare l’RTL generato dalla HLS, confrontandolo con il codice sorgente HLL originale. In virtù della separazione della funzionalità del progetto dai dettagli di implementazione, con Catapult HLS è possibile effettuare delle modifiche alle specifiche, destinandole poi a un’altra tecnologia. L’RTL può essere semplicemente rigenerato in base al modello HLS modificato e ai nuovi vincoli.

7.2.5 LegUp

LegUp [49] è un tool di HLS open-source, sviluppato dai ricercatori dell’Università di Toronto, e ora supportato da LegUp Computing Inc.

A partire da una descrizione comportamentale scritta in C, LegUp genera una netlist RTL scritta in Verilog HDL, che può essere sintetizzata su FPGA prodotti da Intel, Xilinx, Lattice, Microsemi e Achronix. Funziona in tre modalità: modalità software, modalità hardware e software-hardware. La modalità software è eseguibile sul computer. In modalità hardware, l’intero codice C di input viene sintetizzato in RTL. In modalità software-hardware, il codice C viene sintetizzato in un sistema eterogeneo composto da un processore soft-core e da acceleratori costruiti su un FPGA.

Il tool è sviluppato per il sistema operativo Linux e supporta la maggior parte delle funzionalità di C per la sintesi di FPGA, ma non la ricorsione o l’allocazione dinamica della memoria. Il frontend di LegUp è costruito utilizzando il framework LLVM per generare la rappresentazione intermedia dal programma sorgente in C, che viene ottimizzato mediante tecniche di loop unrolling e di function inlining. Il backend è costituito dall’allocation, dallo scheduling e dal binding. Grazie a uno script di configurazione TCL (tool command language), vengono specificati vincoli legati al timing, alle risorse utilizzate e alla tecnologia target. Tramite un processo manuale, è inoltre possibile inserire POSIX Threads (che consiste in un modello di esecuzione parallelo, indipendente dal linguaggio di programmazione, che permette a un programma di gestire più thread, ovvero più processi contemporaneamente) oppure annotazioni OpenMP, un’API per convertire esecuzioni single-thread in C in esecuzioni in parallelo [44], [4].

7.2.6 Bambu

Bambu [50] è un framework HLS open-source realizzato dal Politecnico di Milano, parte del progetto PandA. Esso è scritto in C++ e accetta in input una descrizione comportamentale in C e un file XML che specifica le configurazioni corrispondenti alle diverse fasi del flusso di progettazione. Usa il compilatore GNU Compiler Collection (GCC) per eseguire target e ottimizzazioni indipendenti dal linguaggio, producendo IR (intermediate representation) nella forma di call-graph e CDFG. Sulla base della struttura del call-graph, il processo di sintesi prevede tre fasi distinte, che portano alla generazione di moduli hardware, più precisamente di un datapath e di un controller FSM, come di consueto nella sintesi di processori, e all’inizializzazione della memoria, decidendo così dove allocare le variabili di ogni funzione del programma in input scritto in C. Tali moduli vengono combinati tra loro per generare una netlist RTL e un programma di testbench in Verilog HDL. I risultati della sintesi possono essere implementati su ASIC (usando Synopsys Design Compiler) o su FPGA (mediante Xilinx Vivado Design Suite o Intel Quartus). Bambu produce anche testbench e script per la simulazione RTL che possono essere eseguiti nei simulatori Xilinx, Mentor Modelsim, Verilator e Verilog Icarus. Questo tool può essere modificato per la generazione di netlist relative a un preciso dominio di applicazione mediante caratterizzazione delle risorse di libreria. Sono molte le ottimizzazioni che vengono effettuate nella fase di front-end di Bambu: le moltiplicazioni e le divisioni vengono trasformate; vengono poi effettuate operazioni quali l’operation chaining e il code motion (che consente di spostare delle operazioni fuori da un loop), riducendo il numero totale dei cicli di clock; inoltre tramite la libreria FloPoCo le operazioni floating-point possono essere convertite in hardware [44], [4].

7.3 Quali fattori è necessario tenere in considerazione per la scelta del tool?

Come si è potuto riscontrare dalla breve presentazione di alcuni tool selezionati, ciascuno dei tool presenta le sue peculiarità e caratteristiche distintive.

Per poter scegliere il tool di high-level synthesis su cui implementare un progetto è necessario tenere conto di molti fattori, tra cui le applicazioni che il tool offre, il prezzo, il supporto della comunità e il suo supporto per gli FPGA di un particolare venditore. Ciascun tool infatti risponde e si adatta a diverse esigenze e per questo motivo attualmente non c'è un tool specifico che domina l'industria [4].

I vendori si distinguono per le funzionalità aggiuntive fornite parallelamente a quelle di base della sintesi ad alto livello [45].

Ad esempio, Xilinx Vivado è sviluppato ad hoc per la sintesi di piattaforme FPGA di sua produzione e consente di realizzare uno srotolamento dei loop ottimizzato [38].

Cadence CtoS ha la capacità di realizzare automaticamente una pipeline che può generare logica operante in parallelo con un throughput elevato [51].

Dalle Figure 7.1 e 7.2 è possibile osservare quali dei tool maggiormente diffusi supportano funzionalità come puntatori, puntatori a puntatori, puntatori a funzioni, floating point, fixed-point, arbitrary bit precision, funzioni ricorsive, allocazione dinamica della memoria e la possibilità di inserire annotazioni.

Come discusso nel Capitolo 6, funzioni ricorsive e allocazione dinamica della memoria rientrano tra le operazioni che non sono ancora consentite. Fa eccezione KiwiC, che attualmente supporta l'allocatione dinamica della memoria e funzioni ricorsive, pur con delle restrizioni.

