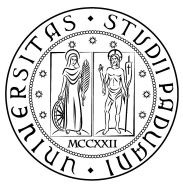


Università Degli Studi Di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Circuiti Asincroni:

Dai Principi Fondamentali all'Implementazione

13.7.2010

Relatore – Dott. Carlo Fantozzi

Laureando – Fabrizio Romano

a mio padre
a mia madre

Indice

Sommario

La maggioranza dei circuiti commercializzati al giorno d'oggi è di tipo sincrono. Negli ultimi anni però, questa tecnologia si è trovata a dover affrontare notevoli problemi legati al consumo di potenza e alle crescenti difficoltà di gestione del clock, in circuiti sempre più piccoli e densi.

Per ovviare a queste problematiche, che richiedono soluzioni tecnicamente complesse e dispendiose, i costruttori stanno portando l'attenzione sull'approccio asincrono che, privo di clock, promette di ridurre i consumi e velocizzare i circuiti. La mancanza di esperienza, strumenti e motivazioni adeguate rende però molto difficile una migrazione totale da un paradigma all'altro. La tecnologia che sembra destinata a prendere piede in questo contesto è quindi l'approccio ibrido Globally Asynchronous, Locally Synchronous. Importanti produttori sono impegnati nella ricerca in questo settore, che è ancora in piena fase evolutiva.

Il presente lavoro è diviso in due parti: nella prima offriremo un quadro generale sui fondamenti della tecnologia asincrona e, nella seconda, vedremo esempi di design che rappresentano l'attuale stato dell'arte.

Parte I.

Introduzione e Nozioni Fondamentali

1. Introduzione

In questo breve capitolo introduttivo intendiamo presentare al lettore, senza entrare troppo in dettaglio, un quadro generale che illustri le principali caratteristiche della tecnologia che sta alla base della progettazione di circuiti asincroni. Offriremo un insieme di motivazioni che giustificano l'interesse dimostrato, soprattutto negli ultimi anni, da ricercatori e case produttrici verso questa tecnologia che si dimostra essere così diversa dalla controparte sincrona. Vedremo quali sono i vantaggi e gli svantaggi che derivano dall'impiego di un clock, le principali caratteristiche strutturali e operazionali dei circuiti asincroni e daremo uno sguardo ai recenti miglioramenti nelle prestazioni e alle varie problematiche che questa tecnologia si trova a dover affrontare.

Concluderemo il capitolo con alcune considerazioni di tipo tecnico e con l'esposizione di come il lavoro è stato organizzato.

1.1. I circuiti asincroni[(1)]

La grande maggioranza dei circuiti digitali prodotti al giorno d'oggi e presenti in massa sul mercato è di tipo sincrono. Questa tecnologia si basa su due assunzioni fondamentali: (1) tutti i segnali sono binari, e (2) tutte le componenti condividono una nozione comune e discreta di *tempo*, definito da un segnale di clock distribuito lungo tutto il circuito.

Il principale beneficio apportato dalle suddette assunzioni è un'enorme semplificazione del lavoro di progettazione e design del circuito: il clock stabilisce delle limitazioni ben precise che interessano il funzionamento di ogni componente e limitano la libertà del progettista riducendo il numero delle potenziali scelte che può operare, semplificandogli il lavoro.

La tecnologia asincrona invece aderisce solamente alla prima assunzione, cioè tutti i segnali elaborati sono binari, ma non possiede un segnale di clock e di conseguenza non vi è, all'interno del circuito, una nozione intrinseca di *tempo*, comune e discreto, condiviso tra le varie componenti.

Per portare a termine compiti quali la sincronizzazione, la comunicazione e la sequenzialità delle operazioni, i circuiti asincroni utilizzano quindi un diverso meccanismo, chiamato *handshaking*, che si basa sulla comunicazione (via *request* e *acknowledgements*) tra le varie componenti. Questa fondamentale differenza dona ai circuiti asincroni delle proprietà che possono essere sfruttate per ottenere risultati migliori, rispetto alle controparti sincrone, in molteplici aree.

Lo sviluppo dei circuiti asincroni risale agli anni '50, ma a quel tempo i progettisti decisero di impiegare i loro sforzi nello sviluppo della tecnologia sincrona, considerata

[(1)] Con il termine *circuiti* intendiamo in senso ampio: circuiti, chip, processori.

più prestante, più robusta e di più semplice progettazione. Recentemente, comunque, la tecnologia asincrona ha svolto importanti passi avanti che hanno incrementato notevolmente le prestazioni e l'affidabilità dei circuiti, eliminando così uno dei principali ostacoli alla loro produzione in massa.

Importanti gruppi come il *California Institute of Technology's Asynchronous VLSI Group* e lo *University of Manchester's Amulet Project* hanno dedicato molto tempo allo studio di questo soggetto e, nonostante le incertezze iniziali e qualche progetto naufragato, al giorno d'oggi anche aziende del calibro della *Sun Microsystems* (ora acquisita dalla *Oracle*), la *Fulcrum Microsystems* e la *Theseus Logic* stanno cercando di produrre un chip asincrono per il commercio. Nonostante ciò, i circuiti asincroni continuano a generare preoccupazioni e incertezze, motivate soprattutto dalla mancanza di strumenti ed esperienza adeguati per la loro progettazione e alla difficoltà di interfacciare un circuito asincrono con uno sincrono. Queste problematiche andranno risolte prima di poter pensare alla produzione di massa.

1.2. Vantaggi e svantaggi nell'uso del clock

Il clock è un segnale generato tipicamente da un oscillatore al quarzo che vibra con una frequenza regolare, espressa comunemente in megahertz o gigahertz, che dipende dal voltaggio applicato. Tutto il lavoro compiuto dal circuito è sincronizzato dal clock, che invia il suo segnale lungo tutta l'estensione del circuito e controlla i registri, il flusso dei dati e l'ordine con cui un processore, ad esempio, esegue i vari compiti che gli sono assegnati.

Un vantaggio dei circuiti sincroni è che l'ordine di arrivo dei dati non ha alcuna importanza, a patto ovviamente che arrivino tutti nel medesimo ciclo di clock: il registro aspetta il fronte di salita successivo prima di catturarli e quindi il sistema può processarli correttamente. I designer quindi non si devono preoccupare eccessivamente di problematiche come ad esempio la lunghezza dei collegamenti, quando lavorano sui chip.

Inoltre, è estremamente semplice calcolare le prestazioni di un circuito, in quanto è sufficiente contare il numero di cicli di clock necessari per svolgere una data operazione per sapere quante operazioni di quel tipo è possibile eseguire in un dato periodo di tempo. Questo è di fondamentale importanza soprattutto ai fini del marketing.

Ci sono però alcuni svantaggi, dovuti proprio alla presenza di un segnale di clock.

Nei circuiti molto estesi il clock deve riuscire ad attraversare tutto il circuito entro un ciclo, e questo pone un limite superiore alla frequenza massima cui può lavorare. Un altro problema è legato al fatto che alcune parti del circuito impiegano molto meno di un ciclo di clock per svolgere un determinato compito, ma devono comunque aspettare il ciclo seguente per poter procedere con le operazioni.

Per ovviare a queste problematiche, i designer hanno adottato diverse soluzioni che si sono rivelate complicate e alquanto costose. Ad esempio è possibile implementare gerarchie di bus, oppure modellare il circuito in modo che possa regolare diverse letture del clock su diversi componenti. E' anche possibile far sì che componenti individuali del circuito possiedano un loro clock isolato e comunichino tra di loro attraverso il bus.

Nonostante tutti questi accorgimenti però, rimane il fatto che il clock consuma un elevato quantitativo di potenza e produce calore; senza contare che i registri impiega-

no una notevole quantità di energia per commutare ed essere così pronti per ricevere i dati ogni qualvolta il clock emetta un tick, e questo viene fatto anche quando non c'è nessun dato su cui operare, con evidente spreco di energia.

1.3. Funzionamento dei circuiti asincroni

Al giorno d'oggi esistono pochissime soluzioni puramente asincrone e, di queste, solo una minima parte è stata commercializzata. Esistono però diverse soluzioni ibride, come ad esempio le architetture GALS[(2)] che mettono in comunicazione in modo asincrono delle micro-componenti sincrone, evitando in questo modo le problematiche che il circuito avrebbe se fosse puramente sincrono. Il passaggio dei dati da un modulo all'altro è gestito via handshaking. Il processore asincrono inserisce l'ubicazione dei dati che vuole leggere sull'address bus e lancia una richiesta. La memoria legge l'indirizzo sul bus, recupera le informazioni e le inserisce nel data bus. La memoria poi manda un segnale di acknowledge per informare il processore che ha letto le informazioni e a quel punto il processore le recupera dal data bus. La gestione di questi segnali è affidata a delle pipeline e a dei sequencer FIFO che permettono così di preservare l'ordine delle richieste.

Questo metodo di handshake consuma un quantitativo di potenza maggiore di quello consumato da un segnale di clock, ma ciò risulta comunque vantaggioso, perché le varie componenti consumano potenza solamente quando hanno un effettivo lavoro da eseguire.

Un altra caratteristica dei circuiti asincroni è che i dati non si muovono tutti assieme come invece avviene nei processori sincroni. Questo porta alcuni benefici: (1) l'ampiezza e la frequenza dei picchi di tensione (dovuti allo spostamento dei dati) è molto ridotta e (2) sono ridotte le radiazioni elettromagnetiche (EMI). Una minore emissione di radiazioni elettromagnetiche comporta una riduzione degli errori dovuti al rumore e una minore interferenza con i circuiti nelle vicinanze.

La mancanza del clock inoltre, assieme alla caratteristica che il circuito lavora solo quando effettivamente deve svolgere dei compiti, si trasformano in una minore richiesta energetica perché viene impiegato solamente il voltaggio necessario per eseguire una data operazione. In questo senso, i circuiti asincroni sono particolarmente adatti per la riproduzione di video, audio e altre applicazioni in streaming. Ciò è dovuto principalmente al fatto che le applicazioni in streaming, nonostante siano interessate da un flusso di dati pressoché regolare, presentano in vari istanti (che possiamo chiamare *tempi morti*) variazioni nel flusso audio/video di minima entità e ciò richiede poco lavoro in termini di correzione degli errori.

I circuiti asincroni quindi, attivando solamente quelle parti che sono interessate da una richiesta di lavoro e lasciando a riposo (e allo stesso tempo pronte alla risposta) tutte le componenti che non devono contribuire all'elaborazione dei dati, sono interessati da un minore riscaldamento e producono meno picchi di tensione. Secondo il Professor Steve Furber, direttore del progetto Amulet dell'università di Manchester (si veda cap. 11 a pagina 133), sono anche più affidabili, perché il meccanismo di

[(2)]GALS: Globally Asynchronous Locally Synchronous.

handshaking lascia il tempo ai dati di arrivare e di stabilizzarsi prima che il circuito prosegua e questo non sempre succede nei circuiti sincroni, specialmente in quelle situazioni in cui i dati arrivano giusto in tempo (cioè subito prima di un nuovo tick del clock) e devono immediatamente ripartire.

Un altro aspetto da tenere in considerazione è che nei circuiti asincroni la comunicazione tramite handshaking non forza il circuito a lavorare sul caso peggiore. Se ad esempio un dato deve attraversare una cascata di porte logiche per essere elaborato, giungerà al termine in un tempo che tipicamente sarà la somma dei tempi medi di elaborazione di ciascuna porta facente parte della cascata. In un circuito sincrono questo non si verifica, perché serve la certezza che il dato in uscita da una cascata di porte logiche sia stato elaborato correttamente e sia stabile, e quindi è necessario aspettare sempre un tempo pari alla somma dei tempi massimi (cioè associati al caso peggiore) di ciascuna porta.

Anche il processo di design risulta semplificato dalla modularità del circuito e inoltre, poiché i dati sono trasportati da un modulo all'altro in modo asincrono, è possibile accoppiare con facilità delle componenti che funzionano a velocità differenti.

1.4. Recenti miglioramenti nelle prestazioni

Tradizionalmente i tempi di elaborazione dei circuiti asincroni non hanno mai retto il paragone con la loro controparte sincrona, nonostante l'elaborazione operi su tempi proporzionali al caso medio e non al caso peggiore. Questo divario di prestazioni trae origine dal fatto che per implementare i circuiti asincroni venivano impiegati solitamente i transistori di tipo p (più lenti e di maggiori dimensioni) in uno schema di logica combinatoriale. Il recente utilizzo della *domino logic* e della *delay insensitive mode*, però, ha dato origine ad un approccio molto più prestante, conosciuto col nome di *integrated pipeline mode*. La *domino logic* migliora le prestazioni perché un sistema può elaborare diverse linee di dati contemporaneamente in un ciclo, mentre l'approccio tipico limitava il lavoro ad una sola linea per ciclo. La *domino logic* è inoltre efficiente perché agisce solamente su quei dati che sono mutati durante il processo, invece di operare su tutti i dati indistintamente. La *delay-insensitive mode* dona, in aggiunta, la possibilità a ciascun blocco di lavorare con un delay arbitrario. In questo modo i registri comunicano alla massima velocità comune e quando un blocco è lento, influenza solo i blocchi con cui comunica. Questo dà al sistema il tempo di convalidare i dati prima di trasmetterli, riducendo gli errori.

1.5. Problematiche

I circuiti asincroni si trovano a dover affrontare due problematiche in particolare.

Innanzitutto è necessario rendere possibile un interfacciamento tra circuiti di diversa natura (sincroni e asincroni). Realizzare ciò è un compito complesso perché, diversamente da quelli sincroni, i circuiti asincroni non completano le istruzioni in tempi regolati da un clock e questo implica difficoltà di comunicazione via bus e problemi di gestione ad esempio di memorie condivise. Sono necessari inoltre dei sistemi che sincronizzino in qualche modo l'arrivo dei dati da un circuito asincrono a uno

sincrono, in modo che quest'ultimo possa utilizzarli correttamente come se fossero arrivati in modo sincrono, e ciò nella pratica è un compito di elevata difficoltà e sovente può portare a errori.

La seconda problematica fondamentale è la mancanza di strumenti dedicati alla progettazione, come i CAD o gli strumenti per la codifica. Questo comporta che i progettisti debbano inventare strumenti nuovi (o adattare quelli esistenti che però sono pensati per la progettazione di circuiti sincroni) e questo richiede notevoli sforzi e tempo, rendendo la progettazione più costosa.

Solo ultimamente le compagnie interessate in questa tecnologia, stanno cominciando a produrre strumenti pensati per lo sviluppo e la progettazione dei circuiti asincroni. La *HandShake*, ad esempio, utilizza un proprio linguaggio di programmazione (*Haste*), così come i *Philips Research Laboratories* utilizzano un proprio compilatore (*Tangram*). La *Manchester University* ha prodotto il *Balsa Asynchronous Synthesis System* e la *Silistix Ltd.* commercializza strumenti per il clockless-design.

Manca, infine, anche l'esperienza nel design asincrono perché le opportunità di lavorare su progetti di questo tipo sono scarse e le università non forniscono che pochissimi corsi sull'argomento.

1.6. Alcune considerazioni

Al momento è considerato improbabile che una compagnia possa produrre un circuito asincrono nell'immediato futuro. E' molto più probabile invece che possano svilupparsi delle soluzioni ibride, che si avvantaggino dei punti di forza di entrambi questi due mondi. Il parere del professor Chris Myers, dell'università dello Utah, è che l'industria si muoverà gradualmente verso le implementazioni GALS: moduli sincroni opereranno indipendentemente l'uno dall'altro a differenti velocità di clock, comunicando tra loro per mezzo del protocollo handshake. In questo modo la distribuzione dei vari clock rimarrà confinata a piccole aree del circuito, evitando le problematiche che derivano in caso contrario.

Riassumendo, i vantaggi sono:

- **Minore consumo di potenza**: dovuto alla mancanza del clock (o alla riduzione in molteplici clock a consumo minore) e al consumo zero in fase di stand-by
- **Alta velocità**: dovuta al fatto che la velocità è determinata dalle latenze effettive (caso medio) invece che da quelle riferite al caso peggiore.
- **Minore emissione di radiazioni elettromagnetiche (rumore)**: dovuta al fatto che la tensione di alimentazione è più bassa e che i vari clock tendono a emettere i fronti di salita in istanti casuali nel tempo.
- **Robustezza riguardo alle variazioni di tensione, temperatura e parametri del processo di fabbricazione**: dovuta al fatto che la temporizzazione è insensibile a rallentamenti (cosa che con un clock non succede)
- **Migliore componibilità e modularità**: dovuta al protocollo handshake e all'indipendenza dei singoli moduli (temporizzati localmente)

- **Niente distribuzione del clock e clock skew**: non serve progettare il circuito in modo che il segnale di clock possa essere distribuito uniformemente lungo tutto il circuito con minimi ritardi di fase.

Gli svantaggi invece sono:

- **Consumo di area e potenza**: implementare la logica di controllo handshake normalmente richiede più area, consuma più potenza e riduce la velocità del circuito(*).
- **Mancanza di esperienza e di strumenti per la progettazione e per il test e la validazione dei dati.**
- **Problemi di interfacciamento con circuiti sincroni.**

(*) Al lettore attento non sarà sfuggito che il consumo di potenza è elencato sia come vantaggio che come svantaggio. La ragione di questa apparente contraddizione è che l'implementazione della logica di controllo handshake è più onerosa in termini di area e di consumo rispetto a quella delle normali linee di distribuzione di un segnale di clock. Il motivo per cui i circuiti asincroni sono però comunque in grado di consumare meno potenza risiede nel fatto che (1) non lavorano costantemente e (2) solo quelle parti del circuito che sono effettivamente interessate dalle operazioni in corso sono attive, mentre il resto viene lasciato a riposo. Il bilancio complessivo dei consumi per queste due modalità è comunque a vantaggio dei circuiti asincroni.

1.7. Organizzazione del lavoro

L'argomento "circuiti asincroni" preso nel suo insieme è, come si può facilmente immaginare, molto vasto e quindi non è pensabile poterlo trattare tutto (in modo esaustivo), in un singolo testo. D'altro canto, focalizzare la ricerca su una singola e ben precisa tematica, avrebbe tolto al lavoro quel minimo di carattere generale che è indispensabile ai fini dell'interesse del lettore. Per questi motivi il testo è suddiviso in due parti. Nella prima abbiamo scelto di esporre tutta una serie di caratteristiche, metodologie, applicazioni e considerazioni che hanno carattere generale e introduttivo, selezionando solo quelle che ricoprono un ruolo fondamentale o che sono comuni agli esempi, più specifici, trattati in seguito. Nella seconda parte invece esporremo, mediante una serie di esempi, l'attuale stato dell'arte di questa interessante tecnologia. Avremo cura di integrare nel testo, quando necessario, anche quelle parti di carattere introduttivo che non abbiamo ritenuto opportuno includere nella prima parte del testo per via della minore importanza o diffusione, in modo da fornire al lettore i presupposti per un'adeguata comprensione di ogni esempio riportato.

Così facendo, speriamo di aver dato al testo un carattere che non sia né troppo generico, né troppo specifico quanto, piuttosto, una giusta via di mezzo tra estensione e profondità.

Una parola sulla traduzione: come si può notare leggendo la bibliografia, il materiale da cui abbiamo attinto è tutto in lingua inglese. Tradurre i termini tecnici dall'inglese all'italiano non è sempre facile o possibile. Non tutti i termini possiedono una corretta traduzione e alcuni obbligano a ricorrere a parafrasi eccessive. Si è cercato anche in questo caso di raggiungere un buon compromesso perciò, ad esempio,

sarà possibile trovare nel testo la parola “protocollo” usata al posto dell’inglese “protocol” perché ciò non comporta alcun problema nell’esposizione e non varia il significato originale del termine. Non abbiamo ritenuto opportuno invece tradurre “handshake protocol” con “protocollo *a stretta di mano*” e ci auguriamo che il lettore si trovi d’accordo con noi in questa scelta.

2. Fondamentali

In questo capitolo cercheremo di esporre alcuni concetti fondamentali che riguardano soprattutto gli aspetti circuitali della tecnologia asincrona.

In particolare vedremo il protocollo di comunicazione handshake e le sue diverse implementazioni, il C-Element e le pipeline di Muller, gli stili di implementazione dei circuiti e una loro breve classificazione, assieme a qualche altro concetto minore ma che vale comunque la pena di accennare.

2.1. Il protocollo di comunicazione Handshake

L'implementazione di un tipo di comunicazione basata su segnali *send* e *receive* è fondamentale per la logica asincrona in quanto viene utilizzata a ogni livello di progetto del sistema: dalla gestione della comunicazione tra un processore e una memoria cache fino, ad esempio, all'interazione tra una parte di controllo e un datapath di una ALU. La comunicazione mediante due canali che interconnettono due componenti asincrone è implementata quindi come un protocollo handshake.

In questa sezione esporremo le implementazioni più adottate nella pratica.

2.1.1. Bundled-data protocols

Il termine *bundled-data* si riferisce ad una situazione in cui i segnali relativi ai dati utilizzano dei comuni livelli booleani per codificare l'informazione e le linee (separate) che sono dedicate ai *request* e agli *acknowledge* sono impacchettate (bundled) con i dati (figura 2.1(a)).

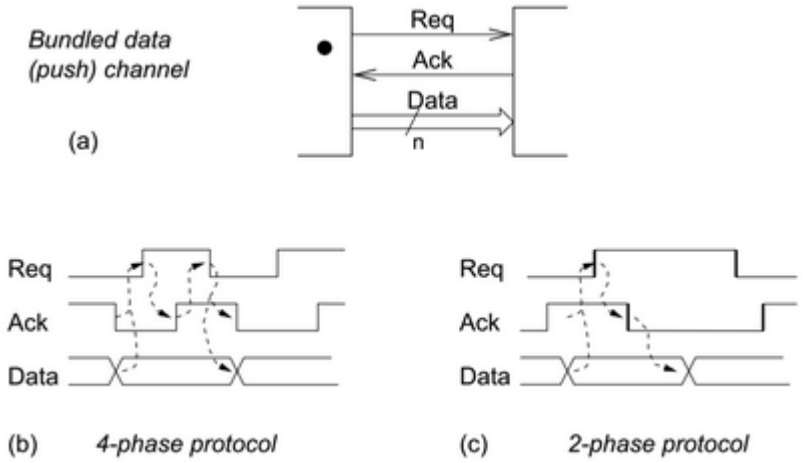


Figura 2.1 – (a) Un canale di tipo bundled-data. (b) Un 4-phase bundled-data protocol. (c) Un 2-phase bundled-data protocol.

Nel protocollo *4-phase* illustrato in figura 2.1(b) anche le linee di request e acknowledge utilizzano normali livelli booleani per codificare l'informazione e il termine

4-phase (4 fasi) è riferito proprio al numero di azioni compiute dal meccanismo di comunicazione:

1. Il trasmettitore emette dei dati e setta il canale di request sul livello 1 (*high*)[1]
2. Il ricevitore assorbe i dati e setta il canale di acknowledge sul livello 1 (*high*)
3. Il trasmettitore risponde settando il canale di request sul livello 0 (*low*), e a quel punto non vi è più garanzia sulla validità dei dati
4. Il ricevitore comunica il buon esito dell'operazione settando il canale di acknowledge sul livello 0 (*low*)

A questo punto il trasmettitore può cominciare un nuovo ciclo di comunicazione.

Il protocollo 4-phase è familiare alla maggior parte di designer di circuiti digitali, ma presenta lo svantaggio delle transizioni di ritorno a livello basso (*return-to-zero transitions*) che sprecano tempo ed energia senza che ce ne sia necessità.

Il protocollo *2-phase* mostrato in figura 2.1(c) elimina questo problema. L'informazione sui canali di request e acknowledge è codificata come una transizione di segnale sulla linea e quindi non c'è differenza tra una transizione di tipo $0 \rightarrow 1$ e una di tipo $1 \rightarrow 0$ ($basso \rightarrow alto$, $alto \rightarrow basso$). Entrambe rappresentano un evento di segnalazione. Idealmente il protocollo di tipo bundled-data a 2 fasi dovrebbe produrre circuiti più veloci rispetto a quello a 4 fasi ma, spesso, l'implementazione di circuiti che siano in grado di rispondere agli eventi è complessa e non vi è una generica presa di posizione su quale delle due opzioni sia la migliore.

> Alcuni chiarimenti sulla terminologia:
>
> Al posto del termine *bundled-data*, alcuni testi usano il termine *single-rail* (forse più adatto quando contrapposto al termine *dual-rail*, che vedremo più avanti nel capitolo). Il termine *bundled-data* enfatizza le relazioni che intercorrono tra i segnali pertinenti ai dati e quelli pertinenti alle comunicazioni, mentre il termine *single-rail* enfatizza l'uso di una singola linea per trasportare un singolo bit di dati.
> Al posto del termine *4-phase handshaking* (o *signaling*) alcuni testi utilizzano i termini *return-to-zero* (RTZ) *signaling* o *level signaling* e al posto del termine *2-phase handshaking* (o *signaling*) alcuni testi utilizzano i termini *non-return-to-zero* (RTZ) *signaling* o *transition-signaling*. Di conseguenza, parlare di *return-to-zero single-track protocol* equivale a parlare di *4-phase bundled-data protocol*, e così via.

I protocolli introdotti fin qui, assumono che il trasmettitore costituisca la parte attiva, cioè quella che inizia il trasferimento dei dati sul canale. Questa configurazione è conosciuta come *push channel*. L'opposto, cioè il ricevitore che richiede nuovi dati, è altrettanto possibile e dà origine ad una configurazione chiamata *pull channel*. In questo caso le direzioni dei segnali di request e acknowledge sono invertite rispetto alla configurazione *push channel* e la validità dei dati è indicata dal segnale di acknowledge che transita dal trasmettitore al ricevitore. Nei diagrammi astratti di

[1] I termini high/low, alto/basso sono riferiti ai livelli 1/0.

circuiti che rappresentano collegamenti e/o canali con dei simboli, è sovente trovare un punto che segnala la parte attiva del canale (come mostrato in figura 2.1(a)).

Per completare il quadro si possono menzionare alcune varianti: (1) un canale senza traffico dati, usato per le sincronizzazioni (a volte viene chiamato *bare-handshake*), e (2) un canale dove i dati fluiscono in entrambe le direzioni e dove segnali di *req* e *ack* indicano la validità dei dati che sono scambiati tra il trasmettitore e il ricevitore. Quest'ultimo in particolare può essere utilizzato per interfacciarsi a una memoria read-only: così facendo, l'indirizzo viene impacchettato con il segnale *req* e i dati vengono impacchettati con il segnale *ack*.

Tutti i protocolli di tipo bundled-data si affidano al cosiddetto *delay matching*, in modo che l'ordine degli eventi (segnali) che partono dal trasmettitore sia preservato nel lato ricevitore. In un push-channel, i dati sono validi prima che il segnale di *req* sia settato sul livello alto (ciò si esprime formalmente scrivendo $Dati\,(validi) \prec Req$) L'ordinamento deve essere valido anche a lato ricevitore e ciò richiede particolare cura in fase di implementazione (fisica) del circuito.

Alcune soluzioni sono:

- Controllare il posizionamento e l'instradamento delle linee, possibilimente cercando di inviare tutti i segnali in un canale come un pacchetto. Questo è semplice in una struttura di tipo *tile-based datapath*.
- Avere un margine di sicurezza al terminale del trasmettitore.
- Inserire e/o ridimensionare i buffer dopo il layout (più o meno come si fa in fase di sintesi e layout con un CAD)

Un alternativa invece è quella di utilizzare un protocollo più sofisticato che non sia sensibile nei confronti dei delay sulle linee, e ciò è appunto quanto vedremo nelle prossime sezioni.

2.1.2. Il 4-phase dual-rail protocol

Il *4-phase dual-rail* protocol codifica il segnale request all'interno dei segnali associati ai dati utilizzando due linee per ogni bit di informazione che viene trasmesso (figura 2.2).

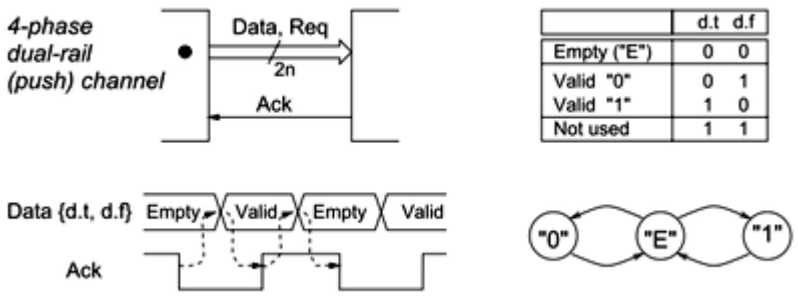


Figura 2.2 – Un canale Delay-Insensitive che sfrutta il 4-phase dual-rail protocol.

In sostanza, è un protocollo a 4 fasi che utilizza due linee *request* per ogni bit di informazione d. Una linea *d.t* è utilizzata per inviare il segnale logico 1 (o *true*) e un'altra linea *d.f* è utilizzata per inviare il segnale logico 0 (o *false*). Osservando un canale a un singolo bit, si può notare una sequenza di handshake dove il segnale di *request* che partecipa in ciascun handshake può essere *d.t* o *d.f*. Questo protocollo è molto robusto: due parti possono comunicare in modo affidabile senza curarsi dei ritardi (delay) presenti sulle linee che li connettono. Per questo motivo viene definito *delay-insensitive*.

Visti assieme, i segnali $\{x.f, x.t\}$ formano una parola di codice: $\{x.f, x.t\} = \{1, 0\}$ e $\{x.f, x.t\} = \{0, 1\}$ rappresentano dati "validi" (0 logico e 1 logico, rispettivamente) e $\{x.f, x.t\} = \{0, 0\}$ rappresenta lo stato "no data" o, in altri termini uno "spaziatore", valore "vuoto" o "NULL". La parola di codice $\{x.f, x.t\} = \{1, 1\}$ invece non viene utilizzata e una transizione da una configurazione valida a un'altra configurazione valida non è permessa, come illustrato in figura 2.2.

Questo ci porta a una visualizzazione più astratta del 4-phase handshaking: (1) il trasmettitore produce una valida parola di codice, (2) il ricevitore assorbe la parola e setta *acknowledge* sul livello 1 (o *high*), (3) il trasmettitore risponde inviando una parola di codice vuota ($\{0, 0\}$) e (4) il ricevitore comunica la ricezione di questo segnale abbassando al livello 0 (*low*) il segnale di *acknowledge*. A questo punto il trasmettitore può cominciare un altro ciclo di comunicazione. Una visione ancora più astratta di questo meccanismo si può interpretare come un flusso di parole di codice "valide" separate da parole di codice "vuote".

Proviamo ora a estendere questo approccio a canali composti da *N* bit paralleli. Un canale dati a *N* bit è formato dalla concatenazione di *N* coppie di linee, ciascuna delle quali utilizza la codifica descritta sopra. Un ricevitore è sempre in grado di capire quando tutti i bit sono validi (cosa che lo porta a settare il segnale di *acknowledge* sul livello 1) o quando tutti i bit sono vuoti (cosa che invece lo porta ad abbassare il segnale di *acknowledge* al livello 0). Questo è un meccanismo abbastanza intuitivo, ma trae comunque fondamento da basi matematiche.
Il codice *dual-rail* è un membro particolarmente semplice della famiglia dei codici *delay-insensitive* e possiede alcune favorevoli caratteristiche:

- Ogni concatenazione di parole di codice dual-rail è anch'essa una parola di codice dual-rail.
- Per un dato *N* (il numero dei bit che vengono comunicati) l'insieme di tutte le parole di codice realizzabili può essere suddiviso in 3 sottoinsiemi disgiunti:
  - {la parola di codice "vuota", dove tutte le *N* coppie di linee sono nello stato $\{0, 0\}$}
  - {le parole di codice intermedie, dove alcune linee assumono lo stato "vuoto" e altre lo stato "valido"}
  - {le 2^N parole di codice "valide"}

La figura 2.3 illustra un handshaking su un canale a *N* bit: un ricevitore vedrà una parola di codice "vuota", una sequenza di parole di codice intermedie (a mano a

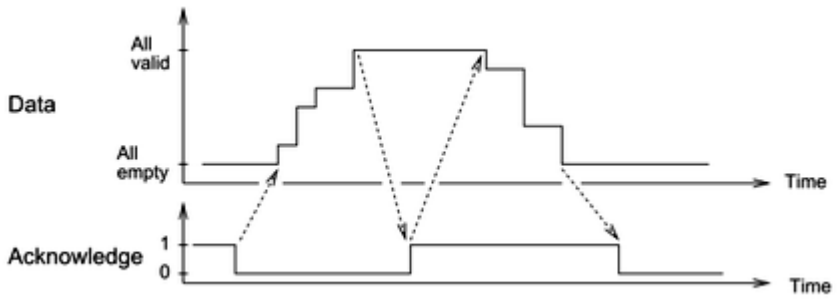


Figura 2.3 – Illustrazione di una fase di handshaking in un 4-phase dual-rail channel.

mano che alcune coppie di linee/bit diventano valide) e alla fine una parola di codice "valida". Dopo aver ricevuto la parola di codice e aver spedito un *acknowledge*, il ricevitore vedrà una sequenza di parole intermedie (a mano a mano che alcune coppie di linee/bit tornano a $\{0, 0\}$, cioè nello stato "vuoto") e alla fine una parola di codice "vuota", alla quale risponderà riabbassando al livello 0 il segnale di *acknowledge*.

2.1.3. Il 2-phase dual-rail protocol

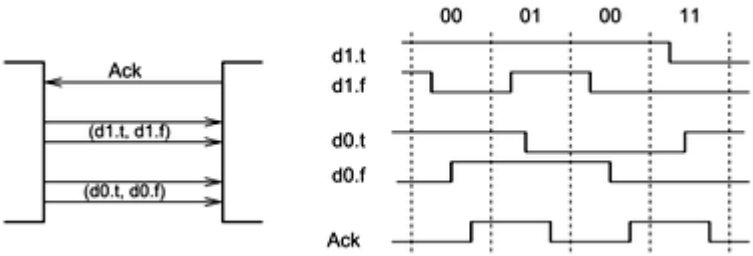


Figura 2.4 – Handshaking su un 2-phase dual-rail channel.

Il *2-phase dual-rail protocol* utilizza a sua volta 2 linee $\{d.t, d.f\}$ per bit, ma l'informazione è codificata come (eventi di) transizione, come spiegato in precedenza. In un canale a N bit, una nuova parola di codice viene ricevuta quando esattamente una linea in ciascuna delle N coppie ha eseguito una transizione. Non esiste la parola di codice "vuota" e un messaggio valido è confermato (da un *acknowledge*) e viene seguito da un altro messaggio valido (anch'esso confermato da un altro *acknowledge*). La figura 2.4 mostra le oscillazioni del segnale su un canale a 2 bit che sfrutta il *2-phase dual-rail protocol*.

2.2. Muller C-Element e pipeline

2.2.1. Il Muller C-element

In un circuito sincrono il ruolo del clock è di definire degli istanti temporali nei quali i segnali siano validi e stabili. Tra un tick e l'altro, il segnale può esibire variazioni di diversa natura e può anche attraversare svariate transizioni finché il circuito com-

binatorio non si stabilizza. Questo tipo di comportamento non crea nessun problema da un punto di vista funzionale.

Nei circuiti asincroni (di controllo) invece, la situazione è molto diversa. L'assenza del clock comporta che, in svariate circostanze, i segnali debbano essere validi lungo tutto un arco di tempo, le transizioni abbiano tutte un certo significato e, di conseguenza, queste oscillazioni e comportamenti bizzarri (che sono chiamati *hazards* e *races*) vanno evitati.

Considerando un circuito come una collezione[(2)] di porte che solitamente includono anche circuiti di feedback, sappiamo che quando l'output di una porta cambia, la sua variazione si propaga all'input della porta successiva e questa può a sua volta decidere se modificare il proprio output in base alla variazione dei segnali in ingresso.

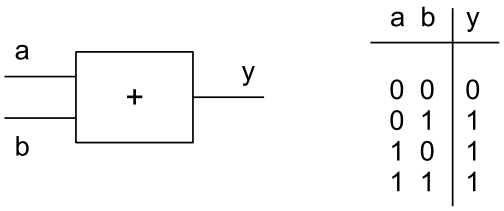


Figura 2.5 – Una porta logica OR.

Il concetto di indicazione (*indication*) o *acknowledge* gioca quindi un ruolo fondamentale nel progetto dei circuiti asincroni.

Considerando ad esempio la semplice porta *OR* di figura 2.5 notiamo che se l'output passa da 1 a 0, possiamo dedurre che *entrambi* gli input in quel momento sono a 0. Nel caso in cui l'output passi da 0 a 1, possiamo invece dedurre che almeno uno dei segnali sia passato a 1, ma non sappiamo dire quale. Per questo motivo possiamo dire che la porta *OR* indica (o *acknowledges*) solamente quando entrambi i segnali di input sono a 0. Attraverso considerazioni simili possiamo dire che una porta *AND* invece indica solo quando entrambi i segnali sono a 1, e così via. Transizioni di segnale che non sono "indicate" (o *acknowledged*) in altre transizioni di segnale sono pericolose sorgenti di *hazard* e dovrebbero essere evitate.

Un circuito che si comporta meglio rispetto a questo tipo di problematica è il cosiddetto *Muller C-element* (figura 2.6).

E' un elemento che trattiene lo stato, simile a un set-reset latch asincrono. Quando entrambi gli ingressi sono a 0 l'output è settato a 0, e quando entrambi gli ingressi sono a 1, l'output è settato a 1. Per le altre combinazioni di input ($\{1,0\}$, $\{0,1\}$) l'output non varia. Di conseguenza, un osservatore che vede l'output cambiare da 0 a 1 ne deduce che *entrambi* gli input si sono portati a 1 (e viceversa per lo stato 0).

Combinando quanto esposto con l'osservazione che tutti i circuiti asincroni si basano su handshake che comportano transizioni cicliche tra 0 e 1, risulta chiaro quanto sia fondamentale il *Muller C-element* (e perché quindi venga usato in modo così esteso).

[(2)]Collezione nel senso informatico del termine, cioè un insieme, un gruppo.

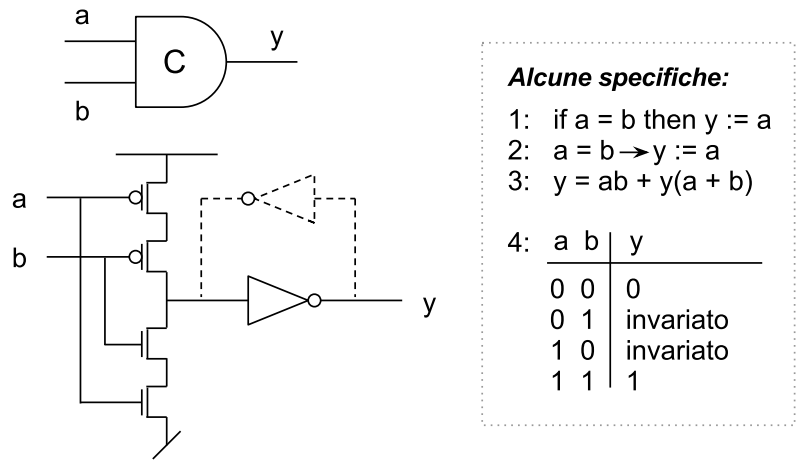


Figura 2.6 – Muller C-Element: simbolo, una possibile implementazione e alcune specifiche alternative.

2.2.2. Muller pipeline

In figura 2.7 possiamo notare un circuito costituito da C-element e inverter. Questo circuito è conosciuto col nome di *Muller pipeline* o *Muller distributor*. Variazioni ed estensioni di questo circuito formano la spina dorsale (della parte di controllo) di quasi tutti i circuiti asincroni. Ciò potrebbe non risultare così ovvio di primo acchito, ma se riusciamo ad andare oltre i dettagli, riusciremo quasi sempre a individuare la presenza della Muller pipeline in un circuito. La pipeline ha un comportamento simmetrico ed elegante che permette, una volta compreso a fondo, di disporre di una solida base grazie alla quale sarà possibile comprendere la maggior parte dei circuiti asincroni.

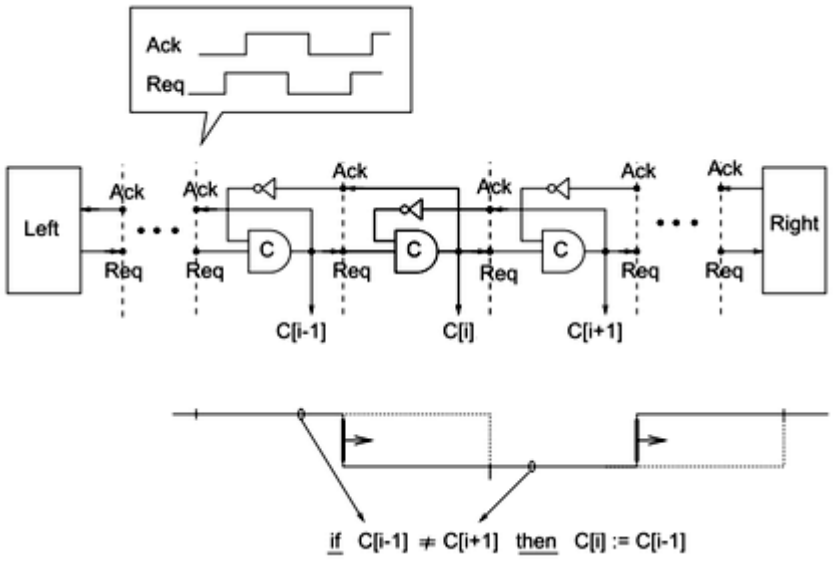


Figura 2.7 – Muller pipeline (o Muller distributor).

La pipeline in figura 2.7 è un meccanismo che ritrasmette gli handshake. Dopo che

tutti i C-element sono stati inizializzati a 0, la parte di sinistra (indicata con *Left*) può cominciare a operare gli handshake. Per capire il funzionamento, consideriamo $l'i^{mo}$ C-Element, $C[i]$. Questo elemento propagherà (cioè riceverà in input e lo registrerà) un 1 dal suo predecessore $C[i-1]$, solo se il suo successore $C[i+1]$ è a 0, e, allo stesso modo, propagherà uno 0 dal suo predecessore solo se il suo successore è a 1. E' utile pensare ai segnali che si propagano in un circuito asincrono come a una sequenza di onde, come illustrato nella parte inferiore di figura 2.7. Visto in quest'ottica, il ruolo di uno stadio di un C-Element nella pipeline è quello di propagare creste (e depressioni) d'onda in maniera controllata in modo da mantenere l'integrità di ciascuna onda.

In ciascuna interfaccia tra stadi di pipeline di C-element, un osservatore vedrà un corretto handshaking, la cui tempistica potrebbe essere differente da quella della parte a sinistra (*Left*) del circuito. Una volta che l'onda è iniettata nella Muller pipeline, si propaga con una velocità che dipende dagli effettivi ritardi delle componenti circuitali. Alla fine, la prima onda iniettata (un *request*) raggiunge la parte destra (*Right*) del circuito. Se questa non risponde all'handshake, la pipeline comincia a riempirsi fino ad essere congestionata e, quando ciò si verifica, smette di operare l'handshake con la parte sinistra (*Left*). La Muller pipeline si comporta come un'onda attraverso strutture FIFO.

In aggiunta a questo comportamento elegante, la pipeline possiede un certo numero di simmetrie. Innanzitutto non ha importanza se viene impiegato un handshake *2-phase* o *4-phase*, il circuito è lo stesso e la differenza sta in come vengono interpretati i segnali e in come viene utilizzato il circuito. In secondo luogo, il circuito opera indifferentemente sia da sinistra verso destra che da destra verso sinistra. E' sufficiente invertire la definizione delle polarità dei segnali e invertire il ruolo di *request* e *acknowledge* per far funzionare il circuito da destra a sinistra. Ciò è analogo al concetto di elettroni e lacune in un semi-conduttore: quando la corrente scorre in una certa direzione possiamo interpretarla come un flusso di lacune in quella direzione o come un flusso di elettroni in direzione opposta.

Per concludere, una proprietà interessante: il circuito opera correttamente con ogni tipo di delay nelle porte e linee; la Muller pipeline è *delay-insensitive*.

2.3. Stili di implementazione dei circuiti

Come abbiamo accennato in precedenza, la scelta del protocollo handshake condiziona l'implementazione del circuito (in termini di area, velocità, consumo, robustezza, etc.). La maggioranza dei circuiti nella pratica adotta uno dei seguenti protocolli:

- **4-phase bundled-data**: è il più vicino come design ai circuiti sincroni e solitamente conduce a un circuito con una elevata efficienza per via dell'uso esteso di assunzioni legate alla temporizzazione.
- **2-phase bundled-data**: introdotto con il nome di *Micropipelines* da Ivan Sutherland nella lezione tenuta in occasione del Turing Award del 1988.
- **4-phase dual-rail**: l'approccio classico. Affonda le proprie radici nel lavoro pionieristico di David Muller (anni '50).

Un fattore comune a tutti i protocolli è che tutte le implementazioni utilizzano una qualche variazione della Muller pipeline per controllare gli elementi di *storage*. Vediamo in breve le principali caratteristiche e differenze di semplici versioni per ciascuna di queste implementazioni.

2.3.1. 4-phase bundled-data

Una 4-phase bundled-data pipeline è particolarmente semplice. Una Muller pipeline è utilizzata per generare impulsi dei clock locali. Un impulso generato in uno stadio si sovrappone con quelli generati negli stadi adiacenti in modo attentamente controllato.

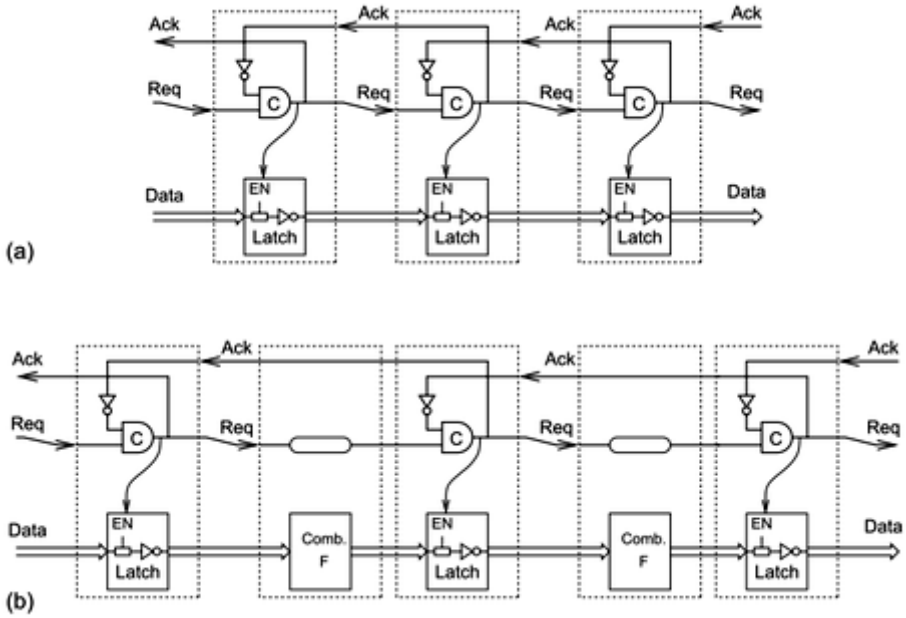


Figura 2.8 – Semplice 4-phase bundled-data pipeline.

La figura 2.8(a) ci mostra una FIFO, cioè una pipeline che non opera sui dati (cioè non fa *data-processing*) mentre la figura 2.8(b) ci mostra come un circuito combinatoriale (in altri termini un blocco funzionale) può essere inserito tra due latch. Per mantenere una corretta operatività, i ritardi vanno regolati e inseriti nei percorsi del segnale di request.

Questo circuito può essere visto come un tradizionale datapath sincrono, composto da latch e circuiti combinatoriali temporizzati da un *gated-clock* pilota distribuito, oppure può essere visto come una struttura di dataflow asincrona composta da due tipi di componenti handshake: latch e blocchi funzionali, come indicato nei rettangoli tratteggiati.

La pipeline mostrata in figura 2.8 è particolarmente semplice e presenta quindi alcuni svantaggi: quando si "riempie", lo stato dei C-element è (0, 1, 0, 1, etc.) e di conseguenza solo un latch su due sta memorizzando dati. Questo non è certo peggiore che avere un circuito sincrono con flip-flop in configurazione master-slave, ma è possibile progettare pipeline asincrone (e FIFO) che sono di gran lunga più efficienti.

Un altro svantaggio è la velocità. Il *throughput*[3] di una pipeline o di una FIFO dipende dal tempo che impiega a completare un ciclo di handshake e per l'implementazione appena illustrata ciò implica comunicare con entrambe le parti (*Left* e *Right*). Esistono implementazioni migliori sia in termini di velocità sia in termini di gestione dei dati.

2.3.2. 2-phase bundled-data

Anche la 2-phase bundled-data pipeline utilizza una Muller pipeline come circuito di controllo, ma i segnali (di controllo) sono interpretati come eventi o transizioni (figura 2.9). Per questa ragione, il circuito si avvale di speciali latch, chiamati *capture-pass*: eventi sugli input *C* e *P* si alternano, facendo in modo che il latch a sua volta alterni tra le modalità *C*apture e *P*ass. Questo tipo di esigenza ha prodotto un latch speciale il cui design è mostrato in figura 2.10.

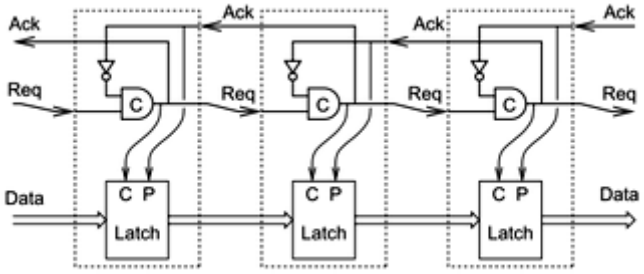


Figura 2.9 – Semplice 2-phase bundled-data pipeline.

Lo switch utilizzato in figura 2.10 è un multiplexer, e il latch controllato da eventi può essere pensato come una coppia di latch level-sensitive (operanti in modo alterno), seguita da un multiplexer e da un buffer. La figura 2.9 mostra una pipeline che non opera sui dati. Anche in questo caso valgono le stesse considerazioni fatte per il 4-phase bundled-data quando inseriamo circuiti combinatoriali tra due latch. L'approccio 2-phase bundled-data fu sviluppato da Ivan Sutherland verso la fine degli anni '80. Il nome *micropipelines* è solitamente utilizzato in modo intercambiabile con *protocollo 2-phase bundled-data*, ma il primo si riferisce anche all'uso di un particolare insieme di componenti che sono basati su eventi. In aggiunta al latch di figura 2.10 questi sono: *AND*, *OR*, *Select*, *Toggle*, *Call* e *Arbiter* (questi ultimi menzionati solo per completezza).

A livello concettuale l'approccio 2-phase bundled-data è elegante ed efficiente. Paragonato al 4-phase bundled-data, evita gli sprechi di tempo e potenza che derivano nella parte *return-to-zero* dell'handshaking. Comunque, come illustrato dal design del latch, l'implementazione dei componenti che rispondono alle transizioni del segnale è spesso più complessa rispetto a quella dei componenti che rispondono ai livelli di segnale. In aggiunta, la logica di controllo tende anch'essa a un livello di complessità maggiore. Detto questo, si può concludere che il 2-phase bundled-data sia preferibile in sistemi con dataflow di tipo *unconditional*[4] e che necessitano di

[3] Il throughput di un canale di comunicazione è la capacità di trasmissione effettivamente utilizzata.

[4] Un dataflow è di tipo *unconditional* quando non contiene blocchi decisionali (*conditional*) dove viene scelto il percorso in base alla verifica di date condizioni.

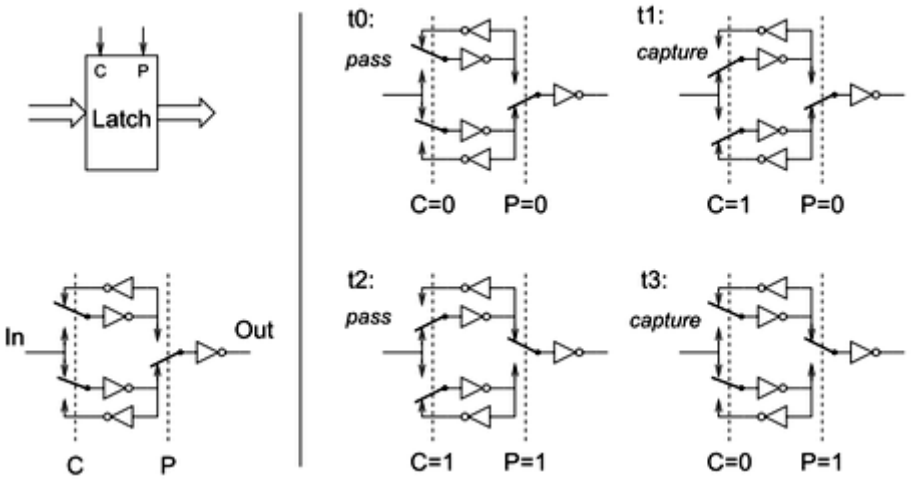


Figura 2.10 – Implementazione e funzionamento di un latch Capture-Pass gestito da eventi. Al tempo t_0 il latch è in modalità Pass e i segnali C e P sono entrambi a 0. Un evento sull'input C fa sì che il latch entri in modalità Capture, e così via.

lavorare a velocità elevate ma, come già menzionato, tutto questo ha il suo prezzo: maggiore velocità significa maggiore area di silicio e maggior consumo di potenza. In questo frangente, il design asincrono non è affatto diverso da quello sincrono.

2.3.3. 4-phase dual-rail

Come per la versione 2-phase, anche la 4-phase dual-rail pipeline è basata sulla Muller pipeline, ma in modo più elaborato che deriva dalla codifica combinata dei segnali di dati e request. La figura 2.11 illustra l'implementazione di una pipeline che non opera sui dati, con estensione di 3 stadi e ampiezza di 1 bit. Può essere considerata come due Muller pipeline connesse in parallelo che utilizzano un segnale di acknowledge comune per ogni stadio per sincronizzare le operazioni. La coppia di C-element in uno stato della pipeline può memorizzare la parola di codice "vuota" $\{d.t, d.f\} = \{0, 0\}$, facendo in modo che il segnale di acknowledge in uscita da quello stadio sia 0; oppure può memorizzare le due parole di codice valide $\{1, 0\}$ e $\{0, 1\}$, facendo sì che il segnale di acknowledge in uscita sia 1. A questo punto, ricordandoci di quanto detto nella sezione 2.1.2, il lettore può notare che siccome la parola di codice $\{1, 1\}$ non è valida (e quindi non si verifica), il segnale di acknowledge generato dalla porta *OR*, indica con certezza lo stato di questo stadio della pipeline (o "valido" o "vuoto").

Ponendo in parallelo N pipeline di questo tipo otteniamo una pipeline a N-bit. Ciò non garantisce che tutti i bit di una parola arrivino contemporaneamente, ma spesso la sincronizzazione necessaria viene fatta nei blocchi funzionali. Se è necessaria la sincronizzazione di bit paralleli quindi, i segnali individuali di acknowledge possono esse combinati in un segnale di acknowledge globale usando un C-element.

La figura 2.12 mostra un latch a N-bit. Le porte *OR* e il C-element nel rettangolo tratteggiato formano una struttura chiamata *completion detector*, che indica se la parola di codice a N-bit codificata dual-rail contenuta nel latch è di tipo "vuoto" o "valido". La figura illustra anche un'implementazione di un *completion detector* che

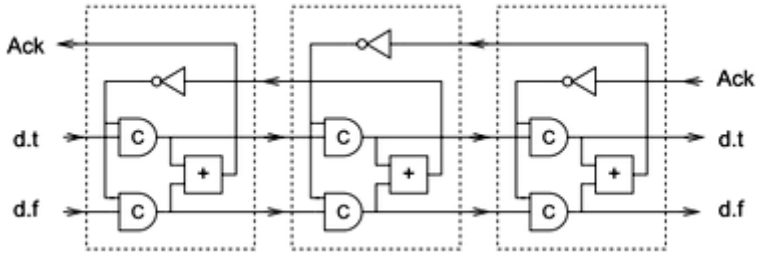


Figura 2.11 – Semplice 3-stage 1-bit wide 4-phase dual-rail pipeline.

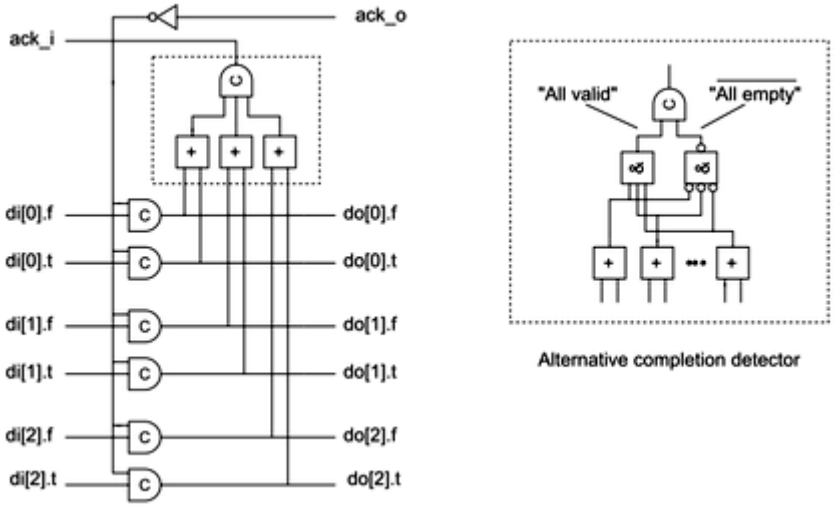
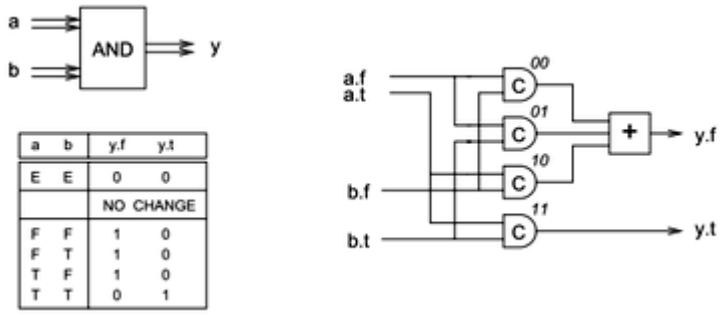


Figura 2.12 – Latch a N-bit dotato di *completion detection*.

utilizza un C-element con due soli input.

Vediamo di analizzare ora come sono implementati i circuiti combinatoriali per circuiti 4-phase dual-rail. Sappiamo che i circuiti combinatoriali devono essere trasparenti agli handshake che avvengono tra i latch. Quindi, tutti gli output di un circuito combinatoriale non possono diventare validi finché tutti gli input non sono validi. In caso contrario, il latch al ricevitore potrebbe settare prematuramente sul livello 1 il segnale di acknowledge (prima che tutti i segnali provenienti dal latch trasmettitore siano diventati validi). Per le stesse ragioni, tutti gli output di un circuito combinatoriale non possono andare a 0 (stato "vuoto") finché tutti gli input non sono a 0. In questo caso il ricevitore potrebbe settare prematuramente il segnale di acknowledge sul livello 0 (prima che tutti i segnali in uscita dal trasmettitore siano andati a 0). Di conseguenza, un circuito combinatoriale 4-phase dual-rail necessita di elementi capaci di ritenere lo stato ed esibisce un comportamento simile a un'isteresi nelle transizioni $''vuoto'' \longrightarrow ''valido''$ e $''valido'' \longrightarrow ''vuoto''$.

Un approccio particolarmente semplice, che utilizza solo C-element e porte *OR*, è illustrato in figura 2.13, che mostra una possibile implementazione di una porta *AND* di tipo dual-rail. Il circuito attende finché tutti gli input diventano validi e quando questo accade, esattamente uno dei C-element si porta a 1. Questo fa sì che la linea



a	b	y.f	y.t
E	E	0	0
		NO CHANGE	
F	F	1	0
F	T	1	0
T	F	1	0
T	T	0	1

Figura 2.13 – Una porta *AND* 4-phase dual-rail: simbolo, tabella di verità e implementazione.

di uscita si porti a 1 a sua volta, e la porta produce il risultato valido desiderato. Quando tutti gli ingressi sono a 0, anche tutti i C-element vanno a 0, e l'output della porta *AND* diventa 0 (o "vuoto"). Si noti che sia l'operazione *AND*, sia il ciclo di isteresi ($''vuoto'' \longrightarrow ''valido''$ e $''valido'' \longrightarrow ''vuoto''$), sono diretta conseguenza dei C-element che garantiscono così la trasparenza per l'handshake. Si noti inoltre che la porta *OR* non è mai affetta da più di un input settato sul livello 1.

Altre porte di tipo dual-rail come ad esempio *OR* o *XOR* possono essere implementate in modo simile. Per realizzare un dual-rail inverter invece è sufficiente scambiare tra loro le linee 0,1.

Il numero dei transistor impiegati in questo tipo di porte dual-rail è piuttosto alto e nella pratica si adottano soluzioni più efficienti (grazie a manipolazioni circuitali basate su considerazioni matematiche). In questa breve introduzione l'interesse è più focalizzato sui concetti fondamentali di base che non sulle ottimizzazioni.

A questo punto, disponendo di porte *AND*, *OR*, etc. è possibile realizzare circuiti combinatoriali per espressioni booleane arbitrarie utilizzando le normali tecniche di sintesi di circuiti. La trasparenza all'handshake, che è una proprietà fondamentale dei singoli blocchi di base, si mantiene anche nel più esteso dei circuiti combinatoriali.

2.4. Concetti fondamentali e classificazione

I circuiti asincroni possono essere classificati come *self-timed*, *speed-independent* o *delay-insensitive* (risp. auto-temporizzati, indipendenti dalla velocità, indipendenti dai delay), in base alle tipologie di assunzioni sui delay che vengono fatte. In questa sezione vogliamo introdurre importanti aspetti teorici che sono correlati alla classificazione. L'obiettivo è quello di fornire, oltre ai concetti di base, anche i presupposti, per coloro che desiderano approfondire, per potersi orientare nella letteratura.

2.4.1. Indipendenza dalla velocità

In questa sezione vedremo le basi di un modello di David Muller per un circuito e le condizioni in base alle quali possiamo definire quest'ultimo come *speed-independent*. Un circuito è modellato assieme al suo (fittizio) ambiente, come una rete chiusa di porte. "Chiusa", in questo caso, significa che tutti gli input sono collegati a degli out-

put e vice versa. Le porte sono modellate come operatori booleani con delay arbitrari diversi da zero (*non-zero* delay) e si assume che le linee siano ideali. In questo contesto, il circuito può essere descritto come una serie di funzioni booleane concorrenti, una per ciascuna serie di output uscenti da una porta. Lo stato del circuito è l'insieme di tutti i vari output di ogni porta.

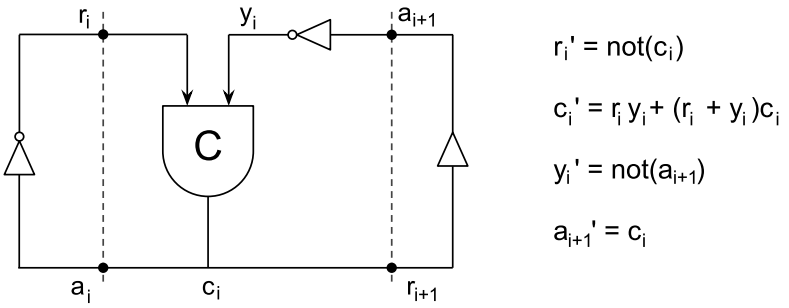


Figura 2.14 – Modello di Muller di una Muller pipeline con delle porte che simulano il comportamento dell'ambiente circostante.

In figura 2.14 questo concetto è illustrato per uno stadio della Muller pipeline con un inverter e un buffer che simulano un handshake come se fosse fatto realmente dalle parti sinistra e destra di un ambiente circostante.

Una porta il cui output è coerente con i suoi input è detta *stabile*, il suo "prossimo output" è identico al suo output attuale, $z_i' = z_i$. Una porta i cui input sono cambiati in modo da provocare una variazione nell'output è detta *eccitata*, il suo "prossimo output" è differente dal suo output attuale, quindi $z_i' \neq z_i$.

Dopo un arbitrario lasso di tempo, una porta eccitata può spontaneamente cambiare il proprio output (rendendolo conforme agli input) e stabilizzarsi. Si dice che una porta eccitata "scatta" (*fires*) e dopo essere "scattata" si stabilizza, dopodiché altre porte si ecciteranno e a loro volta "scatteranno", e così via. Per illustrare questo meccanismo, immaginiamo che il circuito in figura 2.14 sia nello stato $(r_i, y_i, c_i, a_{i+1}) = (0, 1, 0, 0)$. In questo stato (l'inverter) r_i è eccitato e ciò corrisponde alla parte sinistra dell'ambiente (che ricordiamo, è simulato) che sta per settare il segnale di request sul livello 1. Dopo che r_i è passato a 1, il circuito raggiunge lo stato $(r_i, y_i, c_i, a_{i+1}) = (1, 1, 0, 0)$ e c_i diviene eccitato. E' possibile, proseguendo nell'analisi, costruire il grafo completo con tutti con gli stati possibili, ma qui ci limiteremo a esporre solamente alcune idee fondamentali.

Nel caso generale, è possibile che numerose porte siano eccitate allo stesso tempo (cioè in un certo stato del circuito). Se una di queste porte, diciamo z_i, scattasse, la cosa interessante è ciò che accade alle altre porte che in quel momento sono eccitate e hanno z_i tra gli ingressi. Le opzioni sono due: (1) possono rimanere eccitate o (2) possono ritrovarsi in una condizione per cui non è più necessario un cambio nel loro output.

Un circuito si definisce speed-independent se il caso (2) appena menzionato non si verifica mai. Una porta eccitata che diventi stabile senza scattare è una potenziale causa di hazard. Dato che i delay sono sconosciuti, la porta potrebbe aver cambiato il proprio output, ma potrebbe anche non averlo fatto o potrebbe essere in procinto di farlo mentre il circuito magari si aspetta che l'output non vari.

Visto che il modello implica una variabile di stato (booleana) per ciascuna porta (e per ciascuna linea nel caso dei circuiti delay-insensitive) lo spazio degli stati diventa

molto ampio anche per circuiti molto semplici. Il problema si affronta con tecniche di teoria dei grafi legate alle transizioni dei segnali.

Ora che abbiamo un modello per descrivere e analizzare il comportamento dei circuiti (al livello delle porte), occupiamoci della loro classificazione vera e propria.

2.4.2. Classificazione dei circuiti asincroni

A livello di porte logiche, i circuiti asincroni possono essere classificati come self-timed, speed-independent o delay-insensitive in base alle assunzioni fatte sui delay. La figura 2.15 serve a illustrare la seguente discussione. In figura sono riportate tre porte: A, B e C, dove il segnale di output di A è connesso agli input di B e C.