HLS TOOL USAGE BY PAPERS		
HLS Tool	N	Tool language
AccelDSP	1	MATLAB
Altera OpenCL	3	OpenCL
Bluespec	2	Bluespec language
C2RTL	1	C
Cadence Stratus	1	C/C++/SystemC
CAPH Toolset	2	CAPH language
Catapult-C	2	C/C++/SystemC
Chisel	2	Scala
Cadence C-to-Silicon	2	C/C++/SystemC
Convey Hybrid-Threading	1	HT language
HCE	2	C
HIPAcc	2	HIPAcc language
Impulse C	1	C
LegUp	3	C
MATLAB Simulink HDL Coder	1	MATLAB functions, Simulink models
Maxeler MaxCompiler	1	Java
ROCCC	1	C
Xilinx System Generator for DSP	2	MATLAB Simulink
Xilinx Vivado HLS/Autopilot	18	C/C++/SystemC
Undisclosed	4	N/A

Figura 7.3: Utilizzo dei tool di sintesi emerso in una ricerca del 2019 [24]

Una ricerca condotta nel 2019 su 46 paper e 118 applicazioni ad essi associate ha fatto emergere che Vivado HLS (noto formalmente come Autopilot) è il tool di sintesi ad alto livello più diffuso, almeno in ambito accademico, come riportato dalla Figura 7.3. Tutti gli altri tool presentano un uso più sporadico. La popolarità di Vivado è probabilmente da imputarsi al fatto che Xilinx è il maggior venditore di FPGA, i cui strumenti di sintesi sono presenti nella suite di Vivado HLS.

Il numero elevato di tool presenti nel panorama accademico e commerciale è comunque indice della relativa immaturità di questo campo [24].

Altri fattori che è opportuno prendere in considerazione per poter scegliere il tool che più incontra i requisiti di un dato progetto sono il linguaggio adottato dal tool, il supporto per tipi di dato, la possibilità

di esplorare lo spazio di progettazione, la capacità del tool di generare un progetto che sia il più possibile corretto, l'efficacia degli strumenti di verifica messi a disposizione.

Tali criteri, tuttavia, consentono solamente di effettuare un confronto statico tra i vari tool; per questo motivo è nata l'esigenza di effettuare un confronto "dinamico", che consenta di testarli, permettendo così agli utenti di effettuare la scelta che più si adatta alle loro esigenze.

Per poterli testare è necessario applicare dei benchmark - appunto test del software che consentono di effettuare operazioni di simulazione - per poi confrontare tra loro i risultati in termini di performance e uso delle risorse.

In uno studio del 2015 [42] è stato proposto un metodo sistematico di analisi dei tool che consiste appunto nel testarli sottoponendoli agli stessi benchmark.

Oggetto dell'indagine sono stati tre tool open-source - Dwarv, LegUp e Bambu - e un tool commerciale; i benchmark, in linguaggio C, sono stati scelti in modo tale da coprire diversi domini di applicazione, sia control-flow, sia data-flow e sono stati tatti da ChStone BenchmarkSuite, o forniti da Bambu e Dwarv.

La prima differenza tra i tool consisteva nel fatto che essi fanno uso di diversi compilatori e che consentono l'implementazione del codice in C su diversi dispositivi FPGA. Ogni compilatore ha un diverso insieme di ottimizzazioni che vengono eseguite prima della sintesi ad alto livello, con un impatto sui risultati di sintesi.

Altre differenze sostanziali tra i tool, riguardano l'implementazione della memoria. Per ogni test di benchmark, alcuni dati vengono mantenuti in locale (cioè nelle BRAM istanziate all'interno del modulo), mentre altri dati sono considerati globali. Questi ultimi vengono quindi tenuti fuori dal nucleo e sono accessibili mediante un controller della memoria. La decisione riguardo a quali dati mantenere in locale e quali definire come globali differisce da un tool all'altro.

Per testare i compilatori sono stati effettuati due esperimenti: nel primo caso, ogni tool ha eseguito i vari benchmark con delle impostazioni predefinite, in una modalità "push-button"; nel secondo caso, l'implementazione è stata guidata dall'utente, con ottimizzazioni del programma e inserimento di vincoli messi a disposizione dal tool (come flag del compilatore e annotazioni del codice per abilitare varie ottimizzazioni) [42].

Per valutare le prestazioni del circuito sono state utilizzate tre metriche: la frequenza massima, la latenza del ciclo (ovvero il numero di cicli di clock necessari per completare i calcoli in un dato benchmark) e il wall-clock time (minimo periodo di clock moltiplicato per la latenza del ciclo). Per valutare l'area, sono state considerate la logica, i DSP (digital signal processor) e l'utilizzo della memoria. Grazie a queste metriche, è stato possibile effettuare una valutazione analitica della QoR dei tool, rendendo così possibile il confronto tra tool e rispondendo all'obiettivo di fornire alle aziende dei criteri per la scelta del tool più adatto alle loro necessità.

Un altro fattore non trascurabile che influenza la scelta di un dato tool da parte di un'azienda è certamente quello economico.

Disporre delle licenze per l'utilizzo di tool di high-level synthesis comporta infatti dei costi non indifferenti per le aziende. Informazioni relative agli accordi stipulati tra software house, ovvero le produttrici dei tool, e multinazionali che li impiegano nella fase di progettazione di SoC non sono state reperite. Tuttavia, consultando i siti dei vari produttori di tool è stato riscontrato che il costo di una singola licenza è dell'ordine dei migliaia di dollari; in questa cifra spesso sono inclusi anche il supporto all'utenza e la possibilità di scaricare gli aggiornamenti per un dato periodo di tempo, in seguito al quale sarà necessario rinnovarla per poter usufruire dei suddetti servizi.

Ad esempio, Vivado HLS è incluso all'interno di una suite di tool della Xilinx. L'acquisto di una singola licenza dà diritto all'uso di tutta la suite Vivado ML.

Oltre al tool di high-level synthesis Vivado ML fornisce supporto per tutti i dispositivi Xilinx e comprende al suo interno Vivado IP Integrator, Dynamic Function eXchange, Vitis High-Level Synthesis, Vivado Simulator, Vivado Device Programmer, Vivado Logic Analyzer, Vivado Serial I/O Analyzer, Debug IP, Synthesis and Place and Route; grazie a questi tool il progettista è supportato in ogni fase della progettazione dell'hardware.