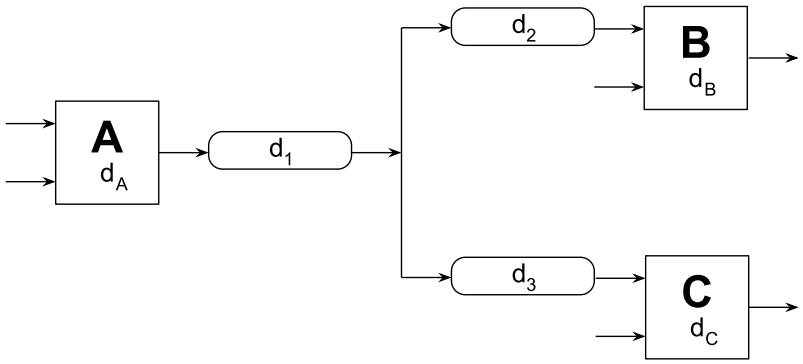


Figura 2.15 – Un frammento di circuito con delay di porta e linea. L'output della porta A si divide negli input per le porte B e C.

Un circuito *speed-independent* (SI) è un circuito che opera "correttamente"[(5)] assumendo delay positivi, limitati ma non conosciuti, per le porte e linee ideali, cioè a delay 0. Riferendoci alla figura 2.15 ciò significa d_A, d_B, d_C arbitrari e $d_1 = d_2 = d_3 = 0$. Assumere delay nulli per le linee non è propriamente realistico per gli odierni processi nei semi-conduttori. Permettendo invece d_1 e d_2 arbitrari e imponendo $d_2 = d_3$, i delay su linea possono essere condensati nelle porte e da un punto di vista teorico il circuito rimane di tipo *speed-independent*.

Un circuito che opera "correttamente" assumendo delay positivi, limitati ma non conosciuti, sia per le porte che per le linee, viene chiamato *delay-insensitive* (DI). Riferendoci sempre alla figura 2.15 questo significa avere $d_A, d_B, d_C, d_1, d_2, d_3$ arbitrari. Circuiti di questo tipo sono estremamente robusti.

Un modo per mostrare che un circuito è di tipo delay-insensitive consiste nell'utilizzare un modello di Muller del circuito dove le linee (dopo le biforcazioni) siano modellate come dei buffer. Se questo modello equivalente è delay-insensitive, allora anche il circuito originale è delay-insensitive.

Sfortunatamente, la classe di circuiti delay-insensitive è piuttosto piccola. Solo circuiti composti da Muller C-element e inverter possono essere delay-insensitive e in questo senso la Muller pipeline nelle figure 2.7 e 2.14 è un esempio importante.

Circuiti che sono delay-insensitive con l'eccezione di qualche (accuratamente controllata) biforcazione di linea dove $d_2 = d_3$, sono detti *quasi-delay-insensitive* (QDI).

[(5)] Correttamente è tra virgolette perché è inteso in linea teorica. Possono sempre sorgere errori per svariati motivi e in quel caso, nonostante il circuito si sia comportato "correttamente", i dati in output possono essere errati. Della gestione degli errori ce ne occuperemo più avanti.

Tali biforcazioni di linea (*wire forks*), dove le transizioni dei segnali si verificano allo stesso tempo a ogni estremità, sono dette isocroniche (*isochronic*) e tipicamente si trovano nelle implementazioni (a livello delle porte logiche) dei blocchi base fondamentali del circuito, dove il progettista può controllare efficacemente i delay su linea. Ad alti livelli di astrazione, la composizione dei blocchi solitamente è di tipo delay-insensitive.

Poiché la classe dei circuiti delay-insensitive è così ristretta (virtualmente esclude qualsiasi circuito che esegua delle computazioni), nella pratica quasi tutti i circuiti che in letteratura vengono definiti delay-insensitive sono in realtà quasi-delay-insensitive.

Per concludere questo paragrafo, alcuni cenni ai circuiti self-timed: l'indipendenza dalla velocità e la non sensibilità ai delay come introdotte sopra sono proprietà ben definite (matematicamente) all'interno del modello (non limitato) a porte e linee. Circuiti la correttezza delle cui operazioni dipende da assunzioni più elaborate o ingegneristiche sono semplicemente detti self-timed.

2.4.3. Isochronic forks

In base a quanto detto sopra è chiaro che la distinzione tra circuiti di tipo speed-independent e delay-insensitive dipende dalle biforcazioni delle linee e, più nello specifico, dal fatto che i delay ai terminali delle linee (dopo le biforcazioni) siano gli stessi o meno. Se i delay sono gli stessi, la biforcazione è detta isocronica.

La necessità di realizzare biforcazioni isocroniche è collegata al concetto di indicazione introdotto nella sezione 2.2.1 a pagina 15. Considerando una situazione come quella in figura 2.15, dove A abbia mutato il proprio output. Ciò produce un cambiamento negli input di B e C e, dopo un certo lasso di tempo, le porte B e C possono reagire a questo cambiamento modificando a loro volta il proprio output. Se ciò accade, diciamo che il cambiamento dell'output di A è *indicato* dal cambiamento dell'output di B e C. Se invece fosse solo B a cambiare il proprio output, non potremmo concludere che la porta C abbia anch'essa visto il cambiamento nel proprio input (e quindi nell'output di A). In questo caso è necessario rafforzare le assunzioni ponendo $d_2 = d_3$ (cioè stabilendo un isochronic fork) grazie alle quali possiamo concludere che dato che la variazione dell'output di A è stata indicata dalla conseguente variazione dell'output di B, allora anche C deve aver visto il cambiamento nel proprio input.

2.4.4. Relazione con i circuiti

Negli approcci 2-phase e 4-phase bundled-data i circuiti di controllo sono solitamente di tipo speed-independent (o in alcuni casi delay-insensitive), ma i circuiti di datapath con i loro delay accoppiati sono di tipo self-timed. Circuiti progettati seguendo il paradigma 4-phase dual-rail generalmente sono di tipo quasi-delay-insensitive.

Nei circuiti mostrati nelle figure 2.11 e 2.13 le biforcazioni che si connettono agli input di molteplici C-element devono essere isocroniche, mentre quelle che si connettono agli input di molteplici porte OR sono delay-insensitive.

Le differenti classi di circuiti, DI, QDI, SI e self-timed sono metodologie che non si escludono mutuamente, piuttosto sono utili astrazioni che possono essere sfruttate a livelli di progetto differenti. In quasi tutte le applicazioni pratiche, troviamo un mix. Ad esempio, nei processori Amulet (si veda cap. 11 a pagina 133), il design SI è

utilizzato per i controller asincroni locali, bundled-data per il data-processing locale e DI per la composizione ad alto livello.

Un altro esempio che possiamo portare è un progetto riguardante un banco di filtri da inserire negli amplificatori auricolari (*hearing-aid*) utilizzati da coloro che hanno problemi all'udito. Utilizza il protocollo DI 4-phase dual-rail all'interno dei moduli RAM e dei circuiti aritmetici per garantire una robusta *indicazione*, e l'approccio 4-phase bundled-data con controllo SI al livello più alto di design, risultando quindi diverso dal design dell'Amulet.

Questi esempi servono per enfatizzare che la scelta dei vari approcci/protocolli è dettata in base a considerazioni che variano da progetto a progetto, a seconda delle diverse caratteristiche che il circuito deve assumere per essere ottimizzato sotto determinati aspetti.

E' anche importante sottolineare ancora che l'indipendenza dalla velocità e la non sensibilità ai delay sono proprietà matematiche che possono essere verificate per una data implementazione. Se un componente astratto (come ad es. un C-element o una porta complessa tipo And-Or-Invert) è sostituito da una sua implementazione che impiega semplici porte e possibilmente qualche biforcazione di linea, allora il circuito potrebbe non mantenere le proprie caratteristiche originali (cioè indipendenza dalla velocità o non sensibilità ai delay).

Come esempio possiamo riferirci alla Muller pipeline in figura 2.7 e 2.14. Se rimpiazziamo il C-element con una implementazione che usa solo porte *AND* e *OR*, la pipeline non è più di tipo delay-insensitive. Inoltre, anche semplici porte sono in realtà delle astrazioni: nei CMOS, le primitive sono transistor n e p e anche la più semplice delle porte include almeno una biforcazione.

3. GALS - Globally Asynchronous Locally Synchronous

Come abbiamo accennato nei capitoli precedenti, il design e l'implementazione di circuiti asincroni si trovano a dover fare i conti con una realtà che nella pratica non contempla né l'uno né l'altra. Gli strumenti tecnologici di oggi sono pensati per funzionare con processori e circuiti sincroni, e quindi sono predisposti per interfacciarsi a essi seguendo logiche basate sulla temporizzazione mediante un clock. Completano il quadro la mancanza di esperienza e di strumenti per il design. D'altro canto, i problemi e le difficoltà tecniche che la circuiteria sincrona ha conosciuto negli ultimi anni impongono la ricerca di una soluzione con un paradigma diverso, basato su diverse assunzioni, che permetta di trovare rimedi altrimenti impossibili.

Uno degli approcci più usati, in particolar modo negli ultimi anni, è chiamato *GALS*, cioè Globally Asynchronous Locally Synchronous. Questa soluzione ibrida consente di ovviare ad alcune problematiche legate all'approccio puramente sincrono, sfruttando i benefici che derivano dall'eliminazione del segnale di clock globale del circuito. In questo capitolo introdurremo i principali concetti relativi a questo approccio, vedremo le principali classificazioni e presenteremo al lettore alcuni esempi che possano aiutarlo a comprendere questo interessante modello.

3.1. Introduzione

La popolarità del design Globally Asynchronous Locally Synchronous (GALS d'ora in avanti) sta aumentando considerevolmente nell'ambiente industriale ai giorni nostri. La ragioni sono molteplici: (1) i costanti miglioramenti nella tecnologia utilizzata per costruire i semiconduttori hanno portato a una riduzione delle dimensioni (si pensi al fatto che ormai è considerato normale l'uso della tecnologia a $45nm$ per i microprocessori *high-end*) e al conseguente aumento del numero di dispositivi sul singolo chip. Ciò rende sempre più difficile il design di una rete di clock globale che possa controllare efficacemente tutti i blocchi del design e, inoltre, reti di questo tipo incrementano significativamente il consumo globale di potenza. (2) Le tempistiche del mercato, sempre più ristrette, portano all'aumento del riutilizzo dei blocchi IP[(1)] in cui ciascun blocco è progettato e ottimizzato per differenti velocità di clock da diversi gruppi di progettisti. (3) Molti progetti richiedono domini di clock multipli regolati su frequenze diverse per via della natura delle computazioni e delle comunicazioni che operano.

Il termine GALS fu utilizzato per la prima volta nella tesi di dottorato di Chapiro: "Globally-Asynchronous Locally-Synchronous Systems" [20], nella quale egli fornì una soluzione utilizzando una circuiteria che sfruttava i cosiddetti *pausible-clock*. Un clock di questo tipo è un normale clock che viene inserito in un circuito capace di arrestarlo o metterlo in pausa.

[(1)]IP: Intellectual Property.

Dopo la pubblicazione della tesi di Chapiro, il termine GALS ha acquistato popolarità sia in ambiente accademico che in quello industriale. Spezzare le assunzioni di sincronia nel design digitale è sempre "sconvolgente" per i progettisti e, per alleviare le difficoltà, i ricercatori dell'EDA hanno proposto varie soluzioni basate su GALS.

Gli strumenti, le tecniche di verifica e le metodologie di testing per il design asincrono non hanno una diffusione così vasta come invece la loro controparte sincrona e quindi al giorno d'oggi, dopo più di vent'anni dal lavoro di Chapiro, il loro utilizzo è purtroppo ancora limitato.

Alcune domande chiave che un designer di circuiti sincroni potrebbe porsi sul design, verifica e test di un'architettura GALS, sono:

- Il design digitale sincrono è stato studiato e compreso in modo molto approfondito: le metodologie e le difficoltà tecniche sono ben note e stabilite e i venditori di strumenti offrono una grande quantità di ambienti per il design che vengono sfruttati per la progettazione, verifica, sintesi e per il testing. Anche se esistono importanti problemi legati a dimensioni, complessità, consumo di potenza, variazioni dei processi di creazione e costo per eseguire validazioni, utilizzare l'astrazione sincrona e misurare i delay di una computazione per un ciclo di clock semplifica le metodologie di design. Oltretutto queste operazioni sono ormai profondamente incorporate nella attuali pratiche di design. C'è quindi un modo per risolvere i problemi evitando il design GALS e rimanendo completamente nel mondo sincrono?

- Sarebbe possibile rimanere nel mondo sincrono modificando le interconnessioni e i blocchi computazionali usando protocolli di tipo *latency-insensitive*. Questo design *elastico* rimane sincrono ma permette flessibili variazioni delle latenze e può essere visto come una implementazione discretizzata di un protocollo asincrono. Ci sono teorie generali per progettare sistemi *latency-insensitive* e qual è il modo migliore per implementarli? Questi sistemi allevierebbero in qualche modo il bisogno di un approccio GALS? E ancora, sarebbe possibile utilizzarli in un contesto di domini di clock multipli?

- Ci sono molti modi per progettare sistemi basati su GALS usando diverse tipologie di temporizzazione (ad esempio, clock locali completamente asincroni; sorgenti sincrone con trasferimenti del clock dal trasmettitore al ricevitore; *pausible-clocks*; multi-clock sincroni e armonici). Alcune di queste metodologie rendono i domini di clock completamente indipendenti al costo di problemi di sincronizzazione tra i domini stessi. Altre richiedono che vengano soddisfatti alcuni vincoli temporali nella generazione del clock, ma eliminano i problemi riguardo alla sincronizzazione. Quali metodologie sono preferibili e per quali classi di circuiti?

- Il design di circuiti asincroni esiste da lungo tempo. Comunque, nonostante alcuni vantaggi (ad esempio, performance legate al caso medio e ridotte emissioni elettromagnetiche), questo approccio non ha preso piede nell'industria su larga scala. Gli strumenti e le metodologie per il design di circuiti asincroni non sono ampiamente disponibili. L'architettura GALS si troverà ad affrontare questi stessi problemi?

- Avendo gli strumenti per fare test e verifiche nel design di circuiti sincroni, è possibile progettare un circuito sincrono e poi suddividerlo in "isole" sincrone, in modo da rendere l'implementazione di un GALS un semplice lavoro di rifinitura del design originale, mantenendone al tempo stesso la correttezza?

- Che tipo di problematiche legate a clock e segnali bisogna risolvere per implementare correttamente protocolli GALS di alto livello? *Fail* nella sincronizzazione potrebbero essere disastrosi in questo tipo di design. E' possibile che l'assenza di tali problemi sia garantita per le eventuali implementazioni in silicio?

- Dato che i protocolli tra blocchi IP sarebbero di tipo self-timed, latency-tolerant e non necessariamente pilotati da clock, la semplicità del controllo centrale può andare persa in un sistema GALS e come conseguenza si sperimenterebbe l'insorgere di problemi di deadlock e altre problematiche che affliggono i sistemi distribuiti. Come possiamo garantire l'assenza di tali problemi?

I ricercatori hanno sperimentato varie strutture, come le *network on chip* (NoCs), le infrastrutture di comunicazione su chip e altre ancora. Nuove problematiche sono sorte in contesti di comunicazioni SoC (*System on Chip*) legate ad esempio a blocchi IP eterogenei e indipendenti. La speranza è che i protocolli GALS vengano sperimentati su queste tipologie di NoCs e che i problemi che sorgeranno spingano la ricerca e lo studio di questo approccio.

I metodi formali devono giocare un ruolo primario nel design di GALS (l'intera storia del design asincrono lo dimostra). Il successo sia dell'analisi che della sintesi è fortemente basato sui modelli formali di concorrenza e lo sviluppo di metodi per la validazione e manipolazione di tali modelli. Fortunatamente, i metodi formali sono maturati lungo il corso degli anni e possono essere applicati con successo a sistemi distribuiti con molteplici "isole" sincrone.

Quando si progetta software real-time per una particolare piattaforma embedded, gli aspetti riguardanti la temporizzazione possono essere astratti grazie alle assunzioni del modello sincrono e la funzionalità del componente software può essere modellata con i cosiddetti *linguaggi sincroni*[(2)], come *Esterel*, *Signal* e *Lustre*, per citarne alcuni. Le prove della correttezza dei modelli funzionali di tali software possono essere stabilite sotto l'ipotesi di sincronia. L'analisi della computazione, la generazione e così via, possono essere ricavate dal calcolo formale. Il codice real-time (embedded) può quindi essere generato in modo da essere corretto per costruzione.

Tuttavia, se il software si trova a girare su nodi distribuiti, l'ipotesi di sincronia viene meno perché la comunicazione di eventi che avvengono tra varie computazioni non è più così rapida da giustificare tali assunzioni. Con questo problema in mente, i ricercatori che operano nel campo della programmazione sincrona hanno proposto un design GALS per risolvere il problema della generazione di codice distribuito, dove l'assunzione di sincronia è giustificata per quelle parti del codice che viene eseguito

[(2)]Un *linguaggio di programmazione sincrono* è un linguaggio ottimizzato per la programmazione di sistemi reattivi, sistemi che sono spesso interrotti e che devono *rispondere* in tempi brevissimi. Molti di questi sistemi talvolta sono chiamati *realtime systems* e molto spesso vengono impiegati nei sistemi embedded. [http://en.wikipedia.org/wiki/Synchronous_programming_language]

nello stesso nodo di una architettura distribuita, ma sono necessari protocolli asincroni per fare da ponte tra queste diverse parti di codice garantendone il corretto funzionamento.

C'è stato un leggero progresso nei metodi formali per il software embedded per il design GALS ma comunque, questo tipo di problematica si concentra solamente sul design GALS nel dominio hardware. Sfortunatamente mancano ancora degli approcci formali sufficientemente robusti e molte soluzioni spesso sono *ad hoc*.

Con questa introduzione speriamo di essere riusciti a comunicare al lettore quante e quali siano ancora le problematiche che vanno affrontate e risolte prima di poter pensare alla produzione di massa per questa tecnologia.

Ora ci concentreremo sulla tassonomia delle architetture GALS, fornendo degli esempi in modo che il lettore sia facilitato nel comprendere le differenze tra i vari modelli.

3.2. Tassonomia dei circuiti GALS

I sistemi digitali a singolo clock appartengono ormai al passato. Nonostante la maggior parte dei circuiti digitali rimanga sincrona, molti design incorporano domini di clock multipli, spesso operanti su diverse frequenze. Utilizzare interconnessioni asincrone disaccoppia i parametri temporali relativi a ciascun blocco. I sistemi che utilizzano questa tecnica si chiamano GALS. La figura 3.1 illustra un esempio.

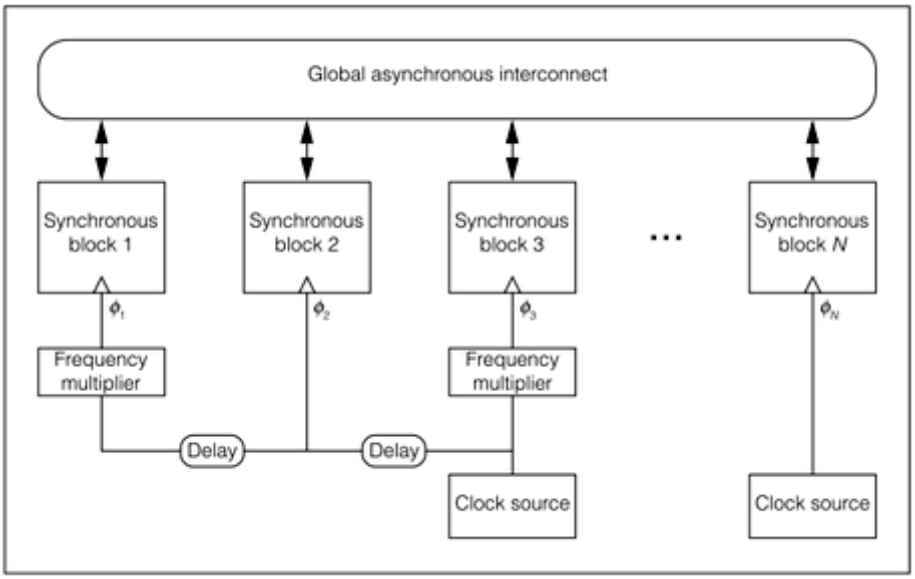


Figura 3.1 – Diagramma di alto livello di un sistema GALS.

Il design GALS offre una maggiore facilità di riutilizzo dei blocchi funzionali, una *timing closure*[3] semplificata e un ridotto consumo di potenza dovuto all'eterogeneità dei clock. Per minimizzare i tempi per l'immissione sul mercato, design SoC di

[3]Timing closure è quel processo in base al quale un design FPGA o VLSI viene modificato per aderire ai vincoli temporali richiesti al circuito.

grandi dimensioni devono integrare molti blocchi funzionali con il minimo sforzo di progetto. Questi blocchi sono progettati solitamente con metodi sincroni standard e spesso hanno diverse specifiche sui tempi di clock. Un approccio GALS può facilitare il rapido riutilizzo dei blocchi poiché inserisce dei circuiti *wrapper* che si prendono cura della comunicazione tra blocchi attraverso le frontiere fra i vari domini di clock. I SoCs inoltre possono anche ridurre il consumo regolando i clock di alcuni blocchi alla loro minima velocità.

Anche i processori ad alte prestazioni soffrono degli stessi problemi. A causa dell'incremento del numero dei transistor (per singolo chip) e delle frequenze, distribuire un clock globale a basso *skew*[4] diventa sempre più difficile. Alcuni studi effettuati su microprocessori basati su architetture GALS dimostrano come sia possibile, regolando con precisione i livelli di tensione di alimentazione e le velocità dei clock di differenti blocchi funzionali, guadagnare in termini di consumo ed eliminando di fatto il bisogno di un segnale globale di clock.

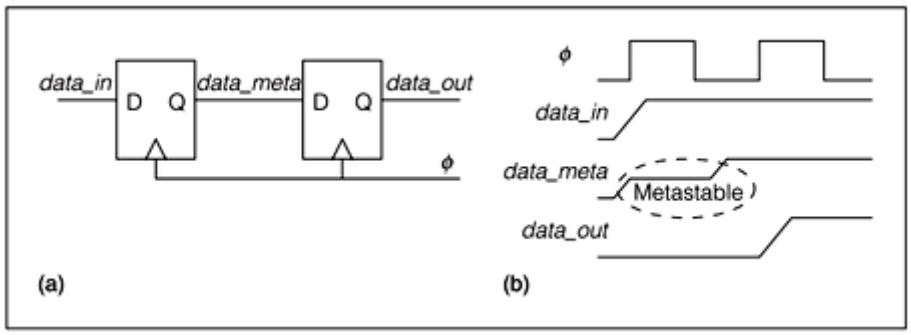


Figura 3.2 – Un two-flop sincronizzatore che mostra la metastabilità: (a) circuito e (b) diagramma temporale.

In base alle precedenti considerazioni possiamo quindi concludere che il problema centrale nei design GALS consiste nell'attraversamento dei diversi domini di clock. Se ad esempio dei dati destinati a un flip-flop (o a un latch) provengono da un diverso dominio temporale, si possono verificare delle violazioni nelle specifiche di *setup* and *hold*. Violazioni di questo tipo possono causare un output metastabile nel quale il livello di tensione può risultare indeterminato per un lasso temporale anche considerevole prima di stabilizzarsi su un livello valido.

Comunque, è possibile minimizzare la probabilità di *failure* dovuti alla metastabilità utilizzando circuiti sincronizzatori che possono anche essere semplici flip-flop connessi in serie. La figura 3.2 illustra a questo proposito un sincronizzatore abbastanza comune che utilizza due flip-flop (*sincronizzatore two-flop*).

La probabilità che si verifichi un *failure* è inversamente proporzionale al tempo dedicato alla stabilizzazione dei segnali, che è a sua volta proporzionale al numero dei flip-flop che costituiscono la catena. Oltre un certo numero di flip-flop, i sincronizzatori possono offrire tempi medi tra *failure* (MTBFs[5]) di milioni di anni o più, se propriamente progettati.

[4]Il clock skew è la misura del ritardo di fase che il segnale di clock presenta in diversi punti del circuito.
[5]MTBFs: Mean Times Between Failures.

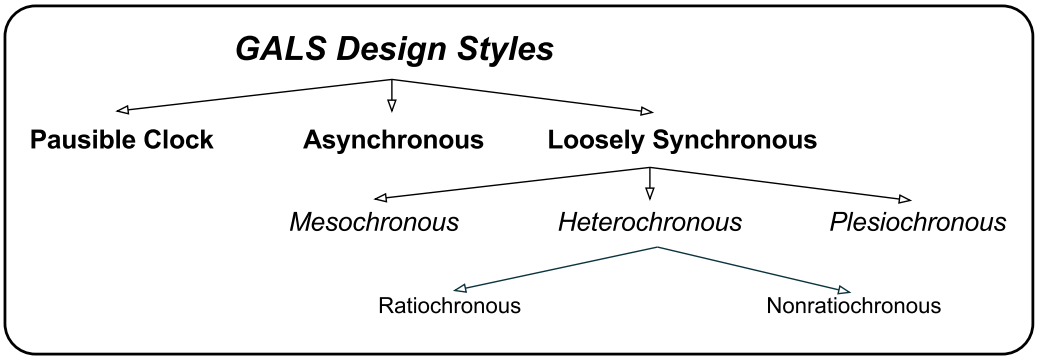


Figura 3.3 – Tassonomia degli stili di design GALS.

Nel resto di questa sezione, vedremo come classificare gli stili di design GALS in base al modo in cui essi trasferiscono i dati attraverso domini con clock regolati diversamente.

Le categorie principali sono tre (come illustrato in figura 3.3):

1. Pausible-clocks
2. Asynchronous
3. Loosely synchronous

Il design *pausible-clocks* si basa su clock generati localmente che possono essere dilatati (o compressi) o messi in pausa, sia per prevenire la metastabilità che per lasciare che un trasmettitore (o un ricevitore) stalli a causa di un canale pieno (o vuoto). Un oscillatore ad anello (ring oscillator) è solitamente il responsabile per la generazione dei clock. Gli Integrated Systems Laboratory alla ETHZ (Swiss Federal Institute of Technology di Zurigo) [32, 45] hanno implementato numerosi chip basati sul modello *pausible-clocks*, tra cui un chip dedicato alla crittografia. Circuiti speciali *wrapper* interfacciano i vari blocchi in modo che ciascun circuito *wrapper* includa un generatore di un *pausible-clock*.

Il design asincrono (*asynchronous*) si presta per il caso generale in cui non vi è alcuna assunzione temporale tra i vari clock. Design di questo tipo sono estremamente flessibili per quanto riguarda l'aspetto della temporizzazione. Per esempio, l'architettura Nexus della Fulcrum Microsystems [65] include uno switch crossbar asincrono che gestisce le comunicazioni tra blocchi che operano a frequenze arbitrarie.

Il design loosely synchronous è pensato per quei casi in cui c'è una relazione ben definita e affidabile tra i clock. E' possibile sfruttare la stabilità di questi clock per ottenere un'elevata efficienza e al tempo stesso mantenere una buona tolleranza per il gran numero di ritardi di fase che interessano le interconnessioni globali. Messerschmitt [79] ha proposto una tassonomia di relazioni temporali che si verificano comunemente:

- *Mesochronous*: il trasmettitore e il ricevitore operano esattamente alla stessa frequenza con una differenza di fase sconosciuta ma stabile. Il processore 80-core di Intel impiega un design di questo tipo. Utilizza una tessellatura sincrona

e un NoC di tipo skew-tolerant come schema di interconnessione pilotato da un clock globale stabile.

- *Plesiochronous*: il trasmettitore e il ricevitore operano alla stessa frequenza nominale. Le frequenze però possono esibire una minima differenza, anche di poche parti per milione, che causa la deriva della fase. Gigabit Ethernet ne è un esempio abbastanza comune.

- *Heterochronous*: Il trasmettitore e il ricevitore operano su diverse frequenze nominali.

Un interessante sottoinsieme delle relazioni di tipo *heterochronous* è il caso in cui il rapporto tra le frequenze dei clock sia proporzionale a un numero razionale, cioè ad esempio che la frequenza del ricevitore sia un multiplo razionale di quella del trasmettitore, e che entrambe siano derivate dallo stesso generatore in modo che la relazione di fase sia periodica e calcolabile. Ci si riferisce a questa configurazione con il termine *ratiochronous*, mentre in caso contrario si parla di *nonratiochronous*.

Per ciascuno di questi tre tipi di design esporremo ora un semplice esempio che sfrutta una comunicazione a senso unico tra blocchi trasmettitore e ricevitore. I blocchi operano in modo sincrono usando due clock differenti e sono connessi da un buffer FIFO che è robusto e privo di metastabilità. La capacità del buffer può essere arbitraria (entro certi limiti). Ad esempio può includere un singolo elemento, ma questo porta ripercussioni sul throughput. Per spedire un dato il trasmettitore esegue assert su *put* e pilota *data_in*. Il buffer accetta il dato sul fronte di salita del *put* e abbassa *ok_to_put* a 0. Se questa operazione riempie il buffer, *ok_to_put* rimane a 0 finché il dato non viene rimosso. Lato ricevitore, il buffer esegue assert su *ok_to_take* quando ci sono dei dati a disposizione. Per estrarre un dato, il ricevitore registra *data_out* ed esegue assert su *take*. il buffer abbassa *ok_to_take* finché non ci sono nuovi dati disponibili. Se il buffer è vuoto, *ok_to_take* rimane a 0 finché nuovi dati non vengono inseriti.

3.2.1. Pausible clocks

Come già accennato, il primo uso del termine GALS si deve a Chapiro, nel 1984. Egli propose l'uso dei *pausible-clocks* per far sì che fosse possibile, per i vari domini (di clock), comunicare senza metastabilità. Con l'approccio di Chapiro, ogni blocco localmente sincrono genera il proprio clock con un oscillatore ad anello. Ciascun periodo degli oscillatori ad anello è settato in modo da soddisfare i requisiti di velocità del blocco che pilota.

Due potenziali vantaggi dell'approccio *pausible-clocks* sono robustezza e potenza. Le pause ritardano gli istanti di campionamento dei clock fino all'arrivo dei dati da altri domini e ciò elimina la metastabilità. Inoltre, mettere in pausa il clock di un blocco che è in attesa di comunicazione evita che quel blocco dissipi potenza dinamica. Un'ulteriore ottimizzazione potrebbe consistere nell'abbassare V_{DD} durante stalli prolungati per ridurre anche il consumo di potenza statica. Quindi, questo tipo di design potrebbe essere utile nei design cosiddetti *power-critical*, dove cioè il consumo di potenza è l'aspetto fondamentale sul quale tutte le considerazioni di progetto vanno basate.

Esempio

La figura 3.4 illustra un esempio di design GALS di tipo *pausible-clocks*. Ciascun oscillatore ad anello contiene una porta *NAND* per gestire le pause. Il clock del trasmettitore può essere abilitato quando ϕ_T è a 1 e/o il buffer FIFO può accettare un nuovo valore (*ok_to_put* a 1) e/o quando il trasmettitore non sta cercando di spedire informazioni (*ready_to_put* a 0). Quando una di queste tre condizioni è abilitata vediamo che la porta *OR* (la cui uscita si chiama clockT_run in figura 3.4), ha in ingresso almeno un segnale a 1 e quindi può abilitare il funzionamento del clock. Allo stesso modo il clock del ricevitore può essere abilitato quando ϕ_R è a 1 e/o il buffer ha dei dati pronti per essere ricevuti (*ok_to_take* a 1) e/o il ricevitore non sta cercando di leggere nuovi dati (*ready_to_take* a 0). In questo modo un fronte di salita del clock indica che si può procedere.

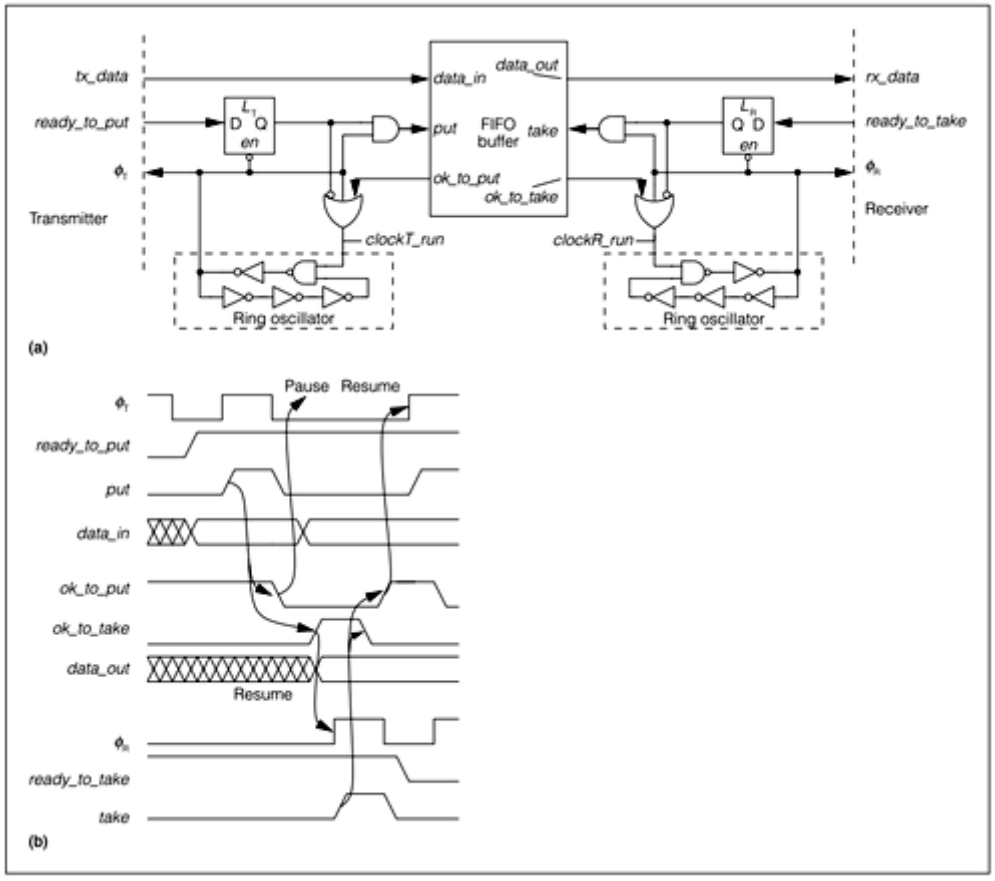


Figura 3.4 – Design GALS di tipo pausible-clocks: (a) circuito e (b) diagramma temporale.