Il costo di una licenza è di \$2995 per la Node-Locked License, che ne consente l'uso su una macchina

specifica, oppure alla cifra di \$3595 per la Floating License, che è installata su un server di rete, consentendo ai dispositivi che vi si connettono di richiamare la licenza per poterla utilizzare; l'uso del prodotto su più dispositivi contemporaneamente è comunque limitato a un numero di utenti pari al numero di licenze acquistate.

Nel caso specifico di Vivado HLS, ogni licenza rilasciata presenta un limite di versione, che determina la fine del periodo di garanzia del cliente, tipicamente della durata di 6 mesi. Tutte le versioni rilasciate fino a quella data potranno essere scaricate liberamente dall'utente. Il limite di versione non costituisce comunque una data di scadenza: le licenze vengono rilasciate permanentemente e in questo modo l'utente può fare uso dei tool e degli IP messi a disposizione dagli stessi indefinitamente, purchè facciano parte delle versioni a disposizione dell'utente.

Capitolo 8

Sviluppi recenti

8.1 Applicazioni

Negli ultimi quindici anni, l'high-level synthesis è stata impiegata per lo sviluppo di migliaia di ASIC e progetti FPGA in un'ampia varietà di domini applicativi come networking, image e video processing e crittografia.

Nel campo del networking, è stata utilizzata principalmente per progettare diversi IP capaci di implementare algoritmi utilizzati in dei protocolli di comunicazione standard. Ad esempio, nell'ambito della comunicazione wireless, l'high-level synthesis è stata impiegata per la tecnologia 3G e 4G e ora viene utilizzata attivamente per il 5G [52]; in tempi recenti è stata utilizzata per implementare dispositivi capaci di emettere cognitive radio (CR), cioè onde radio che possano essere programmate e configurate dinamicamente per utilizzare i migliori canali wireless disponibili in un dato momento, evitando così interferenze e congestioni.

Per quanto riguarda il settore del video processing, è stata molto utilizzata per implementare hardware per la decodifica di video [25]. Uno dei recenti successi dell'high-level synthesis infatti è la progettazione del VP9 decoder video. Anche i decoder video per lo standard di compressione video H.264 e il suo successore HEVC (High Efficiency Video Coding) sono stati progettati mediante high-level synthesis. L'high-level synthesis è inoltre venuta in soccorso per l'implementazione hardware di reti neurali convoluzionali (CNN), un tipo particolare di reti neurali artificiali (ANN) che rappresentano lo stato dell'arte nel riconoscimento e nella classificazione delle immagini con una tecnica di apprendimento supervisionato. Tale tecnica è largamente impiegata per l'analisi dei Big Data, dove a causa dell'enorme quantità dei dati da trattare, è fondamentale trovare tecniche per velocizzare il calcolo.

Nel campo del medical imaging, è stato recentemente pubblicato un caso di studio per lo sviluppo di acceleratori hardware di algoritmi di elaborazione delle immagini per la radioterapia adattiva, una tecnica volta ad aumentare l'accuratezza della radioterapia [53].

In relazione al settore della computer vision, è stata impiegata per la guida autonoma di veicoli e per la virtual reality/mixed reality (VR/MR) per dispositivi Internet of Things (IoT) per la codifica di video a bassissima latenza [54].

Altri esempi di applicazione includono applicazioni aerospaziali, simulazioni litografiche e analisi di dati cosmologici [27].

Nell'ambito della crittografia, l'high-level synthesis viene impiegata per implementare efficienti architetture capaci di eseguire algoritmi crittografici, che fanno uso di strumenti matematici complessi, come la trasformata teorica dei numeri NTT [55] o eseguono operazioni come la moltiplicazione polinomiale [56], che in questo modo vengono notevolmente accelerate.

8.2 State of art e possibili miglioramenti

Oltre alle questioni discusse nel Capitolo 4, in particolare relative all'esplorazione dello spazio di progettazione e alla verifica, sono molte le sfide aperte cui la ricerca sta cercando di trovare risposta nel campo della high-level synthesis.

Direzioni della ricerca per gli sviluppatori di tool

La gestione della memoria

Una delle direzioni della ricerca riguarda l'ottimizzazione degli accessi in memoria. La maggior parte dei tool di HLS manca di supporto efficiente per la gerarchia della memoria e di un'astrazione sufficiente degli accessi alla memoria esterna [24]. Di conseguenza, i software engineers sono esposti a dettagli di basso livello, come le interfacce dei bus e i controller della memoria, che oltre a essere in contrasto con il paradigma della descrizione comportamentale, prerogativa della HLS, complica il lavoro di tali figure professionali, costrette a uscire dalla loro comfort-zone. Per questo motivo, sarebbe preferibile che i trasferimenti esplicativi della memoria esterna venissero nascosti il più possibile ai programmati. Ciò richiederebbe il supporto di gerarchie di memoria efficienti, inclusi il caching automatico e il prefetching per nascondere la latenza della memoria e migliorare la località dei dati [27].

Correlazione tra codice sorgente, RTL e hardware generato

Spesso la relazione tra il codice sorgente e l'hardware generato non è trasparente e questo rende difficile identificare le parti di codice meno ottimali che richiederebbero delle modifiche [24]. I tool dovrebbero rendere visibili in maniera dettagliata le correlazioni tra RTL generato e codice sorgente ad alto livello. Molte delle moderne soluzioni HLS forniscono la possibilità di fare riferimenti incrociati tra il codice C e l'RTL per aiutare i progettisti a capire i risultati della sintesi. Tuttavia, talvolta tali correlazioni sono molto difficili da mantenere a causa delle intense ottimizzazioni che avvengono durante il processo di sintesi [45]. Sarà necessario sviluppare ulteriormente i tool affinché la corrispondenza tra codice sorgente e RTL possa essere più nitida.

Parallelismo task-level

La ricerca sta lavorando perchè l'high-level synthesis consenta una maggiore esplorazione dello spazio di progettazione non soltanto a livello di blocco, ma anche a livello full-system, portando a un'ulteriore transizione nei livelli di astrazione, dal livello algoritmico al livello di sistema, consentendo di progettare l'intero sistema digitale, potenziando il parallelismo tra i vari blocchi e non solo quello a livello di istruzioni e cicli all'interno del blocco stesso.

Nei principali tool esistono già alcuni strumenti per specificare il parallelismo tra processi concorrenti comunicanti, cioè tra blocchi, ad esempio attraverso l'inserimento di annotazioni manuali oppure un altro approccio tenta di estrarre del parallelismo task-level sviluppando dei data flow sincroni. Una tecnica più recente prevede l'impiego delle reti di processo di Kahn [27], che consente di aggiungere un sottile strato sopra la chiamata di funzione utilizzando dei canali FIFO (first-in-first-out) che stabiliscono una comunicazione tra le funzioni.

Un'alternativa messa a disposizione da SystemC è quella di utilizzare le interfacce modulari di I/O che encapsulano le interfacce cycle e pin accurate, consentendo una visione a livello di transazione per la simulazione. Tali interfacce costituiscono un modo per separare le comunicazioni cycle accurate in modo che il resto del comportamento mantenga il suo livello di astrazione ad alto livello, riuscendo a gestire più agilmente il parallelismo tra i blocchi [26].

In un studio del 2021 [57] viene proposto un framework open-source chiamato TAPA (TAsk-PArallel), specializzato nella progettazione di sistemi ad ampio parallelismo task-level. In TAPA, attraverso un'estensione del linguaggio C++, vengono fornite pratiche interfacce di programmazione, simulazioni task-parallel e la possibilità di generare codice gerarchico, riuscendo così a ovviare alle limitazioni che caratterizzano la maggior parte dei tool di HLS e ponendosi così come riferimento nell'ambito del parallelismo a livello di task.

Supporto per tipi di dato dinamici

In [58], è stata proposta una tecnica per gestire le strutture dati dinamiche. Applicando diverse trasformazioni manuali del codice, gli autori sono riusciti ad aumentare le prestazioni, concludendo che i tool di HLS dovrebbero eseguire automaticamente simili ottimizzazioni con strutture dati dinamiche, per lo sviluppo delle quali saranno necessari ulteriori studi [24].

Al 2020, l'unico tool che supportava tipi di dato dinamici, pur con delle restrizioni, era il compilatore KiwiC [4]. Ad esempio, l'allocazione dinamica della memoria è supportata a condizione che venga invocata solo dai costruttori o una volta per tutte sui thread principali.

Ottimizzazione della potenza

Grazie all'esplorazione dello spazio di progettazione, è possibile generare diversi progetti su cui effettuare delle analisi di potenza in modo tale da trovare il progetto che presenta il minor consumo.

Ad esempio, il tool Catapult [48] presenta uno strumento di analisi di potenza incorporato.

Un metodo proposto recentemente per ottenere un'ottimizzazione dal punto di vista della potenza consiste nel clock-gating. Questa ottimizzazione consiste nella disabilitazione del segnale di clock per determinate parti del circuito quando non sono in uso, permettendo così un notevole risparmio energetico [26].

Gestione degli ECO

Gli ECO (engineering change order) sono una strategia di progettazione che lascia spazio alla possibilità di effettuare delle modifiche sull'hardware anche a componenti ultimati, consentendo di correggere eventuali errori.

Mentre gli ECO sono abbastanza comuni nel codice RTL creato manualmente, sono invece piuttosto rari nell'RTL generato dalla sintesi ad alto livello [25]. Nella tradizionale sintesi logica, essi si rendono necessari per il fatto che la sintesi RTL e il place and route sono effettuati prima che la verifica dell'RTL sia stata completata, quindi prima che eventuali errori possano manifestarsi. Per ridurre l'impatto sul lavoro completato, vengono effettuati degli aggiustamenti incrementali (gli ECO appunto) con una metodologia che interferisce il meno possibile con la progettazione. Poiché il codice sorgente comportamentale può essere verificato approfonditamente e i tool di HLS garantiscono che l'RTL generato sia corretto, gli ECO non sono molto comuni nel flusso di sintesi ad alto livello [24].

Tuttavia, i tool di HLS per l'implementazione di ASIC hanno la capacità di effettuare una sintesi incrementale per facilitare i flussi di ECO, anche se tali modifiche tipicamente non vengono catturate dalla descrizione comportamentale ad alto livello.

Rendere il progetto modulare è attualmente il modo migliore per ridurre la portata della modifica e quindi ridurre al minimo lo sforzo richiesto nel caso in cui diventi necessario un ECO [25].

Casi di studio

Portabilità

In uno studio del 2021, sono state valutate la portabilità e le prestazioni di kernel tratti dalla suite di benchmark Rodinia progettata per FPGA di Intel, apportando le modifiche necessarie per creare dei kernel sintetizzabili per un FPGA Xilinx. È stato scelto il linguaggio OpenCL, in quanto sia Intel che Xilinx lo supportano per la progettazione di FPGA.

Le difficoltà di realizzare alcune ottimizzazioni del kernel per l'FPGA Xilinx è stata variabile, a seconda del costrutto utilizzato. Una volta apportata la quantità minima di modifiche per creare hardware sintetizzabile per la piattaforma Xilinx, è stato necessario molto lavoro per riuscire a migliorare le prestazioni. Tuttavia, è stato riscontrato che i costrutti notoriamente più performanti per gli FPGA sono responsabili del miglioramento delle prestazioni in maniera indipendente dalla piattaforma [59].

Sulla stessa lunghezza d'onda è stato un webinar organizzato da Siemens nel 2020, in cui venivano date indicazioni precise su come convertire un progetto realizzato su Vivado HLS in un progetto della piattaforma Catapult HLS [60].

Attualmente il problema della portabilità è una questione ancora aperta nel campo della ricerca e necessita di ulteriori studi.

La questione della sicurezza

In uno studio del 2018 [61] è stata sollevata la questione della mancanza di sicurezza nei sistemi digitali impiegati in settori critici, come quelli di banche, auto e dispositivi medici, dove la sicurezza dovrebbe essere tenuta debitamente in considerazione.

Attualmente urge un'analisi attenta delle vulnerabilità dei SoC, al fine di sviluppare dei meccanismi di protezione - da implementare durante la fase di progettazione e non a posteriori - che evitino la perdita o la manomissione di informazioni.