Il diagramma temporale in figura 3.4 mostra il trasferimento di due dati consecutivi. Assumendo che il buffer possa contenere solamente un dato e che all'inizio sia vuoto, il ricevitore è pronto (*ready_to_take* a 1), ma il suo clock è in pausa perché il buffer è vuoto. Il trasmettitore è pronto, avendo pilotato *tx_data* e *ready_to_put* alla fine dell'ultimo ciclo. Mentre il clock del trasmettitore è a 0, il latch L_T è trasparente[(6)], ma la porta *AND* tiene *put* a 0. Quando il buffer è pronto (*ok_to_put* a 1), viene

[(6)]Cioè è come se fosse un cortocircuito.

prodotto un fronte di salita del clock del trasmettitore che setta *put* a 1, riempie il buffer con il primo dato e abbassa *ok_to_put* a 0. A questo punto, il clock del trasmettitore viene messo in pausa perché è immediatamente disponibile il secondo dato ma il buffer è pieno. Nel frattempo, il segnale *ok_to_take* a 1 fa sì che venga riattivato il clock del ricevitore. Il ricevitore registra *rx_data* e setta *take* a 1, segnalando al buffer che i dati sono stati rimossi. Poiché il buffer non è più pieno, *ok_to_put* va a 1 e ciò fa ripartire il clock del trasmettitore in modo che anche il secondo possa essere trasmesso.

Possibili estensioni

Il caso (semplice) di un singolo trasmettitore e ricevitore può essere generalizzato a design in cui ciascun blocco comunichi con molteplici altri blocchi. Yun e Donohue e Yun e Dooply hanno sviluppato un sistema siffatto utilizzando particolari componenti per eseguire la selezione di un dispositivo di input, e usando la mutua esclusione (con un mutex) per gestire il clock. Circuiti progettati in questo modo mettono in pausa il clock finché la metastabilità non si risolve in uno stato stabile (1 o 0). Bormann e Cheung hanno invece sviluppato un design similare che evita l'uso di *arbitri* e *polling* mediante trasferimenti espliciti.

Nel design GALS è fondamentale considerare le latenze della struttura (ad albero) dei clock. Se la latenza per la distribuzione del clock è maggiore di un singolo ciclo, possono verificarsi operazioni invalide dopo l'istante in cui il clock avrebbe dovuto fermarsi. Alcuni ricercatori hanno proposto di aggiungere delay artificiali tra l'interfaccia GALS e i blocchi sincroni per risolvere questo tipo di problema[(7)].

Problematiche nella progettazione

Il design *pausible-clocks* incapsula vincoli temporali cruciali nei dispositivi che incorporano i generatori dei clock, semplificando il riutilizzo dei progetti. Controllando il clock del ricevitore, queste interfacce assicurano che i dati che arrivano al ricevitore soddisfino i requisiti temporali imposti dal ricevitore stesso, quindi eliminando completamente la metastabilità e, una volta che sono state verificate, possono essere riutilizzate senza bisogno di ulteriori analisi.

Molti ricercatori hanno trovato alquanto difficile progettare oscillatori ad anello orientati a robustezza e performance e per questo motivo attualmente il design di tipo *pausible-clocks* viene considerato come tecnologia di nicchia, nelle migliori ipotesi. Per esempio, il periodo di clock può avere un jitter elevato, variando in modo significativo di ciclo in ciclo tra una pausa e l'altra. Questo problema inoltre si espande quando consideriamo la rete di distribuzione dei clock dove i margini temporali si assottigliano ancora di più.

Uno dei potenziali vantaggi che invece offre l'oscillatore ad anello è che le variazioni sul periodo di clock dovrebbero in teoria seguire le tracce delle variazioni dei delay delle porte logiche (almeno in un dato range di utilizzo). Sfortunatamente, gli strumenti CAD standard non prendono in considerazione questo comportamente durante l'analisi e possono di conseguenza forzare il circuito all'approccio conservativo del caso peggiore.

[(7)]Per approfondimenti si veda in bibliografia: [30, 58, 59].

3.2.2. Asynchronous Interfaces

Il secondo tipo di design GALS che consideriamo è chiamato *asynchronous interfaces*. Questo metodo utilizza circuiti conosciuti come *sincronizzatori* per trasferire segnali che arrivano da un dominio temporale esterno verso il dominio locale. Nonostante le interfacce asincrone presentino in genere un basso throughput, questa limitazione può essere ovviata con degli accorgimenti in fase di progetto.

Esempio

La figura 3.5 illustra un esempio di design GALS asincrono. Il diagramma temporale mostra il trasferimento di due dati, dal trasmettitore al ricevitore, assumendo un buffer FIFO inizialmente vuoto. In questo circuito, i segnali di handshake gestiti dal buffer, *ok_to_put* e *ok_to_take*, possono essere messi a 1 in ogni momento, relativamente al clock del trasmettitore e del ricevitore, rispettivamente.

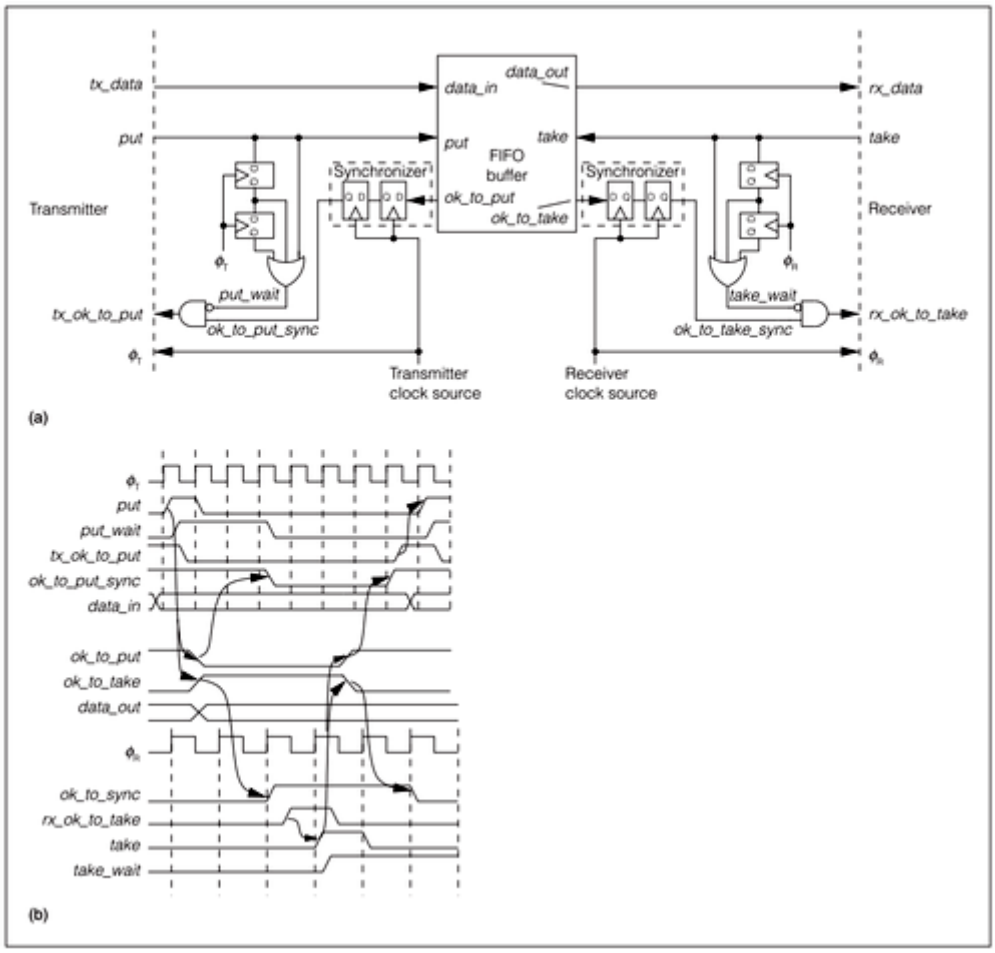


Figura 3.5 – Design GALS asincrono con sincronizzatori: (a) circuito e (b) diagramma temporale.

Questo design utilizza due flip-flop per sincronizzare un segnale con il clock locale ed evitare così la metastabilità. Per occuparsi del delay del sincronizzatore, il segnale *put_wait* evita che il trasmettitore possa spedire dati prima che lo stato del buffer

che segue il *put* precedente si sia propagato attraverso il sincronizzatore. Il segnale *take_wait* ha la stessa funzione per il lato ricevitore. Questo design semplificato può trasferire al più un dato per ogni tre cicli del più lento tra i clock ϕ_T (trasmettitore) e ϕ_R (ricevitore).

Possibili estensioni

Seizovic ha mostrato come sia possibile incrementare il throughput di una interfaccia asincrona inviando le operazioni di sincronizzazione in una pipeline, attraverso un buffer FIFO, assieme ai dati. La probabilità di un fallimento nella sincronizzazione è determinata dal tempo totale in cui i dati sono nel buffer e ciò consente di raggiungere probabilità di fallimento molto basse con elevati throughput per i dati. Un design di questo tipo permette un throughput di un dato per ogni ciclo del più lento tra i clock del trasmettitore e del ricevitore (quindi migliora di circa un fattore 3 le prestazioni rispetto all'esempio in figura 3.5). Boden et al. hanno utilizzato la sincronizzazione con pipeline di Seizovic nel design della rete hardware ad alta velocità *Myrinet*.

Più di recente, Chelcea e Nowick hanno proposto una famiglia generica di sincronizzatori basati su buffer FIFO a bassa latenza. L'idea chiave del loro design è di identificare l'istante in cui il buffer è quasi vuoto (contiene meno dati del numero di flop contenuti nel sincronizzatore) o quasi pieno. I segnali per queste condizioni sono sincronizzati assieme ai soliti segnali *full* e *empty* (pieno e vuoto). Finché la versione sincronizzata di "quasi vuoto" è falsa, il ricevitore può prendere un dato a ogni ciclo, altrimenti può tornare ad usare il segnale *empty* per rimuovere gli ultimi dati dal buffer a velocità minore. Un ragionamento simile si adotta per il trasmettitore. Questa soluzione fa sì che il buffer trasmetta i dati a piena velocità, scegliendo sempre il minimo tra le velocità del trasmettitore e del ricevitore. Questo design supporta combinazioni arbitrarie di componenti comunicanti (sincrone o asincrone), come pure supporta lunghi delay nelle interconnessioni, rendendosi così adatto per design di SoC estesi e con molte differenti (e possibilmente anche variabili nel tempo) frequenze di clock.

Molti progetti recenti cercano il modo per integrare il design sincrono in reti asincrone con il minimo sforzo. Per esempio, il design *Pivot-Point* usa code delay-insensitive per trasmettere dati tra i blocchi locali e la *Pivot-Point* crossbar[(8)].

Problematiche nella progettazione

Il design di tipo *asynchronous interfaces* offre l'approccio più flessibile e forse il più semplice per quanto riguarda l'integrazione con gli strumenti CAD. Il problema principale è costituito dalla modellazione e validazione dei circuiti di sincronizzazione e dall'impatto dei loro delay. I sincronizzatori reali hanno un comportamento più complesso di quello che si trova sui libri di testo e i simulatori di circuiti non hanno una precisione sufficiente per poter verificare se l'affidabilità di questi circuiti sia accettabile o meno. Esistono tuttavia dei recenti modelli si simulazione che affrontano questo problema e sembra ragionevole aspettarsi che il flusso principale di produzione di circuiti GALS utilizzerà sincronizzatori incapsulati in blocchi IP come quelli forniti dalla Fulcrum Microsystems o dalla Silistix, ad esempio. La validazione di

[(8)]Per approfondimenti si veda in bibliografia: [108, 34, 111, 22].

questi blocchi è garantita dai produttori eliminando il problema per il progettista al quale rimane "solo" il compito di assemblarli.

Una buona regola generale per i design di questo tipo è che almeno quaranta delay di porte siano presi in considerazione perché la metastabilità risolva in un valore logico stabile. Per un processo a $0.13\mu m$ con delay di porta di $60ps$, la sincronizzazione aggiunge circa $2.5ns$ di ritardo quando si attraversano domini temporali. Quindi ci aspettiamo che il design GALS asincrono trovi largo utilizzo in progetti SoC che possono permettersi delle latenze extra per la sincronizzazione o che magari hanno basse frequenze di clock. I design per le alte prestazioni richiedono invece l'utilizzo del paradigma *loosely synchronous* descritto di seguito.

3.2.3. Loosely synchronous Interfaces

Il terzo stile di progettazione GALS (*Globally Asynchronous Locally Synchronous*) che affrontiamo prende il nome di *loosely synchronous interfaces* e viene impiegato quando alcuni vincoli sulle frequenze dei blocchi di comunicazione sono conosciuti. In questo stile, il designer sfrutta questi vincoli per assicurare che siano rispettati i requisiti temporali. Questo stile inoltre richiede un'analisi temporale per i percorsi tra trasmettitore e ricevitore ed è meno suscettibile ai cambiamenti dinamici della frequenza del clock. Comunque, questa analisi rende superfluo l'handshaking durante il trasferimento dei dati e quindi il circuito risultante può raggiungere prestazioni più elevate e avere più latenze di tipo deterministico rispetto ai circuiti creati con gli altri metodi.

Esempio

Un design loosely synchronous sfrutta una delle relazioni temporali conosciute che abbiamo descritto in precedenza. Il caso più semplice è una relazione di tipo *mesochronous*, nella quale le frequenze sono perfettamente accoppiate e c'è una differenza di fase sconosciuta, ma stabile. Nella pratica, ciò avviene quando i clock derivano dalla stessa sorgente ma la latenza di arrivo a ciascun blocco è differente.

L'esempio di *mesochronous* mostrato in figura 3.6 è basato sullo schema *Stari* (Self-Timed at Receiver's Input), nel quale i clock ϕ_T e ϕ_R sono derivati dalla stessa sorgente. Il ricevitore utilizza un buffer FIFO di tipo self-timed per compensare la differenza di fase. La chiave per ottenere elevate prestazioni consiste nell'inizializzare il buffer in modo che sia mezzo pieno. Durante le operazioni, il trasmettitore mette un dato nel buffer a ogni ciclo e, allo stesso modo, il ricevitore prende un dato dal buffer, sempre a ogni ciclo. Nessuno dei due ha bisogno di controllare il segnale di stato del buffer (il buffer è supposto essere sufficientemente veloce), ma il buffer rimarrà alla metà della propria capacità $\pm$ un dato perché le frequenze sono accoppiate. Se necessario, informazioni di controllo del flusso di alto livello possono essere incorporate nei dati (per esempio definendo un bit di validità) al posto di fermare il trasmettitore.

Per ottenere un buffer mezzo pieno, è necessaria un'inizializzazione speciale. All'inizio, viene inviato un segnale globale di *reset*, che può necessitare di sincronizzazione. Il blocco *TX_INIT* attende un numero di cicli prefissato per avere la garanzia che il *reset* sia stato completato ovunque e poi abilita il trasmettitore settando *tx_enable* a 1. Il trasmettitore a quel punto comincia a spedire dati. Dopo l'arrivo del primo dato, *empty* va a 0. Poiché il trasmettitore e il ricevitore possono avere variazioni di

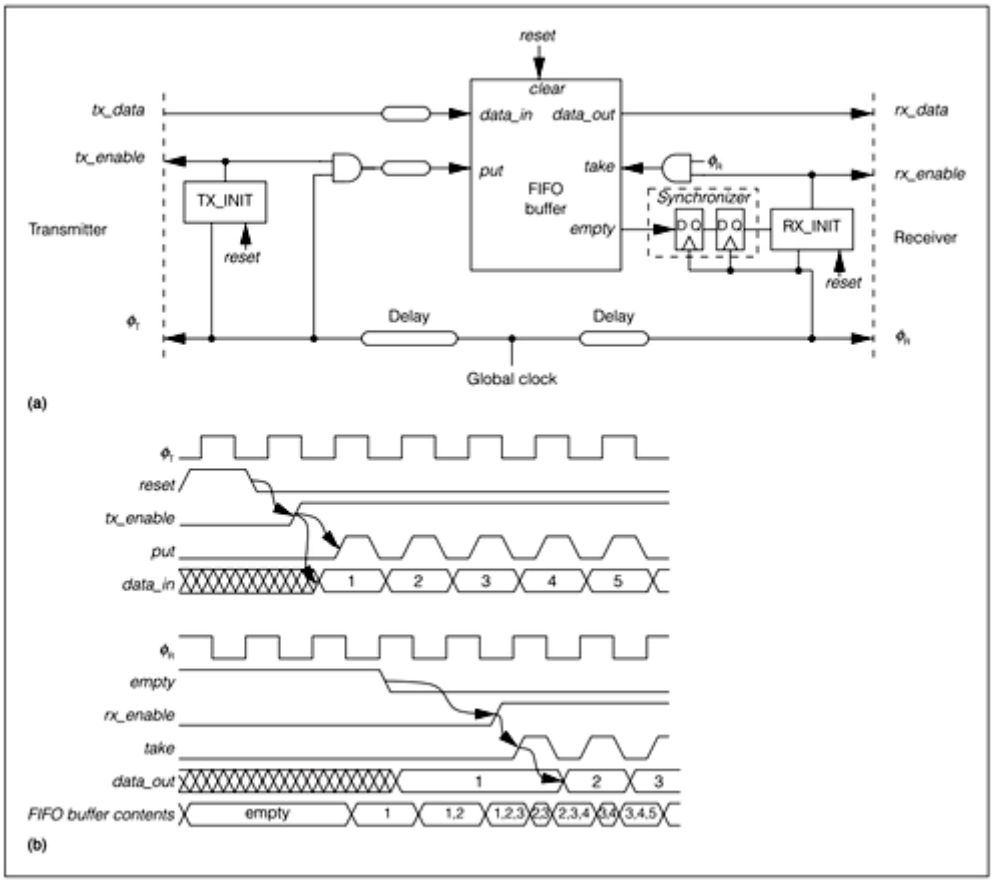


Figura 3.6 – Design GALS loosely synchronous, *mesochronous*: (a) circuito e (b) diagramma temporale.

fase arbitrarie, questo cambio di *empty* è asincrono rispetto al clock del ricevitore e deve quindi essere sincronizzato. Dopo la latenza dovuta al sincronizzatore, il blocco *RX_INIT* riceve il segnale, attende i cicli necessari per far sì che il buffer raggiunga lo stato di "mezzo pieno", quindi setta *rx_enable* a 1. Nel ciclo successivo del ricevitore, questo comincerà a rimuovere dati alla stessa frequenza con cui il trasmettitore li inserisce e a questo punto, nessuna sincronizzazione sarà più necessaria.

Possibili estensioni

Se le variazioni temporali tra il trasmettitore e il ricevitore sono sufficientemente piccole, l'interfaccia di tipo *mesochronous* può essere ampiamente ottimizzata. Una possibile versione, dovuta a Chakraborty e Greenstreet, consiste in un'interfaccia simile con un buffer di tipo clocked, a singolo stadio, che può tollerare circa fino a due periodi di clock di incertezza di fase tra il trasmettitore e il ricevitore. Il buffer proviene da un circuito pilotato da eventi che controlla i clock di trasmettitore e ricevitore e genera una pulsazione in una finestra temporale sicura.

Nel design di tipo *ratiochronous* invece, i blocchi sincroni utilizzano clock che sono multipli (razionali) esatti l'uno dell'altro. I metodi del *mesochronous* possono essere estesi per includere questo caso. Per esempio, un design potrebbe includere blocchi che operano a $300MHz$ e $700MHz$, entrambi derivati da multipli di un riferimento a

$100MHz$. In questo caso, la relazione di fase tra i due clock varia in modo predicibile e periodico. Chakraborty e Greenstreet hanno presentato un design che utilizza moltiplicatori di tipo *binary-rate* per il clock (di trasmettitore e ricevitore) che più rapidamente genera una approssimazione dell'altro, che diventa poi l'input del generatore di clock di tipo *event-driven* appena menzionato.

Un approccio alternativo consiste nel permettere normali trasmissioni tranne quando i dati arriverebbero circa in coincidenza del clock del ricevitore. Merkie et al. hanno proposto un modello di trasmissione preventiva, per i cicli non sicuri, esaminando e modificando il protocollo di comunicazione e sfruttando la relazione di periodicità tra le fasi dei clock. Se il trasmettitore non trasmette mai un dato in un ciclo non sicuro, allora non è necessaria un'interfaccia di sincronizzazione. Comunque, questa soluzione dipende dal controllo della differenza di fase globale tra blocchi comunicanti e porta a vincoli di tempo molto restrittivi.

Nel design di tipo *plesiochronous*, il trasmettitore e il ricevitore hanno clock che operano a frequenze molto ravvicinate. Queste possono andare alla deriva molto lentamente, finendo per violare i vincoli temporali del ricevitore. E' comunque possibile individuare quando sta per avvicinarsi uno stato non sicuro e operare in modo da correggere il problema, tornando in uno stato sicuro. Oltretutto, poiché la deriva della fase avviene lentamente, eventi di questo tipo insorgono con una frequenza molto bassa e quindi la velocità delle operazioni di correzione del problema non è cruciale. Non è necessaria alcuna sincronizzazione in uno stato sicuro, quindi non si hanno penalizzazioni (sotto forma di latenze aggiuntive) durante il normale trasferimento dei dati.

Se il trasmettitore e il ricevitore stanno operando a frequenze stabili ma incorrelate, l'uno può stimare la frequenza dell'altro e viceversa. Questa stima fornisce un numero razionale che approssima il rapporto, permettendo di ricorrere all'uso dei metodi relativi al paradigma *ratiochronous*. Quindi, le tecniche di tipo *plesiochronous* possono gestire il restante sfasamento tra le frequenze[(9)].

Problematiche nella progettazione

La necessità di alte frequenze di clock e basse latenze dei circuiti ad alta velocità li rende buoni candidati per le tecniche dell'approccio *loosely synchronous*. Comunque, per determinare la taglia ottimale dei buffer FIFO, è necessario eseguire un'analisi temporale per limitare la dimensione delle differenze di fase relative tra trasmettitori e ricevitori. Nonostante questo tipo di analisi temporale non sia ancora una pratica comune nei design on-chip, è invece una pratica standard per i sistemi che sfruttano comunicazioni inter-chip di tipo source-synchronous (ad esempio, nelle DRAM sincrone). Questa è un'area dove ci si aspetta di vedere crescere lo sviluppo di strumenti CAD di supporto, che aiutino i progettisti nell'analisi di chip che incorporano molti diversi domini temporali.

3.3. Ultime considerazioni

Il design GALS si avvale dell'estesa infrastruttura utilizzata nel design sincrono e allo stesso tempo evita il problema di dover distribuire un unico segnale globale

[(9)]Per approfondimenti si veda in bibliografia: [2, 41].

di clock (low-skew). Una metodologia GALS è l'approccio naturale per il design di SoC, poiché consente l'integrazione di blocchi progettati indipendentemente e che operano a frequenze diverse. In aggiunta, alcuni approcci GALS permettono una facile gestione di aspetti quali il *dynamic voltage scaling* e altre tecniche di riduzione del consumo di potenza.

Nonostante il paradigma pausible-clocks sia attraente per l'eliminazione (per costruzione) dei problemi legati alla metastabilità, non si integra bene negli strumenti CAD attualmente esistenti e non è facilmente scalabile per il design di circuiti con clock ad alta velocità. E' alquanto probabile quindi che questo approccio non troverà ampio consenso, ma può comunque essere preso in considerazione per la sua capacità di spegnere completamente i clock nei momenti di non operatività, qualora il risparmio di potenza sia la principale caratteristica richiesta al circuito.

Interfacce completamente asincrone, invece, offrono la massima flessibilità. Anche se gli attuali strumenti CAD non possiedono ancora le funzionalità necessarie per supportare le interconnessioni asincrone, è pur vero che gli strumenti commerciali stanno evolvendo in questa direzione, con strumenti che possono controllare circuiti composti da domini multipli per scovare errori strutturali e di protocollo.

Ci si aspetta che i produttori di CAD e IP comincino a offrire supporto per queste problematiche, tanto più, quanto più grande sarà la domanda nel campo del design di SoC e NoC di grandi dimensioni.

Un ulteriore problema che si riscontra con quei design GALS che impiegano *arbiter* e sincronizzatori è che presentano un comportamento non deterministico per costruzione e ciò complica sia il test che la validazione del design. Su questo aspetto verranno eseguiti studi e ricerche approfondite in futuro.

L'approccio di tipo *mesochronous* e le altre tecniche loosely synchronous offrono le più alte prestazioni poiché rimuovono i delay dovuti alla sincronizzazione dai percorsi critici. Comunque, questi metodi richiedono analisi di tempo che gli strumenti CAD standard non permettono e quindi ci si attende che il design di tipo loosely synchronous verrà utilizzato per applicazioni di tipo *performance-critical* che giustifichino l'ulteriore sforzo di progettazione.

I venditori di IP possono aiutare i progettisti di ASIC a sfruttare i circuiti di tipo loosely synchronous, incapsulandoli in blocchi predefiniti e fornendo strumenti dedicati per la validazione basati sui software per l'analisi temporale e sui CAD standard.

Il design GALS fronteggia un problema simile a quello dell'uovo e della gallina: la maggior parte dei designer non ha intenzione di migrare finché non siano disponibili strumenti CAD adeguati, mentre le compagnie che li producono sono riluttanti a fornire strumenti per una tecnologia che non è ancora ampiamente usata.

L'approccio di partizionamento del GALS comunque è inevitabile. Il design con interfacce puramente asincrone è quello che richiede il cambiamento minimo nei blocchi locali e che risolve il problema del clock skew. Per questo motivo, questo stile è quello che ha le potenzialità per essere il primo a ottenere un adeguato supporto (CAD) e quindi diventare l'approccio dominante.

4. Metodi di Sintesi

Nei capitoli precedenti abbiamo visto alcuni stili di implementazione di alcune delle principali componenti dei circuiti asincroni. Abbiamo anche sottolineato come la mancanza di strumenti CAD renda difficile la progettazione di interi circuiti e da ciò la necessità, per ricercatori e progettisti, di ricorrere a strumenti di tipo diverso.

In questo capitolo quindi esporremo i principali concetti legati alla sintesi di circuiti asincroni. Analizzeremo il ruolo dei linguaggi CSP[1] e introdurremo in breve Tangram e Balsa, quest'ultimo in un contesto di sintesi orientata all'ottimizzazione delle prestazioni.

Per chiarire subito l'argomento, possiamo fare un parallelo con il mondo informatico. Prendiamo ad esempio il codice in figura 4.1:

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 a = [x for x in range(10) if x&1]
5 for b in a:
6     print b,
```

Figura 4.1 – Esempio in Python.

Lo scopo di questo codice è creare un vettore di elementi dispari (riga 4) e scriverli nella console (righe 5-6). Quando Python si trova a interpretare la riga 4 per popolare il vettore (usando in questo caso una *list-comprehension*), l'interprete eseguirà del codice che è scritto in C. Allo stesso modo, per stampare il vettore, richiamerà altre funzioni, sempre scritte in C. Il concetto è che noi non siamo interessati al modo in cui Python provvede a inizializzare il vettore, né a come esegue il ciclo *for* per farne la stampa. Ciò che invece ci interessa è che il risultato in uscita sia la stringa "1 3 5 7 9" e quindi che il codice funzioni correttamente. Abbiamo usato una descrizione di alto livello (Python) e lasciamo che sia lui a preoccuparsi di eseguire i nostri comandi, fornendoci il risultato.

Allo stesso modo, esistono framework e linguaggi di alto livello studiati per prendere in input la descrizione di un circuito e restituire in output la sua traduzione in porte logiche. Il progettista non si interessa tanto a come il framework lavori o a che librerie e linguaggi utilizzi. Ciò che gli preme è di poter fornire una specifica di alto livello e sapere che la traduzione in porte logiche che riceverà dal framework sia corretta[2].

[1] CSP: Communicating Sequential Processes.

[2] Ovviamente, ci riferiamo al caso generale. Quando poi si ottimizzano i circuiti allora è fondamentale capire come opera il linguaggio.

4.1. Introduzione

Quasi tutto il lavoro nella modellazione e sintesi di alto livello di circuiti asincroni è basato sull'uso di un linguaggio che appartiene alla cosiddetta famiglia di linguaggi CSP. Al contrario, per la tecnologia sincrona vengono utilizzati altri tipi di linguaggi come ad esempio VHDL e Verilog. I circuiti asincroni sono caratterizzati da un alto tasso di concorrenza[3] e la comunicazione attraverso i moduli è basata, come abbiamo visto, sui canali handshake. Di conseguenza, un linguaggio di descrizione hardware per un circuito asincrono dovrebbe fornire adeguate *primitive* che supportino queste due caratteristiche. I linguaggi CSP sono pensati per questo tipo di esigenza e le loro caratteristiche chiave sono:

- Processi concorrenti
- Composizione sequenziale e concorrente di istruzioni all'interno di un processo
- Passaggio sincrono dei messaggi in canali point-to-point (supportati dalle primitive *send*, *receive* e – possibilmente – *probe*)

I CSP appartengono a una famiglia di linguaggi più ampia che si occupa della programmazione di sistemi concorrenti in generale. Tra i vari linguaggi che appartengono a questa famiglia possiamo citare OCCAM, LOTOS e CCS, così come linguaggi appositamente pensati per il design di circuiti asincroni, tipo Tangram, CHP e Balsa.

Nonostante siano pensati per eseguire il medesimo compito, questi linguaggi impongono una differente impostazione iniziale di progetto: Tangram, ad esempio, è di tipo *control-driven* e utilizza un processo chiamato *syntax-directed compilation*. La traduzione di un programma scritto in Tangram è una struttura di componenti handshake. Tangram è stato creato nei Philips Research Laboratories ed è stato sfruttato per costruire diversi (significativi) chip asincroni progettati proprio in Philips. Per citare un esempio, alcuni circuiti per smart card sono stati creati con Tangram.

Al California Institute of Technology invece, è stato sviluppato CHP (Communicating Hardware Processes), assieme a tutto un set di strumenti che supportano la progettazione (in parte automatica, in parte manuale) di implementazioni (a livello di transistor e altamente ottimizzate) di circuiti di tipo QDI 4-phase dual-rail. CHP ha una sintassi molto simile a a CSP, mentre Tangram ha una sintassi che ricorda più i linguaggi di programmazione tradizionali, caratterizzati da parole chiave (keywords). Nonostante ciò, sono in essenza entrambi molto simili a CSP.

4.2. Concorrenza e trasmissione di messaggi in CSP

La parte dell'acronimo *CSP* che concerne i processi sequenziali denota che ciascun processo è descritto da un programma le cui istruzioni sono eseguite in sequenza, una dopo l'altra. Il ";" è utilizzato per separare le istruzioni (come avviene in C, C++, Java, C#, etc.) e può essere visto come un operatore che combina istruzioni in programmi. In questo senso, un processo CSP è molto simile a un processo VHDL. Comunque, CSP permette anche la composizione parallela di istruzioni in un processo. Il simbolo "||" denota la composizione parallela. Questa caratteristica non si trova in VHDL,

[3] Intesa come la capacità di eseguire molteplici operazioni contemporaneamente.

mentre ad esempio il costrutto *fork-join* di Verilog permette la concorrenza – a livello di istruzioni – in un processo.

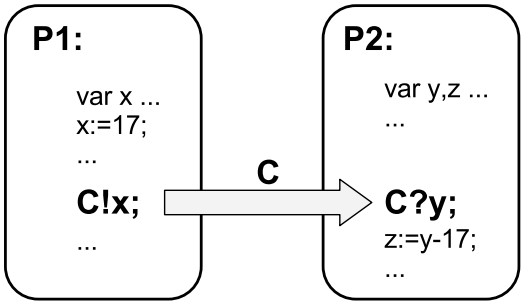


Figura 4.2 – Due processi P1 e P2 connessi da un canale C. Il processo P1 invia il valore della sua variabile x nel canale C, P2 riceve il valore e lo assegna alla sua variabile y.

La parte dell'acronimo *CSP* che invece concerne la comunicazione si riferisce al passaggio di messaggi sincroni utilizzando canali point-to-point, come illustrato in figura 4.2, dove due processi P1 e P2 sono connessi tramite un canale C. Utilizzando un'istruzione *send* (C!x), P1 invia (l'invio è rappresentato dal "!") il valore della sua variabile x nel canale. P2, utilizzando un'istruzione *receive* (C?y) riceve (la ricezione è invece rappresentata dal "?") il valore e lo assegna alla propria variabile y. Il canale è senza memoria e il trasferimento del valore di x da P1 a P2 è un'istruzione atomica[(4)].

Questo ha l'effetto di sincronizzare i processi P1 e P2. Qualunque sia il primo dei due che arriva, dovrà aspettare che l'altro completi il trasferimento (del valore di x). Per indicare questo tipo di situazione solitamente si utilizza il termine francese *rendezvous*.

Quando un processo esegue un *send* [*receive*], si lega alla comunicazione e si sospende finché il processo all'altra estremità del canale esegue la propria istruzione di *receive* [*send*]. Questo comportamento potrebbe non essere sempre desiderabile e infatti CSP è stato esteso con una funzione *probe* che permette ad un processo che si trovi sul lato passivo del canale, di verificare se c'è una comunicazione in attesa senza legarvisi. La sintassi per eseguire il *probing* di un canale C è semplicemente $\overline{C}$.

Come informazione ancillare facciamo notare che alcuni linguaggi per la programmazione di sistemi concorrenti assumono canali di tipo buffer (con capacità limitata o meno). In questi casi il canale si comporta come una FIFO e i processi comunicanti non si sincronizzano quando comunicano. Di conseguenza questa forma di comunicazione prende il nome di *asynchronous message passing*.

Tornando al nostro discorso, è ovvio che l'implementazione fisica di un canale senza memoria sia un semplice insieme di linee con un protocollo per la sincronizzazione tra processi comunicanti. E' altresì ovvio che ciascuno dei protocolli che abbiamo considerato nei precedenti capitoli possa essere utilizzato per questo scopo. Il *Synchronous message passing* è quindi un utile costrutto che supporta la modellazione di alto li-

[(4)]Per chi non avesse familiarità con questo tipo di terminologia, un'istruzione si dice *atomica* quando non è interrompibile da un altro processo mentre è in esecuzione. Questo tipo di istruzioni possono essere eseguite nella programmazione multi-threading o multi-processing senza bisogno di usare mutex o meccanismi di lock.

vello di circuiti asincroni facendo un'astrazione degli esatti dettagli della codifica dei dati e del protocollo di handshake utilizzato nel canale.

Sfortunatamente, sia VHDL che Verilog mancano di queste primitive. E' possibile scrivere codice di basso livello che implementi l'handshaking, ma è altamente indesiderabile mischiare dettagli di livello così basso con del codice il cui scopo è quello di catturare il comportamento ad alto livello di un circuito. Per quanto riguarda VHDL è anche possibile avvalersi di alcuni pacchetti sviluppati con lo scopo di estendere il linguaggio, in modo da fornirgli primitive *send*, *request* e *probe* e poter scrivere così del codice che somiglia molto a quello che viene scritto per i linguaggi delle famiglie CSP.

Vediamo ora un brevissimo esempio, scritto in Tangram, per dare al lettore un'idea di come funzionano questi tipi di linguaggio. Il codice seguente (figura 4.3) realizza un modulo che calcola il massimo comun divisore (GCD) di due valori. L'istruzione "*do x<>y then ... od*" rappresenta un ciclo *while*. Il modulo ha un canale di input dal quale riceve i due operandi, e un canale di output nel quale inserisce il risultato.