Gli autori del paper auspicano che il loro lavoro si ponga come apripista per dare impulso alla progettazione di acceleratori hardware sicuri. Affinchè ciò possa verificarsi, il paper propone di impiegare i tool di high-level synthesis per la sintesi automatica di vari meccanismi di protezione. In questo modo sarà proprio l'high-level synthesis in futuro a consentire non solo la realizzazione di architetture eterogenee efficienti, ma anche sicure al tempo stesso.

Sintesi ad alto livello "push-button"

In uno studio del 2020 [4], sono stati analizzati i flussi di progettazione HLS per piattaforma FPGA dei tool, focalizzandosi sui tool che potrebbero essere eventualmente utilizzati per sintetizzare codice RTL automaticamente. E' emerso che nessuno degli strumenti esistenti è in grado di soddisfare la visione di distribuzione completamente automatica del codice C/C++ in un sistema eterogeneo di FPGA e CPU. Le capacità di cui i tool non sono ancora dotati includono la generazione di codice HLS sintetizzabile da codice high-level language che fa uso di puntatori a puntatori o a funzioni, funzioni ricorsive o allocazioni di memoria dinamica.

Capitolo 9

Conclusioni

Questo elaborato ha cercato di ripercorrere le tappe più significative della storia della high-level synthesis, di illustrarne il flusso di progettazione e di sottolineare il grande potenziale che essa offre, sempre maggiore grazie agli sforzi della ricerca. Inoltre sono stati presentati i linguaggi che nel corso dei decenni sono stati supportati dai vari tool fino ad approdare al linguaggio C/C++ e i tool maggiormente diffusi in ambito commerciale e accademico, cui è seguita una riflessione sui criteri che bisogna prendere in considerazione per valutarli e confrontarli tra loro sulla base dei requisiti del proprio progetto. Infine sono state esposte alcune delle maggiori sfide a cui la ricerca dovrà rispondere nel prossimo futuro.

Come si è potuto evincere dalla ricostruzione storica dell’andamento dell’HLS, negli ultimi vent’anni l’high-level synthesis ha fatto moltissimi progressi, al punto tale da essere gradualmente accettata nel campo dell’industria, mantenendo tuttavia una diffusione limitata e un ruolo subalterno rispetto alla tradizionale sintesi logica, che resta tutt’ora il metodo imperante nel campo della progettazione digitale.

Una delle cause è da ricercarsi nel fatto che, in effetti, il gap tra sintesi ad alto livello e sintesi logica non è ancora stato del tutto colmato [24]: la nuova generazione di tool HLS non è ancora in grado di garantire delle performance e un uso delle risorse pari a quello della sintesi RTL manuale.

Anche se l’high-level synthesis non riesce ancora ad eguagliare il lavoro manuale degli hardware engineers in termini di qualità dei prodotti, i numerosi vantaggi - dalla minor quantità di codice necessario a implementare i progetti, all’esplorazione dello spazio di progettazione, alla facilitazione del processo di verifica, al coinvolgimento dei software engineers - concorrono complessivamente alla determinazione di un fattore, strettamente legato al lato economico, che la rende indiscutibilmente più vantaggiosa della lunga sintesi manuale dell’RTL: si tratta del time-to-market.

Il time-to-market è essenzialmente il tempo che intercorre tra la concettualizzazione di un prodotto e il suo lancio sul mercato. Durante questo periodo, il prodotto deve essere progettato, prototipato, verificato e validato, confezionato e poi rilasciato.

Nel campo dell’elettronica di consumo, la finestra del time-to-market è sempre più breve [62]. Di contro, per soddisfare le aspettative del consumatore, le compagnie devono dedicare molto tempo alla progettazione e allo sviluppo dei loro prodotti. Tuttavia tempi di sviluppo troppo lunghi possono rivelarsi dannosi per i piani di lancio, esponendo l’azienda al rischio che il prodotto possa non ricevere l’attenzione sperata, con ripercussioni negative sul fatturato e sul profitto. Pertanto, il prodotto non solo deve essere competitivo, incontrando il più possibile le esigenze del consumatore, ma deve anche essere rilasciato velocemente per riuscire a cogliere le occasioni date dalle brevissime finestre di mercato.

In questo scenario, l’high-level synthesis si pone come lo strumento più adatto nell’andare incontro agli interessi economici di un’azienda: accorciare il time-to-market, consentendo di tradurre un’idea in un progetto digitale velocemente ed efficientemente, può essere decisivo nella riuscita di un prodotto [21]. Da uno studio del 2019 [24] è emerso che grazie all’high-level synthesis il tempo per lo sviluppo di un progetto si è ridotto a un terzo del tempo impiegato per la sintesi manuale del codice RTL. L’indagine riporta inoltre un incremento della produzione di ben 6 volte. In virtù di ciò, l’high-level synthesis si configura come una scelta particolarmente oculata quando il time-to-market a disposizione è molto ridotto e non vi è un bisogno impellente di migliorare le performance o di ridurre l’uso delle risorse,

risparmiando così moltissimo tempo.

In realtà, la tecnologia di cui si dispone al giorno d'oggi è talmente performante che, anche se la soluzione che viene implementata mediante high-level synthesis non è la migliore possibile, si tratterà comunque di una soluzione buona.

Ipotizziamo che un'azienda si trovi di fronte alla possibilità di cogliere una finestra di mercato per un determinato prodotto.

Per svilupparlo, l'azienda decide di fare uso di un tool di high-level synthesis. In questo modo il prodotto viene sviluppato in poco tempo, ma la sua prima implementazione si rivela non essere del tutto ottimale. L'azienda si troverà quindi di fronte a un dilemma: è meglio dedicare più tempo allo sviluppo del prodotto e perfezionare il codice RTL generato dall'high-level synthesis per ottenere una qualità migliore, rimandandone quindi il lancio, oppure, al fine di conquistare una posizione di rilievo sul mercato, può risultare più vantaggioso lanciare il prodotto subito, anche se non è stato completamente rifinito?