```
  int = type [0..255]
& gcd_if : main proc (in?chan <<int,int>> & out!chan int).
  begin x,y:var int ff
  | forever do
      in?<<x,y>>
    ; do x<>y then
        if x<y then y:=y-x
               else x:=x-y
        fi
      od
    ; out!x
    od
  end
```

Figura 4.3 – Codice Tangram per il GCD con istruzioni *while* e *if*.

Come si può notare il codice è abbastanza leggibile. La sintassi ci ricorda un po' Delphi, un po' Pascal, ma in ogni caso è possibile, anche con moderata esperienza, comprendere i passaggi di un semplice algoritmo come questo con una rapida occhiata.

Non vogliamo però dilungarci troppo sui dettagli di tipo semantico e sintattico di Tangram o di altri linguaggi perché ci preme piuttosto di mostrare il prossimo esempio, grazie al quale avremo anche l'opportunità di introdurre Balsa, un linguaggio della famiglia CSP che viene utilizzato per la sintesi di circuiti.

4.3. Sintesi syntax-directed, performance-driven di un processore asincrono

Molti sistemi moderni sono sintetizzati utilizzando strumenti CAD e anche se la maggioranza di questi sono di tipo sincrono, l'interesse verso i circuiti asincroni è in continua crescita per via delle qualità che possiedono, come, ad esempio, la bas-

sa emissione di radiazione elettromagnetica, e la capacità di gestire con successo le variazioni di processo.

Ci sono diversi sistemi per la sintesi di circuiti asincroni che sono disponibili. Alcuni, come *Petrify* e *Minimalist*, sono dedicati alla sintesi di controller asincroni. Altri, come *TAST* e *CHP*, si occupano sia dei controller che dei datapath e sono più completi, anche se a volte richiedono un maggiore intervento (o una guida) del progettista nella fase di sintesi.

Tangram e Balsa sono invece sistemi completamente automatizzati e sono stati usati con successo nella sintesi di circuiti su larga scala utilizzando la *syntax-directed compilation*. L'obiettivo di questa sezione è focalizzarsi sull'approccio di sintesi ed esaminare le possibilità di ottimizzare le performance del circuito generato.

4.3.1. Syntax-directed synthesis

La traduzione di tipo *syntax-directed*, orientata alla sintassi, è uno strumento di sintesi molto potente. Il processo di sintesi consiste nel compilare delle descrizioni scritte in un linguaggio di alto livello, producendo una rete di moduli già progettati. Questo approccio produce quella che viene chiamata una compilazione "trasparente", cioè c'è una relazione biunivoca tra un costrutto linguistico del linguaggio e la rete di moduli che lo implementano. Questa mappatura diretta garantisce al progettista la flessibilità a livello di linguaggio che poi si manifesta anche nel circuito risultante. Variazioni incrementali a livello del linguaggio risultano in variazioni predicibili nella relativa implementazione. Le specifiche del codice sorgente possono avere grande impatto sulle prestazioni, sul consumo di potenza e sull'area occupata dal circuito risultante.

In molti casi, il sistema di sintesi può valutare il modulo generato per fornire all'utente una stima sulle prestazioni e sull'area occupata dal circuito. Un progettista esperto può quindi ottimizzare il circuito risultante in termini di prestazioni, area e consumo di potenza, semplicemente scegliendo la specifica più adatta. La *syntax-directed translation* è stata utilizzata con ottimi risultati nella sintesi di diversi sistemi embedded, compresi la smart card G3Card SoC, un microprocessore asincrono MIPS e l'ARM996HS, il primo ARM sintetizzabile asincrono messo in commercio.

La G3Card SoC è un buon esempio per un sistema asincrono embedded. Un prototipo è stato fabbricato in un processo a $0.18\mu m$ ed è risultato perfettamente funzionante già alla prima implementazione in silicio. Contiene due differenti implementazioni di un processore asincrono full-featured, ARM compatibile, un'unità di protezione della memoria (MPU), un'interfaccia asincrona verso una RAM sincrona standard, un coprocessore di sincronizzazione e diverse periferiche. Tutte queste componenti sono state sintetizzate utilizzando il sistema di sintesi Balsa, al quale è dedicata la prossima sezione.

4.3.2. Il Balsa synthesis system

Balsa è un sistema di sintesi che genera circuiti macromodulari *puramente* asincroni, chiamati circuiti handshake. Fu proposto all'inizio per essere utilizzato in unione a Tangram (sul quale è fortemente basato) e offre un paradigma molto attraente per la sintesi di circuiti. Complesse descrizioni, scritte in codice sorgente, sono tradot-

te in un circuito costituito da istanze di componenti handshake composte con stile macromodulare.

```
procedure buffer
(
  parameter DataType : type;
  input  in  : DataType;
  output out : DataType
) is
  variable buf : DataType
begin
  loop
    in -> buf;
    out <- buf
  end -- loop
end -- procedure buffer
```

Figura 4.4 – Codice Balsa per un buffer 1-place.

La figura 4.4 mostra come un semplice buffer 1-place è specificato in Balsa. La specifica è parametrizzata nel tipo di dati che sono trattenuti dal registro. I dati sono memorizzati in una variabile (*buf*) e l'operazione è descritta come una ripetizione infinita (*loop*) di due azioni: dati di input ($\rightarrow$) dal canale (*in*) in *buf*, e in seguito (";") output dei dati ($\leftarrow$) contenuti in *buf* al canale (*out*).

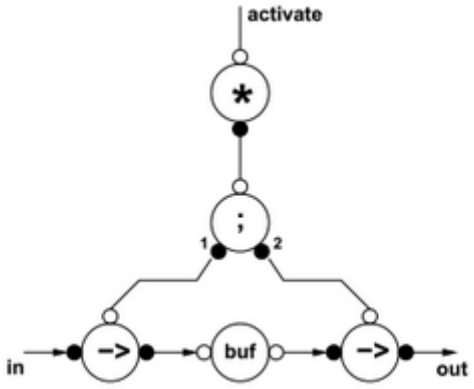


Figura 4.5 – Circuito handshake del buffer 1-place.

La figura 4.5 mostra il circuito handshake per un registro di pipeline sintetizzato con Balsa. I dati sono memorizzati in un latch (*buf*) e un componente *sequencer* (";") è utilizzato per scrivere ($\rightarrow$) e leggere ($\rightarrow$) su *buf*. Un componente *loop* ("*") attiva il *sequencer* a ripetizione. La mappatura biunivoca con i costrutti del linguaggio è molto chiara e anche se il circuito è composto da un certo numero di componenti handshake, non è un circuito complicato. I componenti handshake nella figura sono tutti molto semplici: il componente *transferrer*[5] ($\rightarrow$) è implementato con sole linee,

[5]Un componente transferrer è dotato di tre porte A (Activation), B (input) e C (output). Gestisce il trasferimento di un dato (su richiesta in A) da B a C. Dopo il trasferimento, invia un Acknowledge ad A per comunicare il termine dell'operazione.

il *loop* richiede una porta *NOR* e il *sequencer* consiste di un C-element e di una porta *AND*.

I componenti handshake sono selezionati da un insieme relativamente piccolo e sono di semplice implementazione. Sono interconnessi tramite canali: ciascun canale connette una porta attiva di un componente a una porta passiva di un altro, dove il tipo di porta (attiva o passiva) indica la direzione dell'handshake. Una porta attiva inizia l'handshake (mandando un *request*) mentre una porta passiva lo conferma (con un *acknowledge*).

I canali possono trasportare dati che scorrono o nella stessa direzione del *request* (canale push) o in direzione opposta (canale pull).

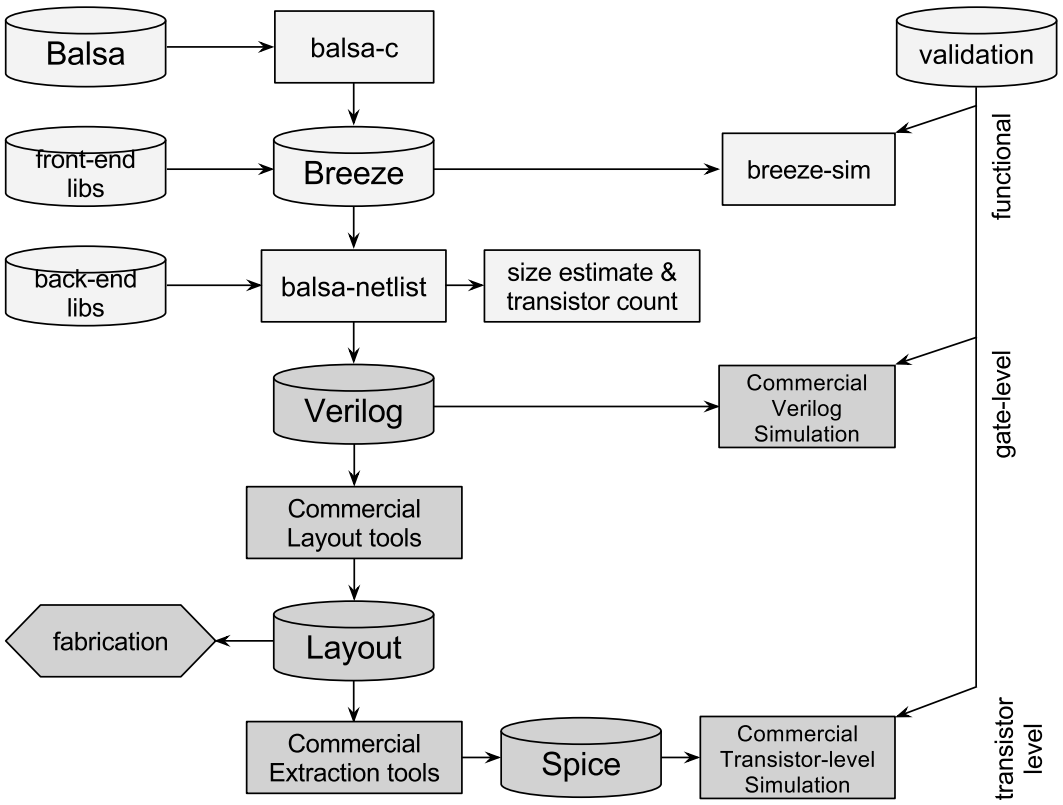


Figura 4.6 – Flusso di design Balsa.

La generazione di un circuito handshake è il primo passo in un flusso di design di Balsa, come mostrato in figura 4.6. Il compilatore Balsa genera una *netlist* intermedia, che può essere utilizzata per la validazione funzionale o per la stima approssimativa delle prestazioni. Il *netlister* Balsa genera a sua volta una *netlist* strutturale Verilog basata su una determinata cella di libreria, sulla codifica dei dati e sullo stile asincrono selezionati. La *netlist* Verilog può essere processata con un layout commerciale e con strumenti di estrazione per un'ulteriore validazione e fabbricazione.

Il SoC G3Card sintetizzato con Balsa è stato basato sul processore *SPA*, un processore implementato usando solamente metodi di sintesi, 32 bit, 100% ARM compatibile. *SPA* è stato implementato come una semplice pipeline a 3 stadi, con uno stile di implementazione simile all'ARM7. Sia il dual-rail (QDI) che il bundled-data

(single-rail data encoding, assunzioni temporali data-boundled) sono stati implementati partendo dalla stessa specifica Balsa.

L'obiettivo principale dell'implementazione dual-rail del processore era quello di riuscire a farne un'analisi di potenza e quindi è stato implementato un circuito abbastanza bilanciato. Le prestazioni non sono uno dei target principali per le smart card, comunque lo SPA è risultato molto più lento di quando si pensasse. La prossima sezione esplora alcune tecniche che vengono adottate proprio per ovviare a questo problema.

4.3.3. Performance-driven synthesis

Le tecniche introdotte in questa sezione hanno lo scopo di ottimizzare le prestazioni di circuiti sintetizzati con Balsa. Nonostante l'area del circuito non sia considerata un fattore di primaria importanza, nella sezione relativa ai risultati mostreremo che è possibile ottenere una sua significativa riduzione. Le tecniche presentate si fondano sull'uso di nuovi componenti handshake che eliminano la sincronizzazione non necessaria tra dati e controllo e permettono un maggiore tasso di concorrenza.

Controllo efficiente della pipeline

Quasi tutti i moderni processori embedded utilizzano (almeno) una pipeline, quindi gli strumenti di sintesi di circuiti asincroni devono generare una efficiente logica di controllo per le pipeline. Balsa non ha speciali costrutti di linguaggio o componenti specifiche per implementare pipeline, infatti gli stadi di pipeline sono di solito specificati all'interno di procedure e i registri vengono implementati usando delle variabili convenzionali. Balsa inoltre non permette letture/scritture concorrenti sulla stessa variabile e ciò significa che, quando una variabile è utilizzata in un registro di una pipeline, gli stadi su ciascun lato della variabile normalmente non possono processare altri dati nello stesso momento.

Come indicato prima, i registri sono variabili all'interno di ciascuno stadio e sia il registro di input che quello di output vengono utilizzati. Questa struttura essenzialmente implementa una pipeline occupata a metà: una pipeline con il doppio di stadi, una metà dei quali processa dati mentre l'altra è vuota, in alternanza. Senza gli stadi vuoti, processi adiacenti non potrebbero operare in concorrenza e questo limiterebbe severamente il throughput della pipeline.

L'uso di nuovi componenti handshake fornisce un'implementazione della pipeline molto più efficiente. I registri non sono più variabili generiche: essi vengono sempre scritti da uno stadio e letti in quello seguente. Questo pattern di scrittura/lettura permette l'uso di una singola variabile come un vero registro di pipeline. In questo caso i registri sono specificati fuori dagli stadi di processo e gli stadi contengono solo la logica di processo. I registri della pipeline implementati utilizzando i nuovi componenti handshake sono risultati molto semplici, e operano assai meglio rispetto ad esempio a dei controller altamente ottimizzati.

In figura 4.7 possiamo vedere il comportamento del (pipeline) controller. R e A rappresentano i segnali di request e acknowledge utilizzati per implementare i canali di *in* e *out*, e Lt rappresenta il segnale che abilita il latch.

Il comportamento descritto in figura mostra che le implementazioni dei controller della pipeline sintetizzati con Balsa utilizzano un protocollo efficiente, completamen-

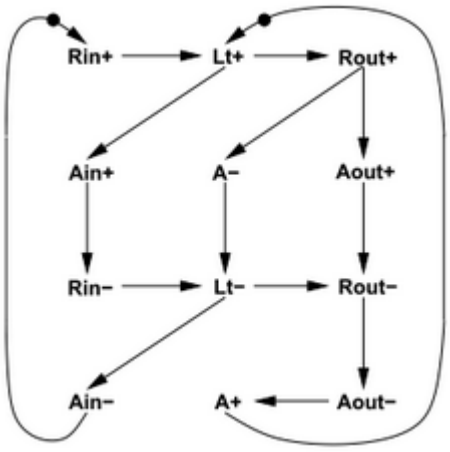


Figura 4.7 – Controllo della pipeline.

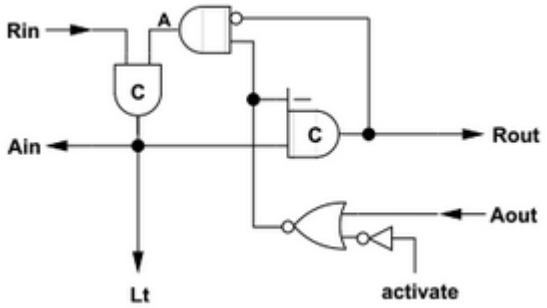


Figura 4.8 – Circuito handshake di controllo della pipeline.

te disaccoppiato, di tipo *request-activated*. La figura 4.8 mostra l'implementazione del controller del registro così come è stato generato dal back-end di Balsa. Il registro è inizializzato dal segnale *activate*.

True asynchronous operation

Le strutture pipeline descritte sopra operano in modo pseudo-sincrono, cioè i trasferimenti di tutti i dati da uno stadio a quello successivo avvengono in simultanea, come in un sistema sincrono. Anche se non c'è un clock globale, i dati avanzano nella pipeline a passi discreti, usando canali di handshake locali. Questa operazione regolare (nel senso del ritmo) è facile da comprendere e analizzare ma può ridurre la prestazione globale del processore sintetizzato.

Le pipeline di tipo true asynchronous non impiegano questo tipo di passi, come si nota in figura 4.9. Ciascuna unità all'interno di uno stadio della pipeline può progredire col proprio ritmo, eseguendo handshake individualmente con le unità che appartengono agli stadi precedente e successivo. Ciò significa che, ad esempio, dati differenti spediti da unità appartenenti al *Decoder* possono arrivare nello stadio *Execute* in tempi completamente differenti. Di consequenza, unità differenti, appartenenti allo stadio *Execute*, possono operare su dati che corrispondono a diverse istruzioni, dando ad alcuni di loro un certo vantaggio. Data la natura elastica delle

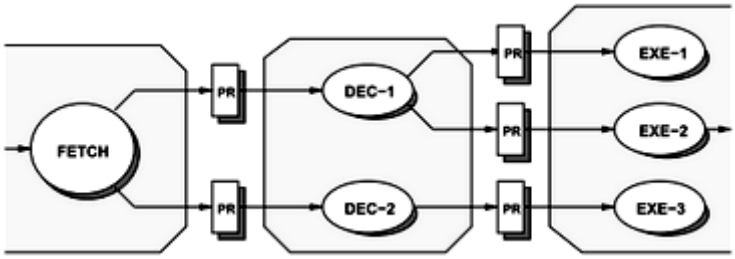


Figura 4.9 – True asynchronous pipeline.

pipeline asincrone, operare in modalità *true asynchronous* risulta vantaggioso nella maggioranza dei casi.

Data-driven operation

Uno degli svantaggi della *syntax-directed synthesis* è l'overhead imposto ai circuiti dall'approccio di tipo *control-driven* della traduzione. I dati e il controllo sono spesso sincronizzati e, poiché spesso le procedure di controllo impiegano più tempo rispetto a quelle che operano sui dati, ciò riduce le prestazioni del circuito perché i dati devono attendere finché il controllo non è pronto.

```
doRegisterRead;
case instruction of
add then
    doShift;
    doAlu
| mul then
    doMul
| ldr, str then
    doMemAccess
end;
doRegisterWriteBack

(a)
```

```
doRegisterRead ||
steerRegData ||
doShift ||
doAlu ||
doMul ||
doMemAccess ||
multiplexResults ||
doRegisterWriteBack

(b)
```

Figura 4.10 – Codice Balsa nello stadio di esecuzione.

La figura 4.10(a) mostra un segmento di una descrizione semplificata in Balsa dello stadio *Execute* del processore *SPA*. Questo è un esempio di una descrizione di tipo *control-driven*. La traduzione fornisce un circuito handshake consistente di un albero di componenti di controllo che dirigono il movimento dei dati attraverso il datapath.

La figura 4.11 illustra il corrispondente circuito. *Transferrers*[(6)] ($\rightarrow$) sono utilizzati per controllare il flusso dei dati attraverso il datapath. Sapendo che l'albero di controllo garantisce la mutua esclusività delle operazioni, un componente *Merge* ($|$) non controllato può essere usato per fondere i risultati delle differenti unità nel registro *write-back*. A ogni passo, il controllo inizia un'operazione e aspetta finché il risultato è pronto, prima di cominciare quella successiva. Il circuito di controllo è generato come parte della traduzione *syntax-directed* e ne rispecchia la descrizione, con 3 *sequencer*,

[(6)]Vedi nota: (5) a pagina 50.

un elemento *case* e molteplici elementi *completion detection*. La latenza attraverso l'albero di controllo sarà facilmente molto larga e questo influisce sulle prestazioni del circuito.

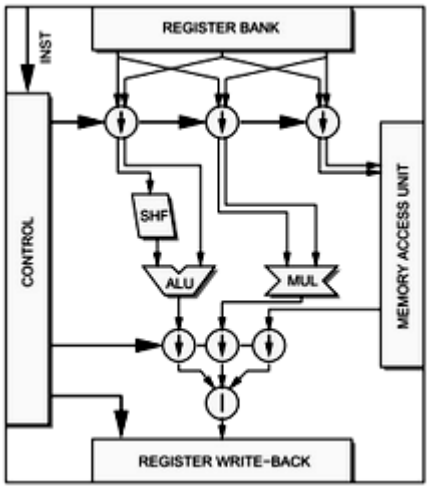


Figura 4.11 – Stadio *Execute* (control-driven).

La sincronizzazione tra dati e controllo sembra essere un prezzo necessario da pagare per garantire la correttezza delle operazioni, ma non è questo il caso poiché nei circuiti asincroni i dati validi identificano se stessi. Non c'è alcun bisogno per un'esplicita sequenzialità delle operazioni: le unità possono aspettare finché i dati non arrivano, processarli e spedire in output i risultati. In figura 4.10(b) è illustrata una descrizione alternativa dello stesso stadio.

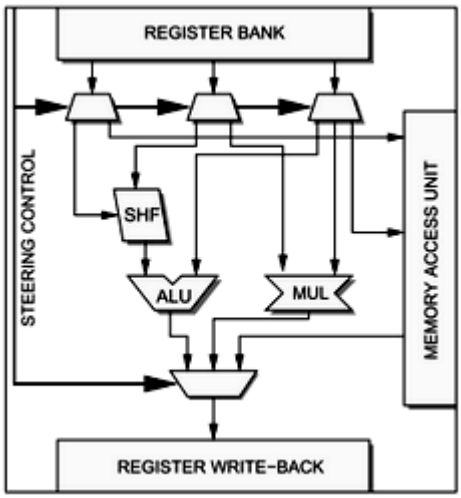


Figura 4.12 – Stadio *Execute* (data-driven).

In questa descrizione, tutte le unità sono attivate in parallelo. Unità *steer*[7] e *multiplex* vengono aggiunte per guidare i dati. La figura 4.12 illustra il rispettivo circuito handshake. Tutte le unità sono pronte per ricevere i dati e cominceranno

[7]Un'unità *steer* si comporta come un'istruzione *case* (o *switch*) di un comune linguaggio di programmazione, cioè un costrutto che realizza in un unico blocco molteplici controlli *if*.

a operare non appena i dati saranno arrivati. In un'istruzione *multiply*, i dati sono inviati all'unità *MUL* e il resto rimane pronto. Nonostante i moduli *steer* e *multiplex* richiedano segnali di controllo, questi possono essere configurati direttamente dal *decoder*, senza coinvolgere nessun tipo di *sequencing* e senza nessun bisogno che i dati siano sincronizzati tra loro. Facilmente il segnali del controllo *steering* sono pronti prima dei dati ma, non ritardando le operazioni, chiaramente migliorano le prestazioni dello stadio.

Speculative operation

L'operatività speculativa è un importante strumento nel moderno design di processori. Utilizzando l'operatività speculativa, si possono ottenere significativi aumenti nelle prestazioni nei casi in cui la maggioranza delle istruzioni viene eseguita (in caso contrario la speculazione avrebbe solo l'effetto di rallentare ulteriormente il processore). L'ISA (Instruction Set Architecture) dell'ARM stabilisce che tutte le istruzioni siano condizionali, cioè che possano essere eseguite o meno in base a codici di condizione. Le statistiche di esecuzione dei programmi mostrano che la maggior parte delle istruzioni viene eseguita, aprendo la possibilità di avviare le istruzioni in modo *speculativo,* scartando i risultati se il test sul codice di condizione fallisce.

L'operatività speculativa non è sempre di immediata implementazione nei circuiti asincroni sintetizzati. In questi sistemi infatti, per la comunicazione dei dati vengono impiegati i canali di handshake, e non dei segnali individuali. E' molto probabile che un sistema finisca in deadlock se un canale non riesce a completare un ciclo di handshake. Per queste ragioni, SPA non ha alcuna operazione speculativa: valuta il codice di condizione di una determinata istruzione in ingresso e avvia l'esecuzione solo se il test ha successo.

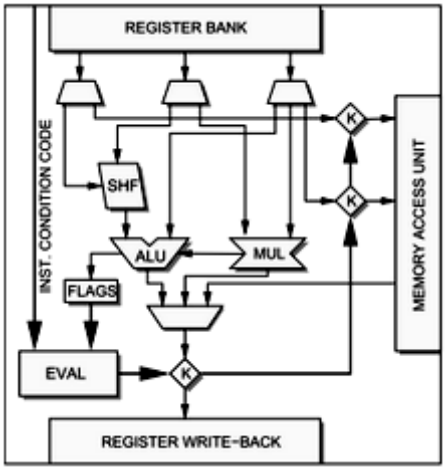


Figura 4.13 – Controllo speculativo delle operazioni.

Un'implementazione orientata alle prestazioni può incorporare le operazioni speculative nell'unità di esecuzione: in questo modo la valutazione dei codici di condizione può essere fatta in parallelo con l'esecuzione delle istruzioni. Se la condizione fallisce, l'istruzione viene scartata in un certo punto di controllo, senza che nessun risultato si propaghi. Le problematiche chiave sono l'ubicazione dei punti di controllo e il bisogno di permettere ai canali di handshake di compiere i loro cicli. La figura 4.13 mostra

come dei moduli *kill* (*K*) vengano usati a questo proposito. Le operazioni di data-processing vengono avviate in modo speculativo e, se il test sulla condizione fallisce, sono scartate prima che i risultati vengano scritti. D'altro canto, le operazioni che conivolgono dati in memoria non vengono avviate in modo speculativo, dato che la penalizzazione sulle prestazioni e l'elevato consumo di potenza dovuti alle istruzioni scartate sarebbero eccessivamente alti. Questa strategia produce un miglioramento nelle prestazioni solo se la percentuale delle istruzioni eseguite è alta, e questo fortunatamente è quasi sempre verificato.

4.3.4. Risultati

In questa sezione illustriamo i risultati delle simulazioni relative a diverse implementazioni (a livello di transistor, pre-layout) del processore SPA che utilizza celle di libreria a $0.18\mu m$. Le simulazioni mostrano che le tecniche descritte in questa sezione forniscono un aumento significativo delle prestazioni.

Processore	**DMIPS**	**Prestazioni rel.**	**Trans.**	**Dimensioni rel.**
Bundled Data				
SPA	10.17	1.00	283.663	1.00
nanoSpa	22.46	2.21	181.749	0.64
nanoSpa (con nuovo HCs)	54.44	5.35	242.724	0.86
Dual Rail				
SPA	6.53	1.00	717.549	1.00
nanoSpa	18.57	2.84	611.578	0.85
nanoSpa (con nuovo HCs)	58.37	8.94	570.920	0.80

Tabella 4.1 – Risultati delle simulazioni.

Le tecniche orientate alle prestazioni sono state applicate nella sintesi del nanoSpa, una nuova specifica dell'architettura SPA originale. NanoSpa è organizzato come una pipeline a tre stadi, *Harvard-style*, ma presenta alcune differenze rispetto alla SPA: (1) lo stadio *decoder* è privo del modulo Thumb e dell'interfaccia del coprocessore, (2) lo stadio *execute* incorpora tutte le unità funzionali presenti in SPA, (3) nanoSPA implementa le modalità operative *user* e *supervisor*, ma è privo delle altre modalità ARM e, (4) nanoSpa non supporta né gli *interrupt* né i *memory abort*. Anche se non facili da calcolare, queste differenze non dovrebbero avere grande impatto sulle prestazioni relative delle due implementazioni.

La tabella 4.1 mostra i risultati dell'esecuzione del programma di benchmark *Dhrystone* per lo SPA originale e due differenti implementazioni del nanoSpa; include i risultati per le versioni bundled-data e dual-rail e le prestazioni dello SPA originale sono riportate per avere un riferimento.

La tabella mostra che l'uso di un efficiente controllo della pipeline, della *true asynchronous operation*, della speculazione e di logica combinatoriale ottimale fornisce risultati eccezionali. Il *core* base del nanoSpa, con le componenti handshake origi-

nali, è 2.21 (bundled-data) e 2.84 (dual-rail) volte più veloce dell'implementazione originale SPA.

La tabella 4.1 mostra anche come l'uso delle specifiche *performance-driven* in combinazione con i nuovi componenti handshake si traduca in un notevole incremento delle prestazioni, raggiungendo un fattore di guadagno del 5.35 per il bundled-data e dell'8.94 per il dual-rail, rispetto allo SPA originale.

Le nuove tecniche e componenti handshake forniscono un guadagno nelle prestazioni che è maggiore per le implementazioni di tipo dual-rail rispetto a quelle di tipo single-rail. Ciò è dovuto in parte al fatto che le implementazioni originali di tipo dual-rail, essendo meno efficienti, erano quelle con maggiore spazio per lo sviluppo e il miglioramento.

In conclusione, la tabella 4.1 mostra anche il numero dei transistor per le differenti implementazioni del processore. Leggendo i dati, è chiaro che le nuove implementazioni sono significativamente più piccole rispetto allo SPA originale. Si può anche notare come i nuovi componenti handshake, nonostante contribuiscano al notevole aumento delle prestazioni, non influiscono più di tanto nella riduzione dell'area. Infatti, la versione dual-rail con le nuove componenti è più piccola. La grande differenza nel numero dei transistor, rispetto all'implementazione originale, indica che c'è ampio spazio per incorporare le nuove funzionalità senza penalizzare le prestazioni.

4.3.5. Conclusioni

Possiamo concludere affermando che la *syntax-directed compilation* dimostra di essere un potente approccio per la sintesi che, combinato con un insieme di componenti handshake efficienti, può generare automaticamente sistemi asincroni per applicazioni reali di qualsiasi complessità.

I risultati di simulazioni estese mostrano che l'introduzione di nuovi componenti handshake usati per implementare controllo parallelo, sequenziale e dell'input, possono raddoppiare le prestazioni di design esistenti senza modificare le descrizioni dei sorgenti. Oltretutto, l'introduzione di nuove tecniche orientate alle prestazioni nell'implementazione del controllo delle pipeline, del comportamento *true asyncronous* e delle operazioni speculative, può triplicare le prestazioni di circuiti esistenti. La combinazione di nuovi componenti e tecniche è stata utilizzata per produrre un processore ARM compatibile a 32 bit che raggiunge circa dieci volte le prestazioni rispetto all'originale.

Con questo capitolo termina la parte introduttiva.

Parte II.

Lo Stato dell’Arte

Nei prossimi capitoli esporremo vari esempi con i quali intendiamo fornire una panoramica sull'attuale stato dell'arte della tecnologia che sta alla base della progettazione e realizzazione di circuiti asincroni.

Come già anticipato nell'introduzione, alcuni concetti di carattere generale che però sono relativi solamente a una data tecnologia o che hanno minore diffusione/importanza, verranno introdotti a corredo di ciascun esempio. Nella rilettura del testo abbiamo aggiunto dei rimandi per i concetti principali, in modo che il lettore sia facilitato quando abbia bisogno di rivedere la teoria che fa da base all'esempio spiegato. La lettura di questa seconda parte può quindi avvenire non necessariamente in ordine.

5. Processore Java per Contactless Smart Card

In questo capitolo introduciamo un esempio che mostra il design di un processore asincrono Java, 16 bit, a basso consumo, per le contactless smart card. La differenza tra le smart card contact e contactless sta nel tipo di interfaccia di collegamento esistente tra il microchip e il mondo esterno. Le prime hanno una contattiera mediante la quale ricevono l'alimentazione e dialogano con l'esterno una volta inserite in un apposito dispositivo terminale detto lettore di smart card. Le seconde hanno un'antenna che reagisce alla presenza di un campo elettromagnetico emesso da uno speciale dispositivo di lettura/scrittura nella banda delle radio-frequenze (con tecnologia RFID), consentendo al microchip di scambiare dati con l'esterno (purché l'antenna si trovi entro una distanza minima dal dispositivo di lettura/scrittura)[(1)].

5.1. Introduzione

La tecnologia delle smart card Java permette a programmi scritti in Java di girare sulle smart card. Da una prospettiva software, la Java Virtual Machine (JVM) definisce un processore stack-based per l'implementazione del bytecode. Il software JVM è quindi in grado di emulare la propria architettura hardware, ed è possibile progettare un microprocessore che possa eseguire le istruzioni in bytecode direttamente in ordine, per ottimizzare le prestazioni [81]. Dato che ci sono pochi design proposti per un processore Java e non tutti sono progettati per essere utilizzati all'interno di una smart card, il processore presentato in questa sede può essere considerato innovativo. Il design è basato su architettura a 16 bit per un processore asincrono che sia in grado di eseguire direttamente il set di istruzioni specificato nella *Java Card 2.2 Virtual Machine Specification* della *Sun Microsystems* [82], con operandi a 16 bit incluse operazioni di lettura/scrittura su registri, operazioni aritmetiche, logiche e di branch. Il resto, che comprende l'invocazione e il ritorno di metodi, operazioni su array e manipolazione di oggetti, è gestito da routine software.

Elenchiamo le funzionalità hadware:

- ALU a 16 bit
- RAM a 16x16-bit (16 moduli da 16 bit)
- Stack a 10 livelli (da 16 bit)
- Address Bus a 16 bit
- Due porte I/O a 16 bit

(1)[http://it.wikipedia.org/wiki/Smart_card]

5.2. Architettura del processore

Le tecniche di design asincrono sono impiegate per ridurre il consumo di potenza. In questo design viene impiegato il protocollo handshake 4-phase bundled-data per via della maggiore semplicità della logica di controllo, rispetto all'implementazione 2-phase. In un sistema per contactless smart card è preferibile ricercare un basso consumo di potenza piuttosto che alte prestazioni, quindi per il design di questo progetto è stato scelto il *normally opaque latch controller* mostrato in figura 5.1.

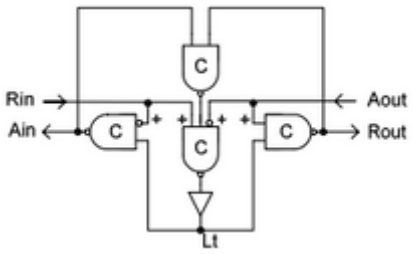


Figura 5.1 – Normally opaque latch controller.

I percorsi per la trasmissione dei bit di controllo e dati tra i vari blocchi funzionali sono mostrati in figura 5.2. Nel processore asincrono Java, al posto dei flip-flop sono utilizzati i latch e per via del design asincrono tutti i blocchi funzionali come lo stack, il blocco delle variabili locali, le porte I/O, il PC latch, l'Operand latch, il blocco additivo (indicato con un "+" in figura), il blocco per la logica e quello per il branch, operano solamente quando necessario.

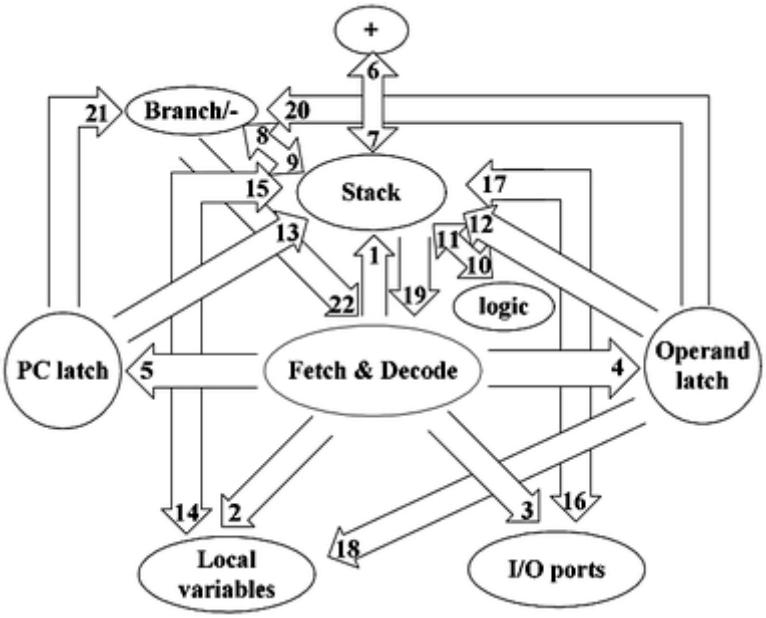


Figura 5.2 – Percorsi per la trasmissione di bit di controllo (1,2,3) e dati.

Alcune ulteriori tecniche che riducono l'attività del circuito sono state impiegate per ridurre il consumo di potenza, come ad esempio la modalità di accesso allo stack. Il processore Java è di tipo stack-based e quindi le sue prestazioni sono limitate dal

frequente accesso allo stack. Utilizzando a questo proposito una tecnica chiamata *result-forwarding*, è possibile incrementare le prestazioni del processore: quando una coppia di istruzioni consiste nella scrittura di un dato in cima allo stack e nella immediata lettura dello stesso, invece di eseguire le normali operazioni di scrittura e lettura associate alle due istruzioni, il dato in questione viene direttamente inoltrato dalla prima alla seconda e ciò permette di risparmiare accessi allo stack.

La RAM 16x16-bit del processore realizza invece il blocco delle variabili locali illustrato in figura 5.2. L'indirizzo della variabile in RAM è fornito dall'Operand latch, sulla destra nella figura. Il PC latch e l'Operand latch sono utilizzati per memorizzare rispettivamente il valore del program counter (relativo all'istruzione corrente) e l'operando a 16 bit letto nell'instruction memory dall'unità di fetch/decode.

Istruzione	**Cammini coinvolti**
sipush const	1, 4, 12
iload index	1, 2, 4, 18, 15
istore index	1, 2, 4, 18, 14
iadd	1, 6, 7
isub	1, 8, 9
ishl	1, 10, 11
ishr	1, 10, 11
iand	1, 10, 11
ior	1, 10, 11
ixor	1, 10, 11
if_scmpeq_w offset	1, 4, 5, 8, 20, 21, 22
if_scmpne_w offset	1, 4, 5, 8, 20, 21, 22
if_scmplt_w offset	1, 4, 5, 8, 20, 21, 22
if_scmpge_w offset	1, 4, 5, 8, 20, 21, 22
goto offset	None
jsr offset	1, 5, 13
ret	1, 19
nop	None
stack2port	1, 3, 16
port2stack	1, 3, 17
sleep	None

Tabella 5.1 – Il sottoinsieme dell'instruction set.

Il sottoinsieme di istruzioni è mostrato in tabella 5.1.

Le istruzioni *stack2port* e *port2stack* non appartengono all'insieme originale definito dalla *Sun*, e sono state aggiunte per permettere al processore di comunicare con le porte I/O; lo stesso vale per l'istruzione *sleep*, che fornisce il supporto per l'omonima modalità.

Come si nota dalla tabella, non c'è trasferimento dati quando vengono eseguite le istruzioni *goto*, *nop* e *sleep* poiché esse sono eseguite solamente nell'unità di fetch/decode.

5.3. Implementazione e risultati sperimentali

Il progetto è stato implementato con tecnologia CMOS a $0.35\mu m$ e le dimensioni del nucleo del processore sono risultate essere di $1.2mm^2$.

Supply (V)	MIPS	Power (mW)	MIPS/W
3.3	18.7	14.1	1326
3	17.2	10.8	1593
2.5	14.2	6.3	2254
2	10.2	2.5	4080

Tabella 5.2 – Risultati sperimentali.

Il bytecode di un programma di test che coinvolge operazioni di lettura/scrittura su registri, operazioni aritmetiche e di branch, è memorizzato nella ROM del processore Java e i risultati mostrati in tabella 5.2 sono riferiti alla sua esecuzione.

Per una contactless smart card, il clock esterno opera a $13.56MHz$ e il consumo deve essere inferiore ai $30mW$.

In conclusione, i risultati sperimentali mostrano che il consumo del processore asincrono Java è di $14.1mW$ (quindi meno della metà del massimo consentito) quando la velocità è 18.7 MIPS (a $3.3V$) e, ovviamente, questo soddisfa i requisiti per il sistema di contactless smart card. Si noti che è possibile ridurre ulteriormente il consumo riducendo la tensione di alimentazione e che, nella modalità sleep, il consumo di potenza è praticamente pari a zero.

6. Il Processore Asincrono 8051

In questo capitolo presentiamo un'implementazione del processore asincrono A8051 che sfrutta una pipeline adattiva.

6.1. Introduzione

I notevoli passi in avanti compiuti dalla tecnologia VLSI hanno portato nuovi orizzonti nel panorama del SoC design. La maggior parte dei SoC richiede più di un processore incorporato per eseguire i calcoli complessi e ciò ha come conseguenza un aumento del consumo di potenza. Nella tecnologia sotto il micron, i delay logici diminuiscono, mentre quelli relativi alle linee aumentano in modo significativo. Oltretutto, il clock di sistema viaggia sulla linea che ha maggiore estensione e ciò produce dei ritardi di fase che degradano le prestazioni. Il design asincrono ha il potenziale di risolvere questi problemi perché, non possedendo il clock, è scevro di tutte le problematiche ne derivano. Tutti i moduli sono di tipo self-timed (si veda sez. 2.4.2 a pagina 25) e comunicano tra di loro utilizzando protocolli handshake. Un altro vantaggio è che ciascun blocco possiede un proprio clock locale e ciò aumenta le possibilità di ridurre considerevolmente il consumo di potenza. Efthymiou [5] ha proposto una pipeline adattiva per saltare gli stadi ridondanti, a regime. La pipeline asincrona adattiva è efficace nel ridurre il consumo perché non spreca potenza eseguendo gli stadi "bolla" [55].

Per ridurre il consumo, il design asincrono si propone quindi come un'attraente alternativa a quello sincrono. La maggior parte dei processori asincroni è basata su un'architettura RISC (Reduced Instruction Set Computer) poiché essa può facilmente adattarsi alla pipeline ma, nonostante ciò, è l'architettura CISC (Complex Instruction Set Computer) che, grazie al suo complesso modello di esecuzione, ha le potenzialità per trarre maggior vantaggio dal design asincrono. Il processore Intel 8051 (i8051) è il più popolare processore *embedded* per sistemi industriali per via delle ridotte dimensioni e del basso costo. Ci sono molte controparti asincrone per l'i8051, l'ultima della quali, il Lutonium [76], vanta prestazioni da $200MIPS$.

In questo capitolo presentiamo un'implementazione asincrona del processore 8051, chiamata A8051 [44]. Si presenta in due versioni: A8051v1 e A8051v2. La prima impiega un'architettura con pipeline fissa a cinque stadi mentre la seconda un'architettura con micropipeline adattiva capace di *stage-skipping* e *stage-combining* (cioè può saltare uno stadio o combinare più stadi). Entrambe le versioni sono state implementate utilizzando tecnologia Hynix CMOS a $0.35\mu m$. L'A8051 inoltre possiede anche alcune funzionalità avanzate come il *multilooping*, il *single-threading* e il *branch-prediction*. Nonostante siano schemi popolari per il design sincrono, è possibile realizzarli in modo più semplice con quello asincrono.

6.2. Caratteristiche dell'A8051

Nonostante l'architettura dell'instruction set dell'A8051 (ISA: Instruction Set Architecture) sia pienamente compatibile con quella dell'i8051, i due processori presentano significative differenze per quanto riguarda gli schemi di esecuzione e l'architettura di sistema.

Innanzitutto, l'A8051 possiede un modello di esecuzione delle istruzioni più semplice di quello dell'i8051. Nell'i8051 alcune istruzioni sono eseguite in multi-ciclo, cioè richiedono più di un ciclo per essere completate. Nonostante queste istruzioni aumentino l'efficienza del codice, il modello di esecuzione, di fatto, è complesso. In particolare, le istruzioni che abbisognano di 2 o 4 cicli macchina generano attività ridondanti nella regolazione della pipeline. L'A8051v2 possiede lo stesso instruction set della versione 1.

Lo schema di esecuzione elimina le operazioni di stadio superflue dal ciclo di un'istruzione, e rimpiazza la ripetizione dell'intero ciclo macchina con la ripetizione locale degli stadi di *operand fetch* (OF) ed *execution* (EX). Strutturalmente è diviso in sette gruppi, derivati dalle caratteristiche del modello di esecuzione. In aggiunta, l'A8051 sfrutta una pipeline adattiva che include lo stage-skipping e lo stage-combining. Il primo viene utilizzato per saltare gli stadi che sono inattivi (idle) mentre il secondo unifica stadi vicini quando il secondo è inattivo.

L'A8051 esegue il fetch delle istruzioni dalla ROM, 4 byte alla volta, poiché la lunghezza delle istruzioni varia da uno a tre byte. Nonostante l'impiego di latch aggiuntivi, in questo modo ogni istruzione entra in pipeline in tempo senza che si verifichino data hazards causati dal collo di bottiglia della memoria. L'instruction register (IR) memorizza i byte extra che vengono recuperati con un'istruzione successiva. L'IR è utilizzato quindi come una cache di istruzioni e ciò aiuta a evitare stalli nella pipeline, causati dalla lunghezza variabile delle istruzioni. Il branch predictor è molto semplice perché i salti incondizionati garantiscono un basso costo e favoriscono il design asincrono. Esso è costituito da un *Predecoder* e un *Target Address Calculator* (TAC). Il predecoder identifica quando l'istruzione di cui si è appena eseguito il fetch produce un salto incondizionato, mentre il TAC calcola l'indirizzo di destinazione e lo trasferisce nel program counter. In questo modo non vengono eseguite operazioni speculative non necessarie per i salti condizionati, risparmiando potenza nello stadio di fetch.

Quando viene eseguito il fetch di un'istruzione di salto incondizionato, gli stadi della pipeline vengono svuotati (*flushed*) dopo che è stato calcolato l'indirizzo di destinazione nello stadio di esecuzione.

L'architettura dell'A8051 è illustrata in figura 6.1.

6.3. Riduzione del consumo di potenza

In questa sezione vediamo perché l'uso di metodi come lo stage-skipping/combining, il multi-looping e il single-threading locale che sono impiegati dall'A8051 permettono di ridurre il consumo di potenza.

Il processore è di tipo DI (vedi sezione 2.4.2 a pagina 25), necessita di un segnale di completamento, ottenuto dalla codifica dati, che indichi che le operazioni di ciascun blocco di logica combinatoriale sono state completate. La sincronizzazione tra

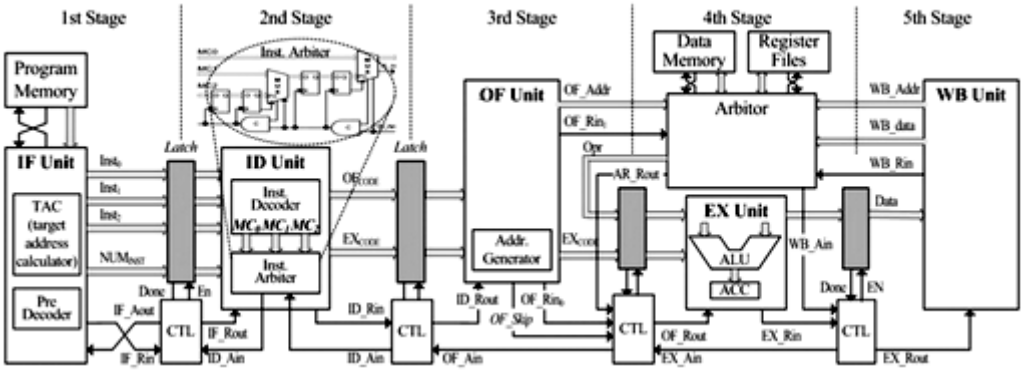


Figura 6.1 – Architettura dell'A8051.

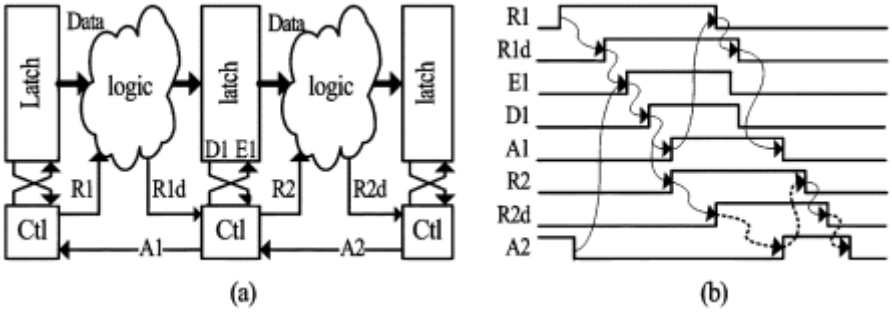


Figura 6.2 – 4-phase handshaking signaling. (a) modello e (b) diagramma delle transizioni.

gli stadi della pipeline richiede un 4-phase handshaking, come illustrato in figura 6.2. Il trasmettitore porta a 1 il segnale R1 (request 1) indicando che intende trasferire dati; il ricevitore li recupera e setta acknowledge a 1 (linea A1) per avvertire il trasmettitore dell'avvenuta ricezione. Il trasmettitore quindi abbassa a 0 R1.

I segnali di controllo transitano in questa sequenza: $R1 \uparrow\rightarrow R1d \uparrow\rightarrow E1 \uparrow\rightarrow D1 \uparrow\rightarrow A1 \uparrow\rightarrow R1 \downarrow$. Le frecce $\uparrow$ e $\downarrow$ indicano il salire e scendere dei segnali (a 1 e 0, rispettivamente). In aggiunta, il lato ricevitore vede anche: $D1 \uparrow\rightarrow R2 \uparrow\rightarrow R2d \uparrow\rightarrow E2 \uparrow\rightarrow D2 \uparrow\rightarrow A2 \uparrow\rightarrow R2 \downarrow$. Alla fine, quando il ricevitore ha terminato correttamente l'acquisizione dei dati, il controller reinizializza il blocco trasmettitore, riportando i segnali al loro livello logico originario: $R1 \downarrow\rightarrow R1d \downarrow\rightarrow A2 \downarrow$.

6.3.1. Stage-skipping e stage-combining

Abbiamo detto che l'i8051 include un certo numero di stati ridondanti, che non eseguono alcuna operazione. Ciascuna istruzione è eseguita in multipli di cicli macchina consistenti di sei stati, ma non tutte le istruzioni richiedono tale quantità. Ad esempio, l'istruzione "INC C" è di un solo byte e quindi il secondo fetch per gli altri due byte non è richiesto. Lo stadio WB quindi non è necessario, perché questa istruzione terminerà nello stadio di esecuzione. Questa istruzione impiega quattro stadi per essere completata e quindi possiede due stadi ridondanti. Un sistema sincrono sprecherebbe potenza cercando di eseguire questi due stati per mantenere la regolarità dei cicli macchina.

La struttura della pipeline che vediamo ora invece utilizza lo stage-skipping e lo stage-combining per ovviare a questo spreco. Utilizzando questi schemi, possiamo unire e saltare alcuni stadi della pipeline in base alle istruzioni inserite, evitando di sprecare potenza. Sono aggiunti degli input extra (EL_N e EC_N) al latch controller e allo stadio della pipeline, rispettivamente, come mostrato in figura 6.3.

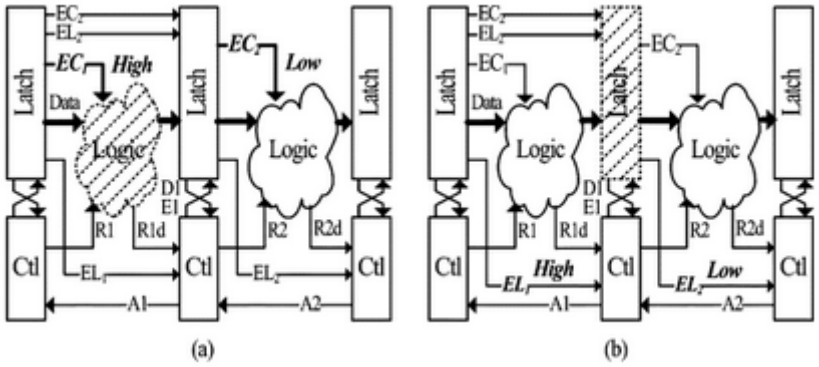


Figura 6.3 – Esempio di stage-skipping/combining: (a) stage-skipping e (b) stage-combining.

Questi segnali sono impacchettati con i dati e agiscono sugli stadi della pipeline quando la richiesta giunge allo stadio successivo. Il segnale EL_N determina se l'ennesimo latch debba diventare trasparente o meno, mentre invece il segnale EC_N determina se l'operazione sull'ennesimo stadio della pipeline debba essere saltata. Il processore salta così le "bolle" con il metodo stage-skipping. Lo stadio ID (*Instruction Decoding*) propaga la micro-istruzione e il segnale EC_N a quello successivo. Viene settato EC_N perché lo stadio ID è in grado di riconoscere gli stadi "bolla" dopo la decodifica delle istruzioni. Lo stage-skipping salta l'esecuzione di una "bolla" semplicemente non eseguendola. All'inizio e alla fine di ciascuno stadio ci sono rispettivamente un multiplexer e un de-multiplexer. Il segnale EC_N può determinare se l'ennesimo stadio venga eseguito o meno controllando il multiplexer e il de-multiplexer. Lo stadio *upstream* della pipeline viene considerato invisibile quando EC_N è alto. Ad esempio, EC_1 determina se il primo stadio verrà eseguito o no, come mostrato in figura 6.3(a). La logica combinatoriale del primo stadio della pipeline viene eseguita quando questo segnale è a 0. Al contrario, se il segnale è a 1, l'input si limita a filtrare al secondo stadio senza che il primo venga eseguito.

La tecnica di stage-combining invece unisce due stadi consecutivi quando il secondo dei due è vuoto. Lo stadio ID sa quando si può effettuare l'unione di due stadi dopo che le istruzioni sono state decodificate. Ad esempio, il latch tra il primo e il secondo stadio diventa trasparente quando EL_1 è a 1, come indicato in figura 6.3(b). I latch della pipeline diventano trasparenti quando lo stadio di *downstream* ha terminato la propria operazione di allocazione ed è pronto per lo stadio successivo. Comunque, se EL_1 è a 0, la pipeline opera normalmente. Ciascun segnale EL_N viene emesso dallo stadio ID e quindi passato in avanti da ciascuno stadio a quello successivo, finché il segnale non raggiunge il latch controller di destinazione, come mostrato in figura 6.3.

La figura 6.4(a) e (b) mostra rispettivamente i diagrammi temporali delle operazioni di stage-skipping e stage-combining.

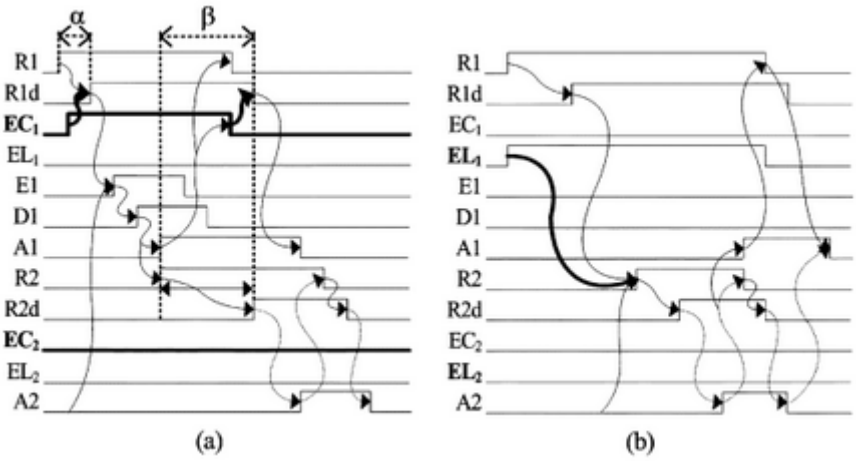


Figura 6.4 – Protocolli per lo stage-skipping/combining di figura 6.3: (a) stage-skipping e (b) stage-combining.

6.3.2. Multi-looping negli stadi OF ed EX

Le istruzioni dell'Intel 8051 tipicamente richiedono due o quattro cicli macchina a seconda (1) del numero di operandi di cui bisogna eseguire il fetch e (2) della complessità dell'operazione. In questo caso, un ciclo macchina consiste di dodici cicli di clock. Queste istruzioni multi-ciclo generano stadi "bolla" per sincronizzarsi con il flusso di esecuzione della pipeline. La maggior parte delle istruzioni multi-ciclo richiedono molteplici operazioni di stadio OF ed EX (ricordiamo: Operand Fetch ed Execution). La ripetizione dell'intero ciclo macchina può essere rimpiazzata eseguendo un multi-looping locale in questi due stadi. Ad esempio, "INC DPTR" è un'istruzione da due cicli macchina. Il byte meno significativo è memorizzato nei primi 12 cicli di clock, mentre quello più significativo nei dodici successivi. L'A8051 può sostituire il secondo ciclo macchina iterando gli stadi OF ed EX. Il multi-looping è realizzato aggiungendo un semplice circuito di arbitraggio che determina l'ordine tra le micro-istruzioni che vanno ripetute nello stadio ID e quelle ancora da decodificare. La maggior parte delle istruzioni viene eseguita in un ciclo lineare di pipeline e solo poche hanno bisogno del multi-looping locale.