Immettere tardivamente il prodotto potrebbe compromettere i profitti, vanificando gli sforzi dei progettisti di apportare dei miglioramenti alla QoR. Per questo motivo è possibile che l'azienda decida di scendere a qualche compromesso in termini di performance e utilizzo delle risorse, immettendo sul mercato la prima versione non perfettamente rifinita del prodotto, sviluppata mediante tool di HLS. Se il prodotto riscuote successo e l'azienda riesce quindi a conquistarsi una buona porzione di mercato, essa avrà tutto l'interesse a finanziare il perfezionamento di tale prodotto perché possa mantenere la visibilità conquistata e diventare sempre più competitivo: ora che è stato testato e che si dispongono di maggiori garanzie in merito alla sua fruibilità da parte dell'utenza, l'azienda sarà molto ben disposta nel concedere ai progettisti più tempo per poterlo rifinire e perfezionare, con l'obiettivo di raggiungere delle performance a cui certamente avrebbe potuto aspirare fin dal principio, ma che per questioni di natura economica non sono state implementate nella prima fase di sviluppo.

In definitiva, anche se i progetti sviluppati tramite high-level synthesis possono essere ulteriormente migliorati, grazie ai progressi raggiunti dai tool di HLS nell'ultimo decennio sussiste già la possibilità di poterne fare un uso attivo in ambito industriale.

Infatti, come già discusso, il movente dell'high-level synthesis è stato quello di permettere ai progettisti di gestire al meglio la complessità dei moderni sistemi digitali e oggi, seppur con alcune limitazioni nella qualità dei risultati, l'high-level synthesis è in grado di assolvere questa funzione.

Come si spiega dunque il fatto che l'high-level synthesis non sia ancora diventata una pratica comunemente accettata e largamente diffusa?

Per quanto l'HLS si ponga sempre più come una necessità, ci sono ancora alcune problematiche che ne frenano la diffusione.

In primo luogo, come già sottolineato, per ottenere il massimo delle prestazioni dalla tecnologia è ancora necessaria una progettazione RTL: l'high-level synthesis non ha ancora raggiunto una maturità tale da poterne consentire un uso esclusivo, cioè che non richieda successivamente il passaggio all'RTL per rifiniture manuali del codice.

In secondo luogo, un altro aspetto critico è legato ai linguaggi supportati dai vari tool: ad oggi non è stato elaborato alcuno standard (libreria SystemC esclusa) e ciascun tool presenta un suo proprio flusso caratteristico, con le sue rispettive estensioni e limitazioni al linguaggio C/C++ o più raramente ad altri linguaggi di programmazione, che rendono ardua la possibilità di passare agilmente da un tool all'altro, anche nel caso in cui i tool adottino lo stesso linguaggio, come mostrato da alcuni studi [59].

Altre criticità riguardano i prezzi dei tools. Essi presentano infatti costi non esigui, che ne precludono l'accesso alle industrie più piccole, impossibilitate a sostenere assiduamente le spese di rinnovo delle licenze, senza le quali, pur continuando a poter utilizzare versioni precedenti del tool, viene meno il supporto da parte del produttore.

Un'altra delle problematiche che continuano a sussistere è quella della formazione. Attualmente sono le aziende a formare i software e gli hardware engineers per fare uso dei tools di high-level synthesis, cercando di colmare i rispettivi gap nell'uso di tali strumenti. In ambito universitario, stanno cominciando ad essere erogati diversi corsi di progettazione hardware che affrontano l'uso dell'high-level synthesis [25]. Avvicinare gli studenti a questa nuova frontiera della sintesi di hardware potrà contribuire ad inte-

grarla a pieno titolo nel flusso di progettazione, riuscendo a usufruire di tutti i suoi vantaggi e stimolando la ricerca di tecniche e algoritmi sempre migliori per potenziare i tool.

Alla luce dei vantaggi e delle criticità qui esposte, quali potrebbero essere le prospettive a lungo termine di questo nuovo approccio alla sintesi digitale?

Se l'high-level synthesis riuscisse a colmare interamente il gap con la sintesi RTL nella QoR e a superare le criticità legate al problema dei linguaggi, trovando uno standard condiviso, e ai costi, rendendoli accessibili, si profilerebbe un ben preciso futuro: l'high-level synthesis non si limiterà ad affiancare la progettazione manuale dell'RTL, ma probabilmente porterà al declino dei linguaggi RTL e subentrerà loro definitivamente, diventando il metodo standard di progettazione dell'hardware, proprio come nell'ambito software i linguaggi di programmazione hanno sostituito il linguaggio assembly, oggi utilizzato solo per specifiche rifiniture del codice da sviluppatori specializzati [24].