L'A8051 divide l'ISA in sette gruppi in base al numero di cicli di macchina richiesti per eseguire un'istruzione. Basate sul raggruppamento delle istruzioni, le strutture delle pipeline riflettono le caratteristiche di ciascun gruppo, fornendo una specifica configurazione di ogni stadio. Le istruzioni che richiedono il multi-looping sono circa il 17% dell'ISA. Queste istruzioni non ripetono l'intero ciclo ma eseguono invece solo due stadi di pipeline. Utilizzando questo tipo di schema di controllo, il numero di stati necessari nei cicli macchina viene ridotto del 44% rispetto a quello dell'Intel 8051. Questo non solo permette di risparmiare sui consumi, ma incrementa anche le prestazioni.

6.3.3. Single-Threading locale nello stadio EX

In generale, un sistema sincrono utilizza una tecnica di clock-gating così che il clock raggiunga solamente i blocchi che stanno eseguendo delle operazioni, in questo modo

riducendo il consumo di potenza. L'A8051 utilizza invece un tipo controllo che dipende dai dati e pilota i vari blocchi utilizzando un protocollo handshake. In particolare, utilizza un controllo a richiesta (on-demand) per inserire gli input nella logica interna di ogni stadio.

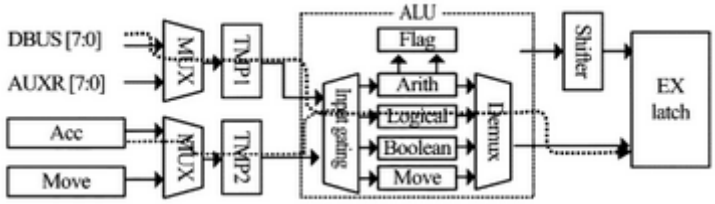


Figura 6.5 – Single-threading locale nello stadio EX.

Un esempio di questo approccio si verifica, ad esempio, nello stadio EX. Dato che le istruzioni realizzano funzioni diverse, ne consegue che utilizzano diversi datapath. Per eseguire istruzioni aritmetiche è richiesto il datapath che comprende la logica computazionale aritmetica e dell'ALU, l'Accumulatore, e non include altra logica nello stadio EX. Quindi è completamente inutile fornire un input a quelle porzioni di logica che non sono interessate dall'operazione in corso e, grazie a queste considerazioni, è possibile ridurre il consumo. Di conseguenza ogni istruzione consuma una quantità di energia che è proporzionale al numero di blocchi interessati dal cammino dei dati e, in questo senso, il comportamento del processore asincrono è simile alla tecnica di clock-gating menzionata prima.

La figura 6.5 mostra il concetto di single-threading, iniettando l'input nelle unità funzionali nello stadio EX. L'ISA dell'A8051 è diviso in cinque classi: aritmetica, logica, trasferimento dati, booleana e instruzioni di salto (*branch*). Ciascuna classe ha un datapath differente nello stadio EX. Ad esempio, il datapath per eseguire l'istruzione "ANL A, Rn" è indicato con una linea tratteggiata in figura 6.5. Questa istruzione richiede due operandi: l'operando di registro viene ricavato con un fetch nello stadio OF, mentre l'altro risiede nell'accumulatore. Il MUX che dirige gli input sceglie il datapath adeguato e lo attiva, risparmiando così potenza grazie al fatto che gli altri datapath rimangono spenti.

6.4. Analisi di potenza

L'A8051 include il datapath completo dei nuclei i8051, una memoria interna e una logica di controllo di tipo pipeline adattiva. Dopo essere stato simulato con il NanoSim, è stato mappato su tecnologia CMOS a $0.35\mu m$ con una tensione nominale di $3.3V$. Per comprendere appieno l'efficienza di questo tipo di pipeline, ne sono state analizzate due versioni differenti. L'A8051v2 ha una complessità di circa 110κ transistor e occupa $16mm^2$ di area in silicio, cioè meno dell'1% in più rispetto alla versione 1. I risultati della simulazione sono stati ottenuti con il benchmark con Dhrystone v2.1 [94].

I risultati del confronto tra l'A8051 e altre controparti sono esposti in tabella 6.1 [55]. L'H8051, il CIP51 e il DS89C420 sono nuclei sincroni i8051. L'H8051 ha un'architettura priva di pipeline. Il CIP51 e il DS89C420 invece sono le ultimissime implementazioni dell'i8051 che sfruttano un'architettura con pipeline. Il Lutonium e il Nanyang_A8051 sono le rispettive controparti asincrone. L'A8051v1 e l'A8051v2 sono ri-

Processore	Tecn.	MIPS	MIPS/W	Cons. med. (mW)	$\mathbf{Et^2}$ $(\mathbf{Js^2})$
H8051	$0.35\mu m\,3.3V$	4	89.5	44.7	6.98×10^{-22}
CIP51	$0.35\mu m\,3.3V$	47	—	—	—
DS89C420	$0.35\mu m\,1.1V$	11	600	18.52	14.0×10^{-22}
Asynch 80C51	$0.5\mu m\,5V$	4	444	9	1.41×10^{-22}
Lutonium	$0.18\mu m\,1.8V$	200	1800	100.0	1.25×10^{-26}
Nanyang_A8051	$0.35\mu m\,1.1V$	0.6	8000	0.07	4.04×10^{-22}
A8051v1	$0.35\mu m\,3.3V$	75.5	1797	46.8	10.39×10^{-26}
A8051v2	$0.35\mu m\,3.3V$	84.2	2316	36.3	6.06×10^{-26}

Tabella 6.1 – Confronto delle prestazioni con altre versioni.

sultati avere una prestazione media di 75.5 e $84.2MIPS$, rispettivamente. L'A8051v2 è circa 1.8 volte più veloce del CIP51 con un clock a $100MHz$ e circa 7.7 volte più veloce del DS89C420. Il Lutonium è circa 2.4 volte più veloce dell'A8051v2, ma è stato sintetizzato utilizzando tecnologia CMOS a $0.18\mu m$.

L'H8051, l'A8051v1 e l'A8051v2 sono invece stati sintetizzati utilizzando la stessa tecnologia e consumano rispettivamente 44.7, 46.8 e $36.3mW$. Ciò significa che l'A8051v2 consuma circa il 22.4% in meno dell'A8051v1 e il 18.8% in meno dell'H8051. I risultati notevoli, in termini di risparmio energetico, raggiunti dall'A8051v2 indicano che la pipeline utilizzata in questo processore è più performante dell'approccio pipeline sincrono convenzionale e ciò è dovuto, come abbiamo visto, alla riduzione delle operazioni degli stadi "bolla". La maggioranza dei processori asincroni mostra risultati migliori sia nel consumo di potenza che nelle prestazioni, se paragonati alle controparti sincrone. Per quanto riguarda invece la misura dell'energia per istruzione, l'A8051v2 si è dimostrato essere circa 3.9 volte più efficiente del processore sincrono 8051 dotato di pipeline, il DS89C420. Per il consumo, l'A8051v2 ha raggiunto i $36.6mW$, che significa una media di $430pJ$ per istruzione, cioè circa 19 volte in più del Nanyang_A8051, rispetto al quale però è molto più veloce.

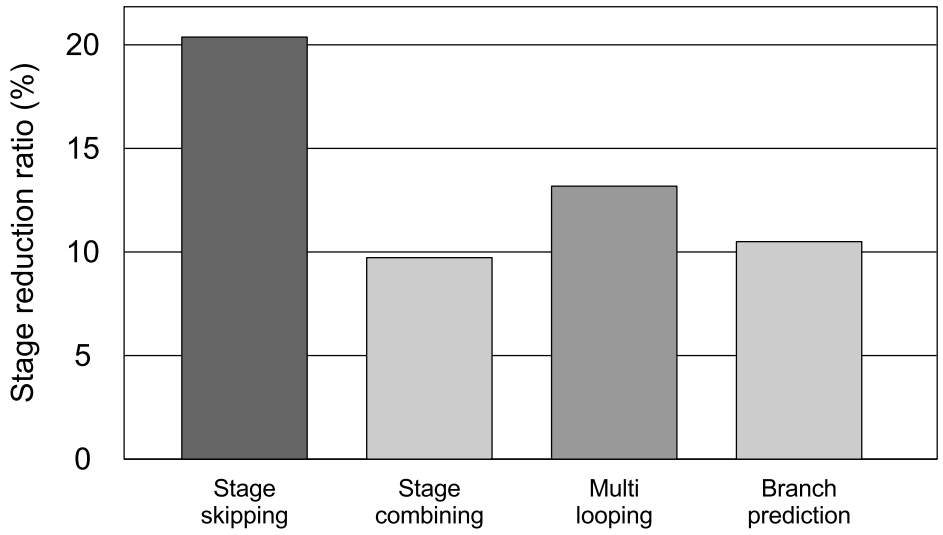


Figura 6.6 – Rapporto di riduzione stadi secondo il metodo di controllo.

La figura 6.6 mostra il rapporto di riduzione del numero di stadi della pipeline ottenuti con le tecniche di controllo stage-skipping, stage-combining, multi-looping e branch prediction. Il paragone è stato effettuato tra gli ISA dell'A8051 e dell'Intel 8051. Queste tecniche riducono gli stadi rispettivamente del 20.3%, 9.9%, 13.2% e 10.9% e la loro azione combinata produce una riduzione complessiva del 54.3% quindi più della metà.

Pipeline	**Stage skipping**	**Stage combining**	**Multi looping**	**Branch Prediction**
Frequenza di utilizzo (nei test)	42%	11%	15%	23%
Risparmio di potenza (rapporto)	3.8mW (8.1%)	1.7mW (3.6%)	2.4mW (5.1%)	2.6mW (5.6%)

Tabella 6.2 – Risultati sul risparmio di potenza del programma di benchmark.

La tabella 6.2 mostra la riduzione dei consumi per ciascuna tecnica, riduzione che nel complesso si attesta sul 22.4%. Per provare l'efficienza del modello di esecuzione semplificato, alcune istruzioni dal programma di benchmark Dhrystone sono state analizzate individualmente.

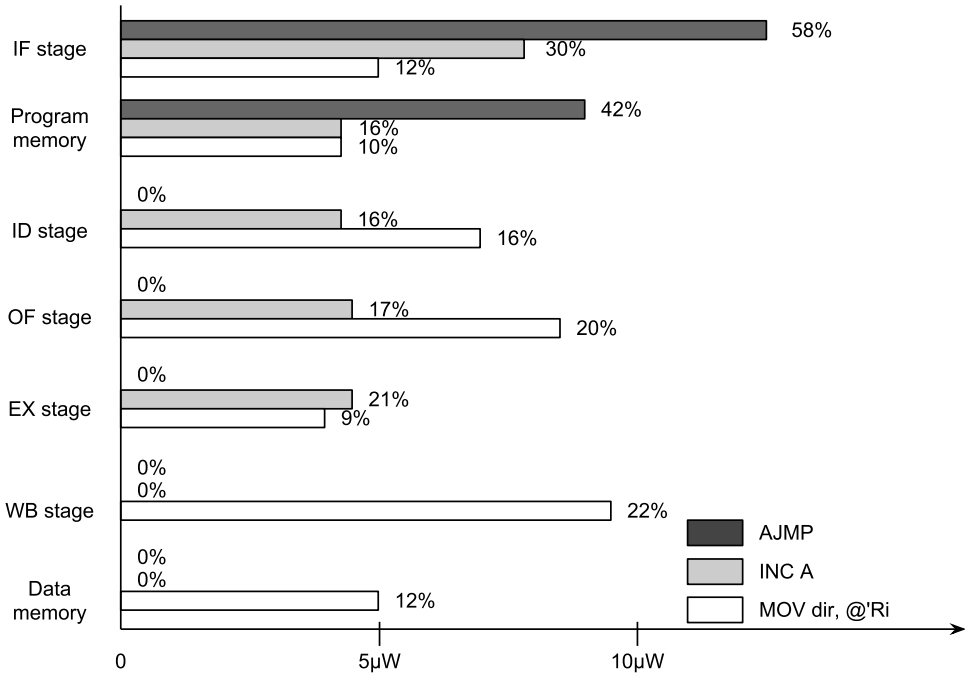


Figura 6.7 – Bilancio energetico nello stadio pipeline (per istruzione).

La figura 6.7 mostra il bilancio energetico per tre tipi di istruzioni: "AJMP", "INC A" e "MOV dir, @Ri". L'istruzione "AJMP" è un salto incondizionato che opera solo nello stadio IF. La simulazione mostra che a parte lo stadio IF e la memoria di programma, nessun altro stadio consuma energia. Il secondo esempio è "INC A", che incrementa il valore dell'accumulatore. L'istruzione consuma potenza negli stadi IF, ID, OF ed EX. L'ultimo esempio invece è "MOV dir, @Ri", che sposta *indirect RAM* su *direct byte*.

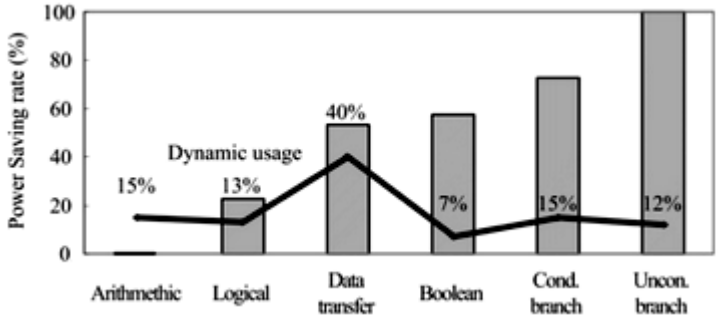


Figura 6.8 – Tasso di risparmio sul consumo per istruzione nello stadio EX, normalizzato con istruzioni aritmetiche.

Dato che tutti gli stadi vengono incontrati per eseguire questa istruzione, il consumo di potenza è visibile in ogni stadio della pipeline.

La figura 6.8 mostra la relazione tra il tasso di risparmio energetico normalizzato e l'uso dell'istruzione nello stadio EX: dato che le istruzioni aritmetiche sono le più dispendiose dal punto di vista energetico, queste statistiche suggeriscono che le istruzioni più importanti ai fini dell'ottimizzazione del consumo siano quelle che concernono il movimento dei dati tra i registri e la memoria (sono circa il 40% di tutte le istruzioni eseguite).

6.5. Conclusioni

Possiamo concludere affermando che per l'architettura A8051, la pipeline adattiva riduce drasticamente il numero degli stati sui quali vengono eseguiti i test. Questo schema di controllo semplificato è basato principalmente su tre concetti: raggruppamento delle istruzioni, un instruction buffer a lunghezza variabile e un branch prediction di tipo statico. Il raggruppamento delle istruzioni crea sette gruppi disgiunti dell'IS (RISC-like) e ciò riduce significativamente il tempo medio di esecuzione. Il branch prediction statico invece riduce la probabilità dello scorrimento dati nella pipeline (*pipeline flush*) e la penalizzazione dovuta ai salti. Queste tecniche, unite ad una pipeline configurata per il single-threading locale nello stadio EX portano la prestazione a $84.2MIPS$, che significa operare circa 1.8 volte più veloci della controparte sincrona, con un consumo ridotto del 22.4% (di cui un 5% dovuto al single-threading).

7. Processore Low-Power, Information-Redundant con C.E.D.

In questo capitolo introduciamo un progetto per un processore asincrono a basso consumo di tipo *information redundant*. A corredo del tema principale, illustreremo anche in cosa consista il *Concurrent Error Detection* (CED[1]) e più avanti nel capitolo avremo modo di vedere un esempio di implementazione su FPGA del circuito proposto.

7.1. Introduzione

Uno dei problemi dell'approccio asincrono è l'alta probabilità di errori nelle transizioni di stato. La responsabilità è individuata nella mancanza del segnale globale di clock, e questo problema affligge sia i circuiti puramente asincroni che le versioni ibride GALS. Oltretutto, poiché le dimensioni dei componenti e la tensione di alimentazione sono sempre più ridotte, i circuiti, in generale, sono sempre più sensibili al rumore e ai cosiddetti *single event upsets* (SEU), che possono generare dei *transient faults*[2].

La natura dei *transient faults* (la breve durata e il verificarsi in tempi casuali), li rende invisibili a test standard che impiegano BIST[3] o strategie scanpath [71], poiché queste sono applicate solo in modo periodico. Per riuscire a individuare fault transitori o intermittenti, che possono manifestarsi sotto forma di *soft errors* (errori "leggeri", cioè non gravi) o sotto forma di *silent data corruption* (cioè come corruzione dei dati "silenziosa"), è necessario implementare un qualche tipo di *Concurrent Error Detection* (CED).

Questo fa sì che i blocchi funzionali vengano testati in parallelo alle loro normali operazioni e ciò produce un sistema consapevole dei propri errori (*error aware system*). In strutture in cui ci sono lunghe pipeline di blocchi funzionali (come le DSP[4] ad esempio), le opportunità che i dati possano corrompersi sono proporzionali alla lunghezza della pipeline, quindi tanti più blocchi funzionali avremo in sequenza, quanto più alta sarà la probabilità che un dato possa corrompersi. Senza un controllo di errore all'interno delle pipeline, può verificarsi la *silent data corruption*, che porta ovviamente a risultati scorretti.

I metodi per eseguire il controllo degli errori sono molti e si differenziano per complessità, costo di realizzazione ed efficienza (cioè per la percentuale di errori che individuano). Sono stati impiegati metodi ridondanti (nel tempo, nell'informazione o direttamente hardware) e si è visto che ciascuno di questi presenta un ben definito insieme di vantaggi e svantaggi. Implementare sistemi di *time redundancy*, ad esempio, rallenta di molto il circuito poiché le operazioni impiegano più tempo per essere

[1] Il CED è un sistema di controllo degli errori che viene utilizzato anche nei circuiti sincroni.
[2] Transient Fault: Errore transitorio.
[3] BIST: Built In Self Test
[4] DSP: Digital Signal Processing.

completate (i dati vengono ricalcolati). Il costo di un sistema hardware ridondante spesso è maggiore di quello di un sistema che impiega metodi *time/information redundancy* e questo ne ha abbassato la popolarità. Rimane comunque utilizzato quando sia necessaria un'estrema affidabilità e il fattore consumo di potenza non sia cruciale.

L'*information redundancy* invece è spesso favorito rispetto al *time redundancy* per via della maggiore velocità che si ottiene nelle operazioni. All'interno dei vari metodi di *information redundancy* troviamo anche gli *Error Correcting Codes* (ECC), che richiedono area e tempi maggiori per l'individuazione e la correzione degli errori. Una buona via di mezzo è la metodologia *detection and recovery*, sia per una questione di area che per una questione di tempi.

Esistono svariati codici per l'*information redundancy* e oltretutto, molte case produttrici di chip implementano design proprietari o utilizzano design *Intellectual Property* (IP) esistenti che già contengono routine per il controllo degli errori. Questo porta dei risultati di certo poco brillanti, per via dell'impiego di diversi codici all'interno di un singolo chip. Il problema di questo approccio sta nell'incremento dell'area, dovuto alla necessità di implementare blocchi per la conversione dei vari codici, e proprio questo incremento è responsabile di un ulteriore aumento delle probabilità di errore nel circuito.

Esempi di codici *information redundant* usati di frequente sono i *Residue Codes* [38] e il *Parity Checking* [91], che possono però individuare solo singoli errori. Il *Berger Code* [51] fornisce una copertura del 100% per gli errori su molteplici bit, ma può richiedere una notevole quantità di area e ciò è sconveniente ad esempio quando sia necessario individuare solo alcune delle categorie – e non tutte – di errori su molteplici bit. Il *Dong Code* [29] invece permette di creare una copertura *ad hoc* per gli errori su molteplici bit per una data applicazione, minimizzando l'area necessaria per la sua implementazione. Con l'eccezione del *Dong Code*, tutti i metodi sopracitati possiedono un set di equazioni per l'*arithmetical & logical prediction* che includono un moltiplicatore. Il *Dong Code* possiede equazioni per svariate funzionalità dell'ALU, come *add/subtract*, funzioni logiche e *rotate/shift* [6], ma manca di protezione per la moltiplicazione.

In questo capitolo vedremo un'estensione del *Dong Code* (per la moltiplicazione) e il conseguente design per un processore RISC a 32 bit che può essere impiegato in applicazioni DSP.

7.2. Berger e Dong Code

Il Dong Code altro non è che il Berger Code modificato, infatti entrambi possiedono diverse caratteristiche comuni, come ad esempio il fatto che sono entrambi dei codici *separabili*. Ciò significa che i bit relativi ai dati e quelli relativi al controllo sono ben identificabili e non necessitano di una fase di decifratura per poter essere processati. Anche se i codici separabili aumentano il numero di bit necessari per rappresentare un dato output e quindi richiedono un più ampio spazio per la memorizzazione dei dati, presentano comunque il vantaggio che una certa quantità di area può essere risparmiata poiché non è necessario implementare la circuiteria che si prenderebbe cura della codifica/decodifica.

Il Berger Code, per definizione, richiede $n = \lceil \log_2(m+1) \rceil$ bit per il codice, dove *m* è il numero dei bit di una parola (*data word*). Ad esempio, per parole di 16 bit, il Berger Code richiede 5 bit aggiuntivi per il *check symbol* (il check symbol è il conteggio del numero di zeri contenuti in una parola).

Esempio:
X=1100110011111111,
quindi il numero di zeri in X, $X_C = 4$
Check symbol del Berger Code: $S_C = b00100$

Utilizzando il Dong Code si può ottenere un maggiore controllo sul livello di errore e quindi sulla copertura dei *fault*, poiché il codice si può adattare all'applicazione in questione. Il Dong Code è formato da due sezioni, C1 e C2. C1 è il conteggio del numero degli zeri mod(2^k)[5] all'interno della parola di codice, dove *k* è il numero di *check bits* impiegati per il *check symbol*. C2 invece è il numero di zeri in C1, e fornisce un controllo sul controllo stesso. Il codice completo Sc_{Dong} è quindi $C1 \mid C2$[6] e richiede $n = k + \lceil \log_2 k \rceil$ bit, dove *k* è un numero intero positivo. Ciò significa che il numero di bit richiesti per il *check symbol* non è più funzione della lunghezza della parola.

Esempio:
X=1100000011100001,
Numero di zeri in X, $X_C = 10$
Dong Code per k=3*:* $C1 = 10 \, mod \, 8 = b010$*,*
$C2 = 2 = b10$, $Sc_{Dong} = C1 \mid C2 = b01010$

Incrementando il valore di *k*, la copertura sull'errore può aumentare (in parallelo aumenta però anche l'area per implementare il controllo). Per fare un esempio, per una parola di informazione a 32 bit, con k=3 per la prima parte del Dong Code (C1) e 2 bit per la seconda parte (C2), viene individuato il 98.54% degli errori unidirezionali.

Tipo di errore sui bit di informazione	**Tipo di errore sui bit di controllo**	**Numero di errori identificati**
Unidirezionale $1 \rightarrow 0$ o $0 \rightarrow 1$	Error free	Errori di peso $\neq (m+1)$ o multipli
Unidirezionale $1 \rightarrow 0$ o $0 \rightarrow 1$	Unidirezionale $1 \rightarrow 0$ o $0 \rightarrow 1$	Tutti gli errori
Bidirezionale $1 \rightarrow 0$ e $0 \rightarrow 1$	Unidirezionale $1 \rightarrow 0$ o $0 \rightarrow 1$	Tutti gli errori

Tabella 7.1 – Capacità di individuazione errori del Dong Code.

In tabella 7.1 sono illustrate le prestazioni del Dong Code per diverse configurazioni e possiamo vedere che questo tipo di codice è particolarmente adatto per sistemi con tolleranza media all'errore, come ad esempio processori DSP di *data logging*.

[5] L'operazione $x \, mod \, y$ fornisce il resto della divisione intera di x per y. Es: $13 \, mod \, 5 = 3$. In letteratura si possono trovare svariate notazioni, la più usata è questa: $13 \equiv 3 \, mod \, 5$. Per approfondimenti: [27] (sez. 3.4 e 4.6).

[6] L'operatore "|" in questo contesto indica una concatenazione.

Inizialmente si può pensare che il Dong Code non offra particolari vantaggi poiché richiede 3 bit per rappresentare C1 e 2 per C2, per un totale di 5 bit per una parola di informazione con una massimo di 16 zeri. C'è da considerare però che il vantaggio di questo approccio non sta tanto nel numero di bit, quanto nella rapidità del calcolo dei codici di controllo: l'uso della funzione *mod* infatti semplifica la complessità del calcolo e quindi velocizza le operazioni.

7.3. Check Symbol Prediction

Il principio operazionale sul quale si basa uno schema CED che incorpora il CSP (*Check Symbol Prediction*) consiste nel fare un confronto del simbolo di controllo generato da un'operazione con un simbolo previsto, già calcolato, in modo da poter identificare gli errori. Le funzioni del CSP e CSG (*Check symbol Generator*) sono calcolate in parallelo. Le funzioni CSP sono basate sugli input dell'operazione in corso, mentre le CSG sugli output. Ad esempio, l'operazione di moltiplicazione utilizza il blocco CSP per calcolare il valore che dovrebbe essere osservato nell'output del CSG, che invece processa i dati in output dall'ALU che rappresentano il risultato della moltiplicazione. In questo modo il CSP, lavorando sugli input e il CSG, lavorando sugli output, possono fare un confronto tra i simboli di controllo e verificare che l'operazione sia andata a buon fine.

$$Sc = nX_c + nY_c - X_cY_c - C_c + n \tag{7.1}$$

L'equazione 7.1 mostra il calcolo del simbolo di controllo per il CSP del Berger Code, applicato alla moltiplicazione [52] di due valori binari X e Y dove:

Sc: simbolo di controllo del Berger Code
Xc: numero di zeri nell'operando X
Yc: numero di zeri nell'operando Y
Cc: numero di zeri all'interno dei registri di carry
n: numero di bit degli operandi

$$C1 = \overline{((X_cY_c + C_c) mod 2^k)} + 1 \tag{7.2}$$

Assumendo $nX_c, nY_c > 2^k$, che è dato da $n \geqslant 2^k$, l'equazione 7.1 può essere modificata per fornire l'equivalente simbolo C1 del Dong Code (eq. 7.2).

Lo schema CED per la moltiplicazione, come mostrato nell'equazione 7.2, può quindi essere incorporato nel blocco CSP (figura 7.1), dove il CSP *equation facilitator* esegue le funzioni necessarie per calcolare il valore di output, Sc_{Dong}, per i dati in input.

Per individuare gli eventuali errori in altre operazioni è necessario che ci siano equazioni adatte a ciascuna di esse. Alcuni esempi di equazioni per le funzioni ALU più comuni sono mostrati in tabella 7.2.

Funzioni di incremento e decremento possono anche essere realizzate come operazioni aritmetiche protette[7], casi speciali delle funzioni ADD e SUB. Le equazioni

[7] La modalità "protetta" è una modalità di lavoro di un chip nella quale, grazie a uno speciale supporto hardware, si impedisce che un'istruzione possa accedere per sbaglio all'area di memoria dedicata a un'altra istruzione.

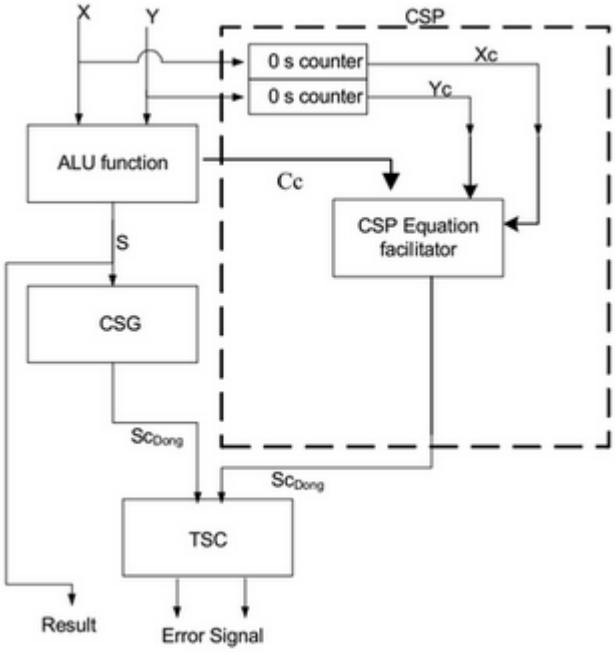


Figura 7.1 – Circuito generico incorporante il CED che utilizza il CSP.

CSP elencate in tabella 7.2 sono usate per generare la prima parte (C1) del Dong Code, mentre la seconda (C2) viene ricavata dal conteggio del numero di zeri in C1. Il simbolo di controllo CSP completo, Sc_{Dong}, è formato da $C1|C2$, ed è confrontato con il simbolo prodotto dal CSG il quale riceve in input gli output dell'ALU o delle unità di registro.

Esempio:
X=5=b0000 0000 0000 0101, quindi, Xc=14
Y=85=b0000 0000 0101 0101, quindi, Yc=12
per l'operazione $X \times Y$, con due operandi da 16 bit. Il carry register Cc risulta Cc=237.

Berger Code (eq. 7.1):
$Sc = nX_c + nY_c - X_cY_c - C_c + n = 16 \times 14 + 16 \times 12 - 14 \times 12 - 237 + 16 = 27 = b11011$
$Check: \ 5 \times 85 = 425 = b00000000000000000000000110101001 \ \Rightarrow Sc = 27 = b11011$

Dall'equazione 7.2, il Dong Code (per k=3):
$C1 = ((X_cY_c + C_c) mod 2^k) + 1 = \overline{(5)} + 1 = b010 + 1 = b011$, $C2 = 1 = b01 \ \Rightarrow Sc_{Dong} = b01101$
$Check: \ 4 \times 85 = 425 = b00000000000000000000000110101001, \Rightarrow C1 = 27 \, mod \, 8 = 3 = b011$, *quindi* $C2 = 1 = b01, \Rightarrow Sc_{Dong} = C1|C2 = b01101$

7.4. Dong Code pipeline operation

Per dare un dimostrazione dell'applicabilità del Dong Code in un contesto realistico e successivamente per applicare il metodo, prendiamo un processore RISC a 32 bit

Operazione	**Equazione CSP**
ADD	$C1 = (X_c + Y_c - C_{in} - C_c + C_{out}) mod 2^k$
SUB	$C1 = (X_c + Y_c + C_{out} - C_c - C_{in}) mod 2^k$
AND	$C1 = (X_c + Y_c - (X \vee Y)_c) mod 2^k$
OR	$C1 = (X_c + Y_c - (X \hat{} Y)_c) mod 2^k$
XOR	$C1 = (X_c + Y_c - 2(X \hat{} Y)_c) mod 2^k$
NAND	$C1 = (-X_c - Y_c + (X \vee Y)_c) mod 2^k$
NOR	$C1 = (-X_c - Y_c + (X \hat{} Y)_c) mod 2^k$
ASHL	$C1 = (X_c + C_{out}) mod 2^k$
ASHR	$C1 = (X_c + C_{out} - X_n) mod 2^k$
LSHL/LSHR	$C1 = (X_c + C_{out} - C_{in}) mod 2^k$
MUL	$C1 = \overline{((X_c Y_c + C_c) mod 2^k)} + 1$
ROL/ROR	$C1 = X_c mod 2^k$
LOAD/STORE	$C1 = X_c mod 2^k$

Tabella 7.2 – Equazioni per il CSP del Dong Code.

con pipeline, implementato sia nello stile asincrono che in quello sincrono, con e senza schema CED.

Operazione	**Descrizione**	**Equivalente**
ADD	Addizione	$R_d = RS_1 + RS_2;$
ADDC	Addizione con carry	$R_d = RS_1 + RS_2 + C_{in};$
SUB	Sottrazione	$R_d = RS_1 - RS_2;$
SUBC	Sottrazione con carry	$R_d = RS_1 - RS_2 - C_{in};$
MUL	Moltiplicazione	$R_d = RS_1 \cdot RS_2;$
AND	AND	$R_d = RS_1 \& RS_2;$
OR	OR	$R_d = RS_1 \| RS_2;$
XOR	Or esclusivo	$R_d = RS_1 \hat{} RS_2;$
JMP	Salto diretto	$PC = Address;$

Tabella 7.3 – Set di comandi.

In tabella 7.3 riportiamo un estratto del set di comandi (per la lista completa si veda [83]) relativi all'ALU, al controllo del flusso del programma, all'accesso alla memoria e allo stato del sistema. L'operazione complessiva può essere spezzata in cinque sezioni: (1) first fetch, (2) load, (3) ALU operation, (4) ALU error detection, (5) store, come mostrato in figura 7.2.

7.4.1. First fetch

Prima che inizi il lavoro, il *program counter* (PC) viene resettato e la prima istruzione che è recuperata con un fetch dalla memoria viene decodificata. In questa fase vengono identificati i registri che saranno interessati nell'operazione e se è richiesto accesso in lettura o scrittura. Il *register file* ha due porte di lettura (A e B) e una per la scrittura (C).

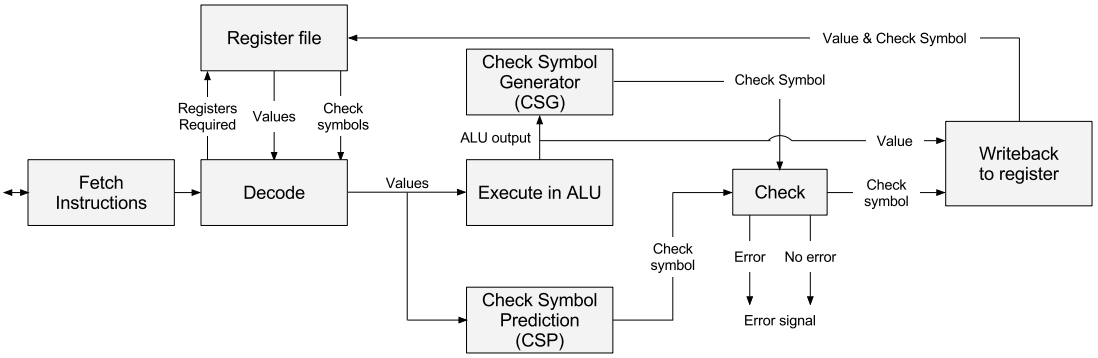


Figura 7.2 – Diagramma del flusso pipeline.

7.4.2. Load operation

Durante l'operazione di LOAD, 37 bit vengono ricevuti dall'ALU (assumendo una parola di informazione di 32 bit e un codice di controllo di tipo Dong Code di altri 5). Inizialmente viene calcolato il numero di zeri nella parola (i primi 32 bit) e confrontato con il valore della parte C1 del simbolo. Se i due numeri non corrispondono viene segnalato un errore, altrimenti l'intera parola (di 37 bit) viene registrata register file dell'ALU configurando l'indirizzo di destinazione e il segnale di write sulla porta C.

7.4.3. ALU

Una volta che il dato è stato caricato nel register file, viene processato da un insieme di funzioni aritmetiche e logiche all'interno dell'ALU. Per la moltiplicazione, ad esempio, l'istruzione comprende un apposito codice operativo (opcode), l'indirizzo del registro di destinazione (Rd) e gli indirizzi dei registri che contengono i due operandi (RS1 e RS2). Successivamente viene eseguito un prelievo di RS1 e RS2 e i valori recuperati finiscono nei registri X e Y in input dell'ALU. L'opcode e il Rd vengono anch'essi registrati in questa fase per poter incrementare il program counter (PC), facendo sì che si porti avanti recuperando e decodificando l'istruzione successiva.

Mentre l'ALU esegue la moltiplicazione, il decoder CSP riceve il codice associato (opcode) e configura il blocco adatto per il calcolo del simbolo di controllo (quindi questo viene fatto in parallelo alla moltiplicazione nell'ALU).

Utilizzando un decoder separato per CSP e ALU, gli errori che si verificano in uno dei due causeranno un *mismatch* in uscita.

7.4.4. ALU Error Detection

Il risultato dell'operazione eseguita nell'ALU è inviato in ingresso all'unità CSG la quale calcola il simbolo di controllo del Dong Code. Questo viene poi confrontato con il simbolo generato in precedenza dal CSP, per mezzo di un'unità chiamata *Totally Self Checking* (TSC) *checker*. Se i due simboli di controllo non combaciano, viene segnalata la presenza di un errore con un *flag*. Questo metodo individua gli errori come mostrato in tabella 7.1, e si applica alle operazioni che coinvolgono l'ALU e a quelle di caricamento/salvataggio nei registri. Non vi è alcun controllo di errore per la selezione dei registri, ma si verificherà comunque un mismatch tra il dato recuperato e il simbolo di controllo corrispondente se questi provengono da registri differenti (per

via di un errore sull'indirizzo). Ciò accade perché in ogni caso le unità di decode sono separate.

7.4.5. Store Operation

Se non si verificano errori, i simboli generati da CSP e CSG sono identici e quindi il dato viene scritto nel register file all'indirizzo Rd, attraverso la porta C.

7.5. Asynchronous Operation

L'applicazione del Concurrent Error Detection a un sistema asincrono fornisce un ottimo metodo alternativo per la correzione di errori "soft" rispetto ai metodi applicati nei sistemi sincroni. Quando si verifica un fault transitorio, il sistema produce un segnale d'errore che può essere interpretato dall'ambiente in modi differenti. Nei sistemi sincroni, quando viene individuato un errore il sistema deve resettare il processo in cui l'errore si è verificato, rifare i conti e le verifiche, lasciando in stallo la pipeline finché l'errore non sia stato corretto. I sistemi asincroni invece possono interpretare un errore in due modi:

- **Errore transitorio, tentativo di recupero**
  Questo metodo permette allo stadio della pipeline in cui si è verificato l'errore di aspettare finché l'errore non scompare. Quando è attiva una segnalazione d'errore, il segnale *completed* non può essere attivato finché la segnalazione d'errore non viene ritirata. Senza il controllo in parallelo (CED) l'errore si propagherebbe causando la corruzione dei dati. In un sistema sincrono il metodo *wait*, che sospende le operazioni finché l'errore non è risolto, causerebbe dei delay molto più lunghi perché dovrebbe attendere per diversi cicli di clock prima di poter permettere al sistema di proseguire.

- **Errore transitorio (estremo)/permanente**
  Un sistema che assuma la sola esistenza di errori transitori, la cui vita dura solo un breve lasso di tempo, è soggetto a situazioni di blocco (lock-up) nel caso di errori permanenti, poiché un errore sarebbe segnalato per un tempo indefinitamente lungo. Per ovviare a questo problema, si concede un tempo massimo di attesa perché l'errore rientri e, quando ciò non si verifica, viene forzato un reset in modo da impedire che il sistema stalli in perenne attesa. Esistono appositi circuiti *timer*, la cui implementazione non occupa molta area, che si preoccupano di controllare da quanto il sistema sta attendendo perché un errore si risolva e, trascorso un certo lasso di tempo, forzano il reset. Circuiti di questo tipo vengono comunemente usati in molti microprocessori.
  Come nei circuiti asincroni, anche in quelli sincroni può succedere che errori permanenti causino il blocco del sistema e, anche in questo caso, l'uso di circuiti *timer* risolve il problema, anche se mediamente i tempi di attesa prima di poter forzare un reset sono più lunghi rispetto a quelli utilizzati nel circuiti asincroni.

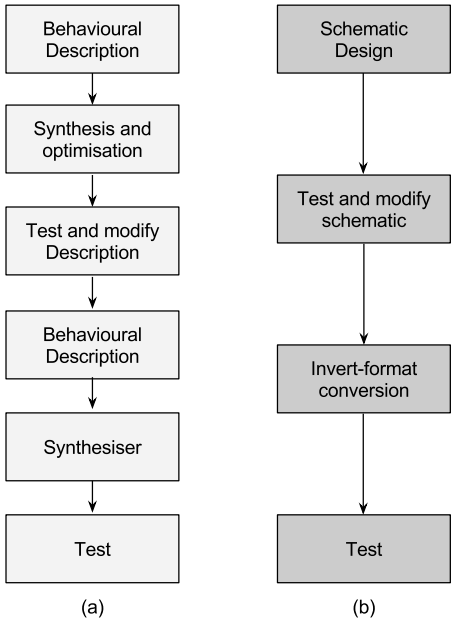


Figura 7.3 – Flusso di progettazione: (a) tradizionale e (b) schematico.

7.6. Implementazione su FPGA

Un FPGA (*Field-Programmable Gate Array*) è un circuito integrato progettato per poter essere configurato dall'utilizzatore. La configurazione solitamente viene eseguita tramite un linguaggio HDL (*Harware Description Language*) simile a quelli impiegati per i circuiti ASIC (*Application Specific Integrated Circuit*). Un FPGA può essere configurato per eseguire qualunque funzione logica sia possibile eseguire su un ASIC.