Bibliografia

- [1] G.E. Moore. «Cramming more components onto integrated circuits». In: *Electronics* 38.8 (apr. 1965), pp. 114–117.
- [2] R.H. Dennard et al. «Design of ion-implanted MOSFET's with very small physical dimensions». In: *IEEE Journal of Solid-State Circuits* 9.5 (ott. 1974), pp. 256–268. DOI: [10.1109/JSSC.1974.1050511](https://doi.org/10.1109/JSSC.1974.1050511).
- [3] P. Gargini. «Roadmap evolution: from NTRS to ITRS, from ITRS 2.0 to IRDS». In: *Photomask Technology*. 2017.
- [4] M.W. Numan et al. «Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains». In: *IEEE Access* 8 (2020), pp. 174692–174722. DOI: [10.1109/ACCESS.2020.3024098](https://doi.org/10.1109/ACCESS.2020.3024098).
- [5] E.P. DeBenedictis. «It's Time to Redefine Moore's Law Again». In: *Computer* 50.2 (2017), pp. 72–75. DOI: [10.1109/MC.2017.34](https://doi.org/10.1109/MC.2017.34).
- [6] K. Wakabayashi. «C-based behavioral synthesis and verification - Analysis on industrial design examples». In: gen. 2004, pp. 344–348. DOI: [10.1145/1015090.1015177](https://doi.org/10.1145/1015090.1015177).
- [7] R.A. Walker e D.E. Thomas. «A Model of Design Representation and Synthesis». In: *22nd ACM/IEEE Design Automation Conference*. 1985, pp. 453–459. DOI: [10.1109/DAC.1985.1585981](https://doi.org/10.1109/DAC.1985.1585981).
- [8] M.C. McFarland, A.C. Parker e R. Camposano. «The high-level synthesis of digital systems». In: *Proceedings of the IEEE* 78.2 (1990), pp. 301–318. DOI: [10.1109/5.52214](https://doi.org/10.1109/5.52214).
- [9] D. Gajski et al. «High — Level Synthesis: Introduction to Chip and System Design». In: 1992.
- [10] W. Meeus et al. «An overview of today's high-level synthesis tools». In: *Design Automation for Embedded Systems* 16 (2012), pp. 31–51.
- [11] L.H. Crockett et al. *The Zynq Book: Embedded Processing With the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC*. Strathclyde Academic Media, 2014. ISBN: 9780992978709. URL: <https://books.google.it/books?id=9dfvoAEACAAJ>.
- [12] *International Technology Roadmap for Semiconductors*. https://www.dropbox.com/sh/r51qrus06k6ehrc/AACQYSRnTdLGUCDZfhB6_iXua/2011Chapters?dl=0&preview=2011Design.pdf. 2011.
- [13] J.P. Elliott. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Springer US, 2012. ISBN: 9781461550594. URL: <https://books.google.it/books?id=IIrkBwAAQBAJ>.
- [14] P. Coussy e A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer Netherlands, 2008. ISBN: 9781402085888. URL: <https://books.google.it/books?id=1EWRGB5J1HkC>.
- [15] H. Selvaraj, L. Daoud e D. Zydek. «A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing». In: vol. 240. Set. 2013. ISBN: 978-3-319-01856-0. DOI: [10.1007/978-3-319-01857-7_47](https://doi.org/10.1007/978-3-319-01857-7_47).

- [16] L. Hili. «Hardware Design using High-Level synthesis». In: *Electronic System-Level (ESL) Day*. September 2011.
- [17] Y. Pan et al. «A user's reflections on the art of high level synthesis». In: *2014 International Symposium on Integrated Circuits (ISIC)*. 2014, pp. 67–70. DOI: [10.1109/ISICIR.2014.7029585](https://doi.org/10.1109/ISICIR.2014.7029585).
- [18] P. Coussy et al. «An Introduction to High-Level Synthesis». In: *Design and Test of Computers, IEEE* 26 (set. 2009), pp. 8–17. DOI: [10.1109/MDT.2009.69](https://doi.org/10.1109/MDT.2009.69).
- [19] G. Martin e G. Smith. «High-Level Synthesis: Past, Present, and Future». In: *IEEE Design Test of Computers* 26.4 (2009), pp. 18–25. DOI: [10.1109/MDT.2009.83](https://doi.org/10.1109/MDT.2009.83).
- [20] Gary Smith EDA. <https://www.garysmitheda.com/research/>.
- [21] S. Sinha e T. Srikanthan. «High-Level Synthesis: Boosting Designer Productivity and Reducing Time to Market». In: *IEEE Potentials* 34.4 (2015), pp. 31–35. DOI: [10.1109/MPOT.2013.2292957](https://doi.org/10.1109/MPOT.2013.2292957).
- [22] D.D. Gajski e L. Ramachandran. «Introduction to high-level synthesis». In: *IEEE Design Test of Computers* 11.4 (1994), pp. 44–54. DOI: [10.1109/54.329454](https://doi.org/10.1109/54.329454).
- [23] B.C. Schafer e Z. Wang. «High-Level Synthesis Design Space Exploration: Past, Present, and Future». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 2628–2639. DOI: [10.1109/TCAD.2019.2943570](https://doi.org/10.1109/TCAD.2019.2943570).
- [24] S. Lahti et al. «Are We There Yet? A Study on the State of High-Level Synthesis». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (2019), pp. 898–911. DOI: [10.1109/TCAD.2018.2834439](https://doi.org/10.1109/TCAD.2018.2834439).
- [25] A. Takach. «High-Level Synthesis: Status, Trends, and Future Directions». In: *IEEE Design Test* 33.3 (2016), pp. 116–124. DOI: [10.1109/MDAT.2016.2544850](https://doi.org/10.1109/MDAT.2016.2544850).
- [26] A. Takach. «Design and verification using high-level synthesis». In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, pp. 198–203. DOI: [10.1109/ASPDAC.2016.7428011](https://doi.org/10.1109/ASPDAC.2016.7428011).
- [27] J. Cong et al. «High-Level Synthesis for FPGAs: From Prototyping to Deployment». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491. DOI: [10.1109/TCAD.2011.2110592](https://doi.org/10.1109/TCAD.2011.2110592).
- [28] A. Mathur et al. «Functional Equivalence Verification Tools in High-Level Synthesis Flows». In: *IEEE Design Test of Computers* 26.4 (2009), pp. 88–95. DOI: [10.1109/MDT.2009.79](https://doi.org/10.1109/MDT.2009.79).
- [29] S. Liu, F.C.M. Lau e B.C. Schafer. «Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration». In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.
- [30] W. Stoye et al. «Using RTL-to-C++ translation for large SoC concurrent engineering: A case study». In: *Electronics Systems and Software* 1 (mar. 2003), pp. 20–25. DOI: [10.1049/ess:20030104](https://doi.org/10.1049/ess:20030104).
- [31] K. Keutzer et al. «System-level design: orthogonalization of concerns and platform-based design». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.12 (2000), pp. 1523–1543. DOI: [10.1109/43.898830](https://doi.org/10.1109/43.898830).
- [32] M.R. Barbacci. «Instruction set processor specifications (ISPS): The notation and its applications». In: *IEEE Transactions on Computers* C-30.1 (1981), pp. 24–40. DOI: [10.1109/TC.1981.6312154](https://doi.org/10.1109/TC.1981.6312154).
- [33] D. Ku e G. Demicheli. «HardwareC – A Language for Hardware Design (Version 2.0)». In: 1990.

- [34] D. Gajski, T. Austin e S. Svoboda. «What input-language is the best choice for high level synthesis (HLS)?» In: *Design Automation Conference*. 2010, pp. 857–858. DOI: [10.1145/1837274.1837489](https://doi.org/10.1145/1837274.1837489).
- [35] J. Sanguinetti. «A Different View: Hardware Synthesis from SystemC is a Maturing Technology». In: *IEEE Design Test of Computers* 23.5 (2006), pp. 387–387. DOI: [10.1109/MDT.2006.111](https://doi.org/10.1109/MDT.2006.111).
- [36] S.A. Edwards. «The challenges of hardware synthesis from C-like languages». In: *Design, Automation and Test in Europe*. 2005, 66–67 Vol. 1. DOI: [10.1109/DATe.2005.307](https://doi.org/10.1109/DATe.2005.307).
- [37] Jianwen Zhu e D.D. Gajski. «Compiling SpecC for simulation». In: *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*. 2001, pp. 57–62. DOI: [10.1109/ASPDAC.2001.913281](https://doi.org/10.1109/ASPDAC.2001.913281).
- [38] *Vivado Design Suite User Guide*. Xilinx.
- [39] Y. Liang et al. «High-Level Synthesis: Productivity, Performance, and Software Constraints». In: *Journal of Electrical and Computer Engineering* 2012 (feb. 2012). DOI: [10.1155/2012/649057](https://doi.org/10.1155/2012/649057).
- [40] *Algorithmic C (AC) Datatypes*. <https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/#data-type>.
- [41] *Accelera System Initiative*. <https://www.accelera.org/>.
- [42] R. Nane et al. «A Survey and Evaluation of FPGA High-Level Synthesis Tools». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: [10.1109/TCAD.2015.2513673](https://doi.org/10.1109/TCAD.2015.2513673).
- [43] *KiwiC Compiler*. <https://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/manual/kiwic/node3.html>.
- [44] S. Ravi e M. Joseph. «Open source HLS tools: A stepping stone for modern electronic CAD». In: *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. 2016, pp. 1–8. DOI: [10.1109/ICCIC.2016.7919615](https://doi.org/10.1109/ICCIC.2016.7919615).
- [45] H. Ren. «A brief introduction on contemporary High-Level Synthesis». In: *2014 IEEE International Conference on IC Design Technology*. 2014, pp. 1–4. DOI: [10.1109/ICICDT.2014.6838614](https://doi.org/10.1109/ICICDT.2014.6838614).
- [46] Intel. *Intel FPGA SDKFOR OpenCL*. <https://www.intel.com.au/content/www/au/en/software/programmable/sdk-for-opencl/overview.html>.
- [47] Intel. *Intel HLS Compiler*. <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html>.
- [48] Mentor Graphics. *Catapult High-Level Synthesis*. <https://eda.sw.siemens.com/en-US>.
- [49] LegUp: *Software IDE for Programming Microchip FPGAs*. <https://www.legupcomputing.com/>.
- [50] *Bambu: A Free Framework for the High-Level Synthesis of Complex Applications*. https://panda.dei.polimi.it/?page_id=31.
- [51] *CtoS User Manual*. Cadence.
- [52] K Jaya Sampath et al. «An efficient Channel Coding Architecture for 5G Wireless using High-Level Synthesis». In: *2021 5th International Conference on Trends in Electronics and Informatics (ICOEI)*. 2021, pp. 674–680. DOI: [10.1109/ICOEI51242.2021.9453001](https://doi.org/10.1109/ICOEI51242.2021.9453001).
- [53] F.D. Robinson et al. «High-level synthesis for medical image processing on Systems on Chip: A case study». In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–2. DOI: [10.1109/FPL.2016.7577390](https://doi.org/10.1109/FPL.2016.7577390).
- [54] K. Fukava et al. «Design and Implementation of Ultra-Low-Latency Video Encoder Using High-Level Synthesis». In: *2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*. 2019, pp. 1–2. DOI: [10.1109/ISPACS48206.2019.8986365](https://doi.org/10.1109/ISPACS48206.2019.8986365).

- [55] D.T. Nguyen, V.B. Dang e K. Gaj. «A High-Level Synthesis Approach to the Software/Hardware Codesign of NTT-Based Post-Quantum Cryptography Algorithms». In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 371–374. DOI: [10.1109/ICFPT47387.2019.00070](https://doi.org/10.1109/ICFPT47387.2019.00070).
- [56] F. Farahmand et al. «Software/Hardware Codesign of the Post Quantum Cryptography Algorithm NTRUEncrypt Using High-Level Synthesis and Register-Transfer Level Design Methodologies». In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 225–231. DOI: [10.1109/FPL.2019.00042](https://doi.org/10.1109/FPL.2019.00042).
- [57] Y. Chi et al. *Extending High-Level Synthesis for Task-Parallel Programs*. 2021. arXiv: [2009.11389 \[cs.AR\]](https://arxiv.org/abs/2009.11389).
- [58] F. Winterstein, S. Bayliss e G. A. Constantinides. «High-level synthesis of dynamic data structures: A case study using Vivado HLS». In: *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 362–365. DOI: [10.1109/FPT.2013.6718388](https://doi.org/10.1109/FPT.2013.6718388).
- [59] A.M. Cabrera et al. «Toward Evaluating High-Level Synthesis Portability and Performance between Intel and Xilinx FPGAs». In: *International Workshop on OpenCL*. IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: [10.1145/3456669.3456699](https://doi.org/10.1145/3456669.3456699). URL: <https://doi.org/10.1145/3456669.3456699>.
- [60] *Porting Vivado HLS designs to Catapult HLS Platform*. <https://webinars.sw.siemens.com/porting-vivado-hls-designs-to/room>.
- [61] C. Pilato et al. «Securing Hardware Accelerators: A New Challenge for High-Level Synthesis». In: *IEEE Embedded Systems Letters* 10.3 (2018), pp. 77–80. DOI: [10.1109/LES.2017.2774800](https://doi.org/10.1109/LES.2017.2774800).
- [62] E.J. Folgo. *Accelerating Time-to-Market in the Global Electronics Industry*. <https://dspace.mit.edu/bitstream/handle/1721.1/43827/262620915-MIT.pdf?sequence=2&isAllowed=y>.