```
Start
 Repeat
  Parse verilog input file
  Locate instance gate name
  Replace with XilinxPrimitiveGate (XPG)
  Replace port names with XPG port names
  Maintain net names and pin names for
             comprehension between designs
 Repeat until end of file / no input files
End
```

Figura 7.4 – Pseudo-codice del programma di conversione CXP.

In figura 7.3(a) è illustrato il flusso di progettazione convenzionale per prototipi ad alta velocità implementati su una FPGA Xilinx, mentre in figura 7.3(b) è mostrato il flusso di progettazione sul livello schematico. Quando si desidera ottenere un'elevato tasso di individuazione degli errori, è spesso necessario implementare codici di stato ridondanti. Questo viene fatto per impedire che inversioni di bit possano portare il sistema a entrare in stati non corretti. Durante la sintesi, i modelli comportamentali all'interno del design sono interpretati e convertiti in logica. Questo processo è

soggetto a ottimizzazioni (in termini di sintesi) ed è diverso per ciascuna implementazione e persino per sintetizzatori differenti. Per mantenere l'equivalenza tra un prototipo FPGA e il chip finale, è quindi necessario lavorare sul livello schematico con porte logiche standard e a tal fine è stato sviluppato un programma di conversione (*CXP: Cadence Xilinx Parser*), mostrato in figura 7.4, per facilitare la conversione tra i due sistemi senza incappare nei problemi collegati all'ottimizzazione del circuito.

Questo processo di conversione fornisce un meccanismo che permette di progettare il chip utilizzando una qualsiasi delle due metodologie, tradizionale o schematica. Il flusso di progettazione in 7.3(b) garantisce maggiore flessibilità al progettista e gli permette anche di accertare le funzionalità di un circuito su un dispositivo riconfigurabile, rendendo possibile la realizzazione hardware di un prototipo per la successiva fase di test intensivi.

7.7. Risultati

Architettura	**Potenza (mW)**	**Differenza**
(1) Sync. ASIC	116.3	Riferimento (0%)
(2) Sync. CED, ASIC	149.1	> 35% rispetto a (4)
(3) Async. ASIC	95.3	< 22% rispetto a (1)
(4) Async. CED, ASIC	110.5	> 16% rispetto a (3) < 5% rispetto (1)

Tabella 7.4 – Consumo dell'implementazione asincrona vs. sincrona.

I risultati che presentiamo sono relativi a quattro diverse implementazioni di un processore RISC a 32 bit: sincrono/asincrono, su FPGA/ASIC. Per l'implementazione FPGA è stata utilizzata la Cadence Design Suite su un processo a $0.35\mu m$ sull'FPGA Xilinx xc2v1000-6fg456.

Come mostrato in tabella 7.5, per l'implementazione ASIC asincrona è stato ottenuto un risparmio del 4%, in termini di area, per i modelli che non utilizzano il CED. Al contrario, le implementazioni che utilizzano il CED (con Dong Code) su tutte le funzioni dell'ALU e sui suoi 32 registri a 32 bit, richiedono un'area maggiore del 13% per l'implementazione ASIC.

Per l'FPGA invece, il design asincrono risparmia il 5% in termini di numero di porte rispetto al design sincrono. Sempre rispetto all'FPGA, il numero di porte per i design sincrono e asincrono che utilizzano il CED, aumenta del 26 e 20% rispettivamente se paragonati agli equivalenti senza CED.

Per quanto riguarda il risparmio di potenza invece, ci riferiamo ai dati in tabella 7.4: la versione sincrona con CED consuma il 35% in più dell'equivalente asincrona, mentre per le versioni senza CED, quella asincrona consuma il 22% in meno di quella sincrona.

Questi valori sono stati calcolati utilizzando un test composto da 32 operazioni tra cui *load*, *store* e numerose operazioni dell'ALU. Per via della limitatezza del set di comandi offerto dalle implementazioni RISC, fornire risultati relativi al consumo di potenza e alle prestazioni, ricavati con programmi di benchmark industriali (ad esempio Dhrystone/Whetstone), significherebbe fornire risultati ingannevoli e quindi

Architettura	**Area**	**Differenza**
(1) Sync. ASIC	$1846 \times 10^3 \mu^2$	Riferimento (0%)
(2) Async. ASIC	$1774 \times 10^3 \mu^2$	< 4% rispetto a (1)
(3) Sync. CED, Dong Code, ASIC	$2100 \times 10^3 \mu^2$	> 13% rispetto a (1)
(4) Async. CED, Dong Code, ASIC	$2010 \times 10^3 \mu^2$	> 13% rispetto a (2)
(5) Sync. CED, Berger Code, ASIC	$2165 \times 10^3 \mu^2$	> 17% rispetto a (1)
(6) Async. CED, Berger Code, ASIC	$2090 \times 10^3 \mu^2$	> 17% rispetto a (2)
(7) Sync. FPGA	$45949\,(gate\,count)$	Riferimento (0%)
(8) Async. FPGA	$43851\,(gate\,count)$	< 4% rispetto a (7)
(9) Sync. CED, Dong Code, FPGA	$58018\,(gate\,count)$	> 26% rispetto a (7)
(10) Async. CED, Dong Code, FPGA	$52927\,(gate\,count)$	> 20% rispetto a (8)

Tabella 7.5 – Area dell'implementazione asincrona vs. sincrona.

non comparabili. I valori per il consumo di potenza, espressi in tabella 7.4, sono stati ottenuti con il *Synopsys Nanosim* per il design ASIC.

8. AsAP

8.1. Introduzione

L'elaborazione di segnali digitali (DSP) al giorno d'oggi è sempre più richiesta, e il carico di lavoro per le applicazioni che svolgono questo compito è sempre più complesso. Queste applicazioni spesso devono svolgere molteplici task[(1)] (di tipo DSP) e sovente costituiscono i componenti fondamentali di svariati sistemi come: comunicazioni via cavo e wireless, multimedia, sensori remoti ed elaborazione dati, applicazioni biomediche, etc. Molte di queste sono incorporate in un sistema più ampio, e sono fortemente vincolate in termini di energia e costo. Inoltre, spesso richiedono un elevato throughput e dissipano un porzione significativa della potenza consumata complessivamente dal sistema. Per queste ragioni, è di fondamentale importanza massimizzarne le prestazioni (che spesso significa massimizzare il throughput) e minimizzarne sia la dissipazione di energia per operazione che il costo in termini di area di silicio.

Il fatto che le frequenze di lavoro dei clock moderni siano sempre più elevate, così come il numero dei circuiti presenti su un singolo chip, fa sì che le prestazioni dei chip siano limitate da fattori come il consumo di potenza piuttosto che da vincoli tecnici di circuito. Questo implica una nuova era di design ad alte prestazioni focalizzata su implementazioni che consumino il meno possibile. Oltretutto, le tecnologie di fabbricazione del futuro impongono nuove sfide, come le variazioni dei parametri nei grandi circuiti e i delay su linea, che possono ridurre significativamente le frequenze massime dei clock. Vale la pena quindi concentrarsi in particolare su quelle architetture che accettano queste sfide e sfruttano i vantaggi delle tecnologie del futuro.

Ci sono diversi approcci al design di processori per l'elaborazione del segnale. Per esempio l'ASIC può fornire altissime prestazioni ed efficiente impiego dell'energia, ma è un approccio poco flessibile per quanto riguarda la programmazione. Al contrario, i DSP programmabili (pDSP) offrono grande flessibilità, ma sono meno prestanti e consumano di più. Gli FPGA costituiscono invece una sorta di via di mezzo. L'obiettivo del progetto AsAP (*Asynchronous Array of Simple Processors*) è di sviluppare un sistema che gestisca enormi carichi di lavoro (in termini di elaborazione dei segnali) fornendo elevate prestazioni e, al tempo stesso, gestendo in modo efficiente i consumi. Questo approccio si dimostra essere particolarmente adatto per l'implementazione con le tecnologie più avanzate e mantiene la flessibilità (e la semplicità nella programmazione) dei processori programmabili.

Il sistema AsAP comprende una matrice (*2D-array*) di semplici processori programmabili interconnessi tramite una mesh configurabile [14]. I processori, ciascuno dei quali possiede un clock indipendente (e all'occorrenza arrestabile), sono inseriti in uno schema GALS (si veda capitolo 3 a pagina 29) [20].

[(1)] La parola inglese *task*, che significa "compito", "lavoro" è di difficile traduzione in questo contesto perciò lasceremo scritto *task, task-level*, etc.

AsAP utilizza un'architettura con semplici processori dotati di piccole memorie per incrementare notevolmente l'efficienza nei consumi. Questa configurazione possiede una flessibilità che le permette di essere adattata con successo ad altri scopi per soddisfare i requisiti di svariate applicazioni e ciò ammortizza l'elevato costo di progettazione. Lo stile GALS e la comunicazione di tipo *nearest-neighbor* (cioè col blocco più vicino) potenziano la scalabilità del circuito e permettono di mitigare effetti quali le variazioni dei dispositivi, le limitazioni delle linee globali e i *failures* di singoli processori. Un prototipo per un chip 6x6 AsAP, perfettamente funzionante, è stato implementato con tecnologia CMOS $0.18\mu m$ [124].

Il punto di forza di questa architettura sta nel fatto che si presta per implementare il parallelismo di tipo *task-level*. A differenza del parallelismo di tipo *data-level*, dove più processori lavorano in contemporanea su uno stesso insieme di dati, nel parallelismo di tipo task-level, ciascun processore viene dedicato a un task ben preciso, e opera in modo indipendente su una certa porzione di dati. Il concetto è simile a quello di una catena di montaggio, dove le diverse parti che andranno a comporre il prodotto finito vengono assemblate con una sequenza ben precisa dai macchinari facenti parte della catena. Anche se il singolo prodotto viene elaborato in sequenza entrando nella prima macchina, passando poi alla seconda, e così via fino all'ultima, a regime, tutte le macchine lavorano su pezzi diversi. L'elaborazione è quindi effettuata in modo sequenziale sul singolo prodotto, ma avviene in parallelo su un insieme di prodotti la cui cardinalità è pari al numero delle macchine che compongono la catena.

Per dare un'idea del guadagno fornito da questa architettura immaginiamo due circuiti che eseguono una generica trasformazione su un dato. Il primo circuito prende in input il dato, lo elabora, lo invia in output e poi ricomincia dal dato successivo. Il secondo circuito invece, è in grado di suddividere l'elaborazione in 5 fasi indipendenti (cioè in 5 task), a cui dedica 5 processori. In questo circuito, un dato in input entra nel primo processore, subisce un'elaborazione, viene inviato al secondo il quale, a sua volta, lo elabora e lo invia al terzo, e così via. Supponendo per semplicità che i 5 processori impieghino tempi simili per elaborare i dati, possiamo facilmente comprendere che nel secondo circuito i dati entreranno con un ritmo 5 volte maggiore rispetto al primo. Questo significa che il secondo circuito, pur mantenendo la corretta sequenzialità delle operazioni sul singolo dato, è in grado di operare su 5 dati alla volta (quindi in parallelo) e quindi raggiunge velocità circa 5 volte maggiori rispetto al primo.

Il fatto che la comunicazione tra un processore e il suo vicino sia gestita in modo asincrono, permette al circuito di godere di 2 ulteriori vantaggi. Innanzitutto non è necessaria una sincronizzazione del flusso dati. Ciascun processore riceve i dati in input quando è pronto, li elabora e quindi li spedisce in output. Non è necessario che il flusso di dati sia regolato in termini di tempo, perché i processori sono in grado di comunicare tra di loro in modo autonomo. Supponiamo che il processore P_n debba spedire i propri dati al processore P_{n+1}. Se quest'ultimo è occupato, il processore P_n si sospende, in attesa di poter completare la spedizione. Non appena il processore P_{n+1} è pronto a ricevere nuovi dati in input, il processore P_n viene risvegliato e porta a termine la spedizione. Questo meccanismo chiaramente permette di risparmiare potenza, sia per la mancanza di un clock globale che regoli l'intero flusso dati, sia per

la capacità di sospensione di ciascun processore.

Il secondo vantaggio è il risparmio sulla memoria, come spiegato in sez. 8.2.1.

8.2. Motivazioni e funzionalità principali

Molteplici funzionalità chiave distinguono il processore AsAP. In figura 8.1 sono illustrate le funzionalità e i benefici che apportano.

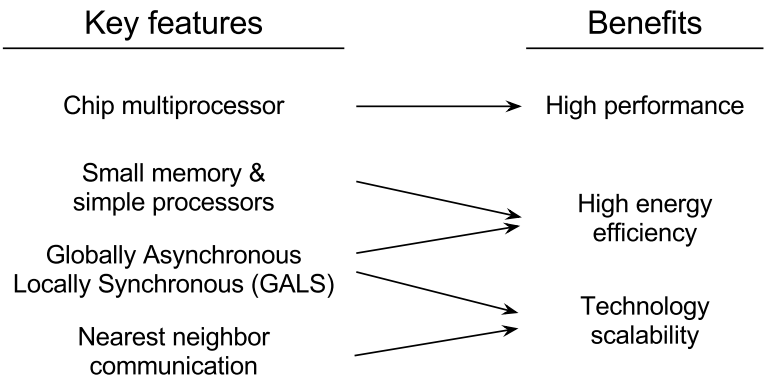


Figura 8.1 – Funzionalità chiave dell'AsAP e benefici risultanti.

8.2.1. Chip multiprocessore e parallelismo di tipo task-level

Incrementare la frequenza di clock dei processori ha permesso di incrementarne le prestazioni ma, ultimamente, questo approccio diventa di sempre più difficile applicazione. Le CPU più avanzate consumano già più di $100W$ a $2GHz$ [103]. Il costo per raffreddare tali circuiti abbassa la soglia del massimo consumo di potenza accettabile e riduce le frequenze e le prestazioni che il processore potrebbe raggiungere [68]. La tecnica di incrementare la frequenza del processore approfondendo gli stadi della pipeline sta raggiungendo il suo limite, poiché richiede sempre più registri e logica di controllo e di conseguenza il design è sempre più difficile e il prodotto meno efficiente.

Un altro approccio per aumentare le prestazioni di un sistema consiste nella parallelizzazione a livello di istruzioni, dati e/o task. Il parallelismo di tipo task-level è particolarmente adatto per molte applicazioni DSP, ma sfortunatamente non può essere usato con facilità sui processori tradizionali che lavorano in modo sequenziale.

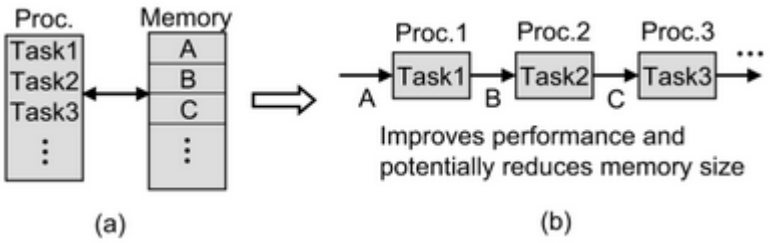


Figura 8.2 – Applicazione multi-task eseguita su: (a) architettura tradizionale e (b) stream-oriented multi-processor adatto al parallelismo task-level.

In figura 8.2(a) è illustrato un sistema tradizionale che impiega un processore potente che esegue i task in sequenza, ed è dotato di una memoria capiente che trattiene i risultati intermedi delle varie elaborazioni. La stessa applicazione può girare su un multi-processore utilizzando in modo molto efficiente il parallelismo di tipo task-level, come illustrato in figura 8.2(b), dove ciascun processore si prende cura di un task differente. Ogni processore invia i propri risultati al successivo e ciò elimina la necessità di avere una memoria molto ampia che trattenga risultati intermedi.

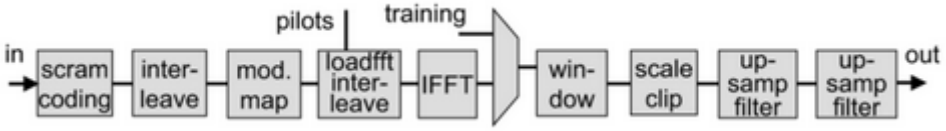


Figura 8.3 – Percorso di trasmissione in banda base per una LAN wireless IEEE 802.11a/g (54Mb/s, 5/2.4GHz).

Il parallelismo di tipo task-level è quindi molto popolare nelle applicazioni DSP. La figura 8.3 mostra un esempio di una moderna applicazione complessa che esibisce un alto tasso di parallelismo di tipo task-level: la catena di trasmissione di un trasmettitore LAN wireless di tipo IEEE 802.11a/g. Contiene più di 10 task e ciascuno di essi può essere mappato direttamente su un processore dedicato, in modo da suddividere il lavoro sfruttando il parallelismo di tipo task-level.

Per fare un esempio su come questo sistema funziona, immaginiamo che abbia 10 task $t_1..t_{10}$, e supponiamo di dover processare un flusso dati $d_0, d_1, d_2, ...$ Al tempo 0, il dato d_0 entra nel processore t_1, il quale comincia l'elaborazione. Quando t_1 termina il proprio compito, invia d_0 a t_2, il quale a sua volta comincia l'elaborazione. Nel frattempo t_1 riceve in ingresso il dato d_1 e ricomincia. Seguendo questa logica, la catena $t_1, ..., t_{10}$ si popola task dopo task. A regime, il processore t_{10} lavora sul dato d_n, t_9 lavora su d_{n+1}, t_8 lavora su d_{n+2} e via dicendo fino a t_1, che lavora su d_{n+9}. In questo modo, la sequenzialità delle operazioni è mantenuta per ciascun singolo dato d_m, ma di fatto l'elaborazione avviene in parallelo su 10 dati differenti.

8.2.2. Requisiti di memoria dei task

Per via del sempre crescente numero di transistor per singolo chip, i moderni processori programmabili utilizzano una crescente quantità di memoria *on-chip* e di conseguenza aumenta anche la percentuale di area del chip a questa dedicata.

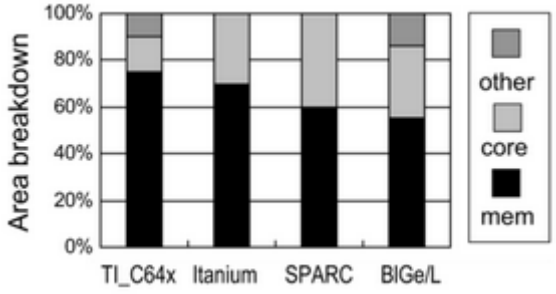


Figura 8.4 – Dettaglio dell'area per 4 processori moderni.

La figura 8.4 mostra il dettaglio dell'area per quattro moderni processori con memorie che occupano tra il 55 e il 75% dell'area totale [103, 43, 1, 102]. Ampie memorie riducono l'area disponibile per le unità funzionali, consumano molta potenza e richiedono tempi maggiori per le transazioni, quindi le architetture che minimizzano la dimensione della memoria possono sfruttare meglio l'area del chip e ottenere così maggiore efficienza e prestazioni. Una notevole caratteristica dei DSP dedicati con task specializzati è che molti hanno requisiti di memoria limitati se messi a confronto con sistemi general-purpose. Per calibrare bene questi nuclei (*kernels*) è necessario che il livello di memoria sia distinto tra quella "che può essere usata" e quella che "tipicamente viene usata".

Task	**Requisiti memoria istruzioni (words)**	**Requisiti memoria dati (words)**
N-point FIR	6	$2N$
8-point DCT	40	16
8x8 2-D DCT	154	72
Conv. coding (k=7)	29	14
Huffman encoder	200	350
N-point convolution	29	$2N$
64-point complex FFT	97	192
N merge sort	50	N
Square root	62	15
Exponential	108	32

Tabella 8.1 – Requisiti di memoria per task DSP comuni, assumendo un processore di tipo *single-issue*.

La tabella 8.1 illustra gli effettivi requisiti di memoria (istruzioni/dati) per differenti task usati comunemente in applicazioni DSP. Queste cifre sono riferite a un semplice processore di tipo *single-issue*[(2)], *fixed-point*. I dati mostrano che alcune centinaia di parole (words) sono sufficienti per molti task e quindi è sufficiente un quantitativo di memoria di gran lunga inferiore a quanto invece si trova nei moderni processori DSP (tra i $10KByte$ e i $10MByte$).

8.2.3. Approccio GALS

Normalmente un moderno circuito integrato implementa un approccio di tipo *Globally Synchronous*, ma con l'aumentare dei delay su linea e delle variazioni nei parametri delle tecnologie di tipo *deep-submicron*, diventa sempre più difficile progettare circuiti estesi dotati di clock che operano su frequenze elevate. Oltretutto, un clock globale ad alta frequenza consuma una parte significativa della potenza totale consumata dal circuito. Per esempio, 1/4 della potenza consumata da un core-2 Itanium [103] viene consumato dai circuiti di distribuzione del clock e dal clock buffer finale. In più, l'approccio sincrono manca della flessibilità necessaria per poter controllare in modo indipendente la frequenza del clock tra le sotto-componenti del sistema (per poter aumentare l'efficienza).

[(2)]Un processore è di tipo *single-issue* se può eseguire una singola istruzione per ciclo di clock.

L'approccio GALS è molto diverso e potrebbe migliorare le prestazioni sia in termini di velocità che di efficienza ma, come abbiamo già detto nella parte introduttiva di questo testo, al momento mancano gli strumenti per il design e quindi le implementazioni risultano difficili da progettare e i circuiti risultanti non sono così efficienti per via delle dimensioni che occupano.

L'approccio GALS separa le componenti del circuito che processano i dati in modo che ciascuna possa operare con un clock indipendente. L'uso di questa architettura permette di eliminare i problemi legati alla distribuzione del segnale globale di clock e ciò apporta benefici in termini di consumo e di complessità di progetto. Un altro beneficio significativo consiste nell'opportunità di spegnere facilmente (e completamente) un circuito che non debba compiere nessun lavoro. Inoltre, il fatto che i clock operano in modo indipendente rende possibile scalare le frequenze in modo indipendente e ciò, unito alla conseguente possibilità di scalare anche la tensione di alimentazione, riduce drasticamente il consumo.

8.2.4. Linee di comunicazione

L'ultimo punto che esploriamo in questa sezione dedicata alle motivazioni e alle funzionalità, riguarda le linee di comunicazione *on-chip*. Queste linee di comunicazione creano una sfida davvero considerevole poiché aumentano il consumo del chip come pure i ritardi. Come abbiamo già evidenziato, l'architettura GALS si sta imponendo come potenziale alternativa proprio perché risolve i problemi legati alle lunghe linee che trasportano il segnale di clock lungo tutto il circuito, riducendo i consumi e semplificando il design.

Ci sono diversi metodi per eliminare le linee globali. Soluzioni NoC (*Networks on Chip*) [118] trattano moduli differenti (su un chip) come se fossero nodi differenti in una rete e utilizzano tecniche di instradamento (*routing*) al posto di semplici collegamenti con bus per trasferire i dati. I NoC forniscono un potente metodo di comunicazione, ma spesso consumano grandi quantità di area e potenza. Un altro metodo consiste nella comunicazione locale, dove ciascun processore è connesso solo a processori appartenenti al dominio locale. Uno degli esempi più semplici è la cosiddetta *nearest neighbor communication*, nella quale ciascun processore è connesso e comunica direttamente solo con i processori a esso adiacenti. Questa architettura è altamente efficiente in termini di area e consumo e può fornire una comunicazione sufficiente per molte applicazioni DSP. La sfida più difficile, quando si utilizza la *nearest neighbor communication*, consiste nell'accoppiare in modo efficiente applicazioni che esibiscono una comunicazione distante. Fortunatamente, per molte applicazioni ciò è possibile e quindi questo metodo si rivela particolarmente efficace.

8.3. Il sistema AsAP

Veniamo ora al sistema di processori AsAP. Questo sistema comprende un vettore bidimensionale (cioè una matrice) di semplici processori programmabili. Ciascun processore possiede un clock regolato secondo lo stile GALS e le interconnessioni sono gestite da una mesh riconfigurabile. I processori AsAP sono ottimizzati per calcolare in modo particolarmente efficiente algoritmi DSP in modo individuale o in unione con i processori vicini.

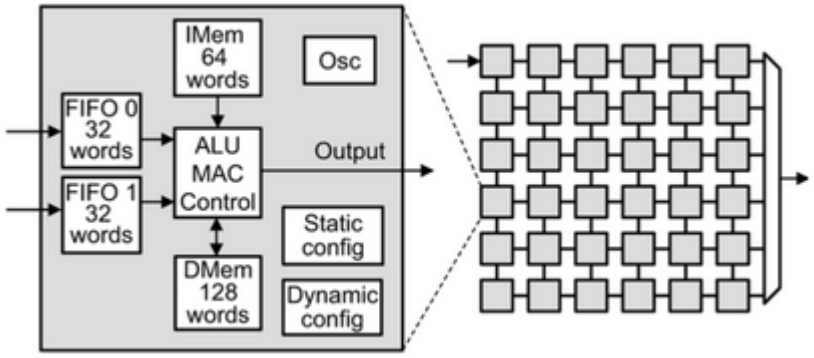


Figura 8.5 – Diagramma di un processore AsAP.

La figura 8.5 mostra il diagramma a blocchi di un processore AsAP e la risultante matrice di processori, organizzata come una mesh 6x6. I dati entrano nella matrice dal processore in alto a sinistra ed escono da uno dei processori che appartengono alla colonna di destra, selezionato da un mutex. Circuiti di I/O sono disponibili su ciascun lato dei processori periferici ma, per via delle limitazioni del *package* I/O utilizzato, in questo caso molti di loro sono sconnessi.

Ciascun processore è di tipo *single-issue* e contiene: un oscillatore clock locale; due interfacce asincrone di tipo dual-clock che forniscono la comunicazione con i processori adiacenti; una semplice CPU che include un'ALU, un MAC[(3)] e logica di controllo. Ciascun processore inoltre contiene una instruction memory a 64 parole , una data memory a 128 parole e della logica (a configurazione statica e dinamica) che fornisce funzioni configurabili come ad esempio le modalità di indirizzamento e interconnessione con gli altri processori. Ogni processore può ricevere e spedire dati ai processori adiacenti. Fino a un massimo di due, in ricezione, mentre in trasmissione può spedire a una qualsiasi combinazione dei suoi quattro vicini. Le porte di ingresso sono due perché questa è la configurazione che meglio si adatta alle applicazioni che abbiamo visto, in quanto l'uso di una terza porta è sfruttato molto di rado.

AsAP supporta 54 istruzioni di tipo RISC e, oltre alla *bit-reverse*, che è utile per il calcolo della FFT, non sono state implementate istruzioni specifiche per altri algoritmi. Lo stile di design utilizzato è il GALS, con un oscillatore locale all'interno di ogni processore e l'ampiezza massima raggiunta dagli alberi di distribuzione dei singoli clock è minore di $1mm$ con la tecnologia a $0.18\mu m$. Questo approccio presenta notevole scalabilità e rende semplice aggiungere altri processori alla matrice. In un sistema sincrono, l'albero di distribuzione del clock globale deve essere ridisegnato quando si aggiungono altri processori e ciò può risultare estremamente difficile per processori di grandi dimensioni.

8.3.1. Design di un singolo processore AsAP

Pipeline e datapath

Ciascun processore AsAp possiede una pipeline a 9 stadi, come mostrato in figura 8.6. Lo stadio *IFetch* esegue il fetch delle istruzioni secondo quanto indicato dal

[(3)]MAC: Multiplier ACcumulator.

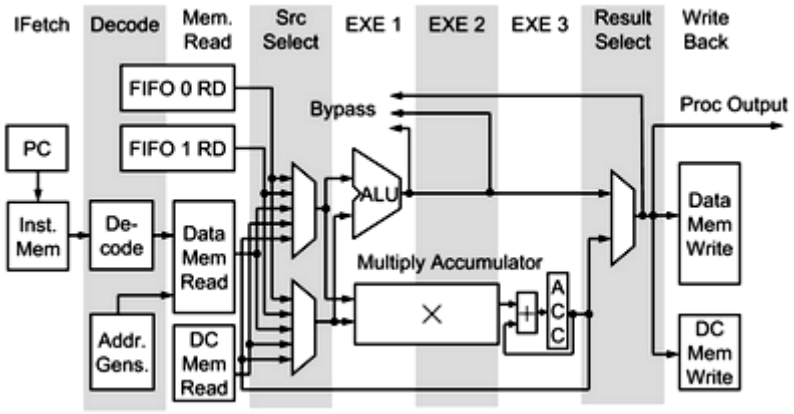


Figura 8.6 – Pipeline a 9 stadi AsAP.

program counter (PC). Non sono implementati circuiti di branch-prediction e tutti i segnali di controllo sono generati nello stadio *Decode* e inseriti nella pipeline in modo appropriato. Gli stadi *Mem Read* e *Src Select* effettuano il fetch dei dati dalla *data memory* (DMEM), dall'*immediate field* (l'interfaccia asincrona FIFO di collegamento con gli altri processori), dalla *dynamic configuration memory* (DCMEM), dall'*accumulator register* (ACC) o dalla logica di trasmissione ALU/MAC. Lo stadio di *execution* occupa 3 cicli e impiega logica apposita per eseguire un bypass sull'ALU e sul MAC, alleviando le penalizzazioni dovute ai *data hazards* nella pipeline. Gli stadi *Result Select* e *Write Back* selezionano i risultati in uscita dall'ALU e dal MAC e li scrivono in memoria (data memory, DC memory) o sugli output verso i processori adiacenti. L'unità MAC è divisa in 3 stadi per abilitare (i) elevate frequenze di clock e (ii) la capacità di avviare l'esecuzione di istruzioni MAC e moltiplicative in ogni ciclo (figura 8.7).

Il primo stadio genera i prodotti parziali del moltiplicatore 16×16, mentre il secondo usa sommatori di tipo *carry-save* per comprimere i prodotti parziali in un unico output a 32 bit anch'esso di tipo *carry-save*. Il terzo stadio utilizza un sommatore a 40 bit per sommare il risultato del secondo stadio con il valore dell'*accumulator register* (ACC, a 40 bit). Poiché l'ACC normalmente viene letto di rado, è possibile leggere direttamente solo i suoi 16 bit meno significativi, mentre i 16 bit più significativi vengono letti grazie a uno shift di 16 posizioni verso destra (quindi vanno a occupare il posto dei 16 bit meno significativi). Questa semplificazione riduce l'hardware e accorcia i tempi nello stadio finale, che è quello critico.

Oscillatore locale

La figura 8.8 illustra lo schema dell'oscillatore clock programmabile da cui ogni processore trae il proprio segnale di clock. L'oscillatore è un complesso anello costituito interamente da celle standard. Tre metodi sono usati per configurarne la frequenza:

1. La dimensione dell'anello può essere configurata a 5 o 9 stadi mediante il segnale *stage_sel*.

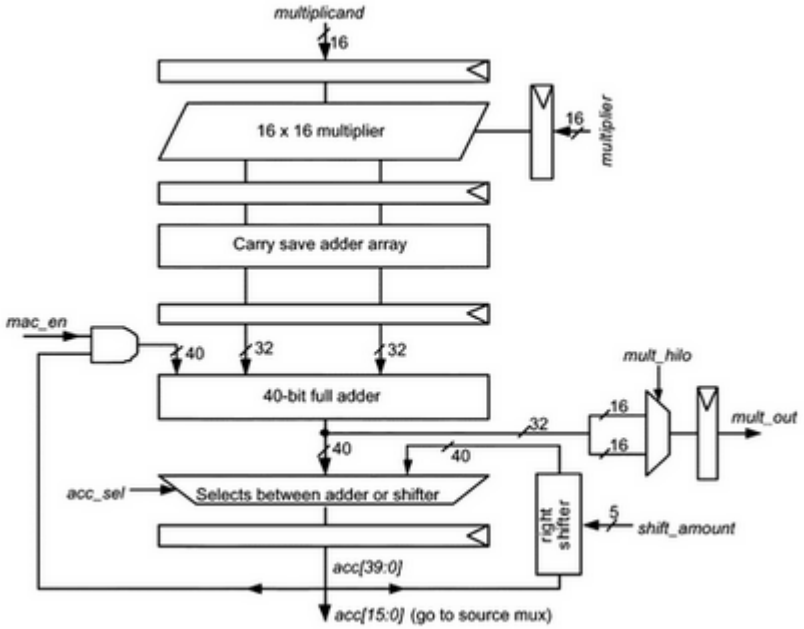


Figura 8.7 – Diagramma dell'unità MAC a 3 stadi.

2. Sette inverter a 3 stati sono connessi in parallelo con ciascun inverter. Quando un inverter a 3 stati è acceso, il carico di quello stadio aumenta e così pure la frequenza dell'anello [113].

3. Un componente *clock divider* in output può dividere il segnale di clock fino a 128 volte.

Il segnale di *halt* permette l'arresto immediato del clock (cioè senza che il clock emetta impulsi parziali prima di arrestarsi). Per mantenere la corretta operatività, i processori devono entrare in stallo quando tentano di leggere dati da una FIFO vuota o quando tentano di scriverne in una FIFO piena. L'oscillatore può essere configurato per arrestarsi durante gli stalli, in modo che il processore non consumi potenza di nessun tipo (eccezion fatta per le correnti di *leakage*) mentre è in stallo.

La figura 8.9 mostra un esempio di forma d'onda relativa all'arresto e ripartenza di un clock. Il segnale *stall_fifo* è inviato quando l'accesso alla FIFO viene sospeso per via di una delle due condizioni sopracitate. Dopo un periodo di 9 cicli di clock, durante i quali la pipeline del processore forza lo scorrimento dei dati, il segnale *halt_clk* va a 1, arrestando l'oscillatore. Il segnale *stall_fifo* torna a 0 quando la causa dello stallo è stata rimossa e quindi *halt_clk* fa ripartire l'oscillatore alla massima velocità in meno di un periodo di clock. Utilizzando questo metodo, il consumo viene ridotto del 53% per un *encoder JPEG* e del 65% per un trasmettitore 802.11a/g, semplicemente facendo sì che il consumo di potenza attiva nei periodi in cui il processore non ha lavoro da svolgere sia nullo.

In figura 8.10(a e b) sono illustrate le frequenze per l'oscillatore con anello a 5 e 9 inverter, rispettivamente. Nel complesso, l'oscillatore può produrre 524288 frequenze diverse in un range che spazia da $1.66MHz$ a $702MHz$ (fig. 8.10(c)). In (d) invece è illustrato il numero di occorrenze di diversi salti di frequenza nel range utile $[1.66, 500]MHz$. In questo intervallo, circa il 99% dei salti è minore di $0.01MHz$, il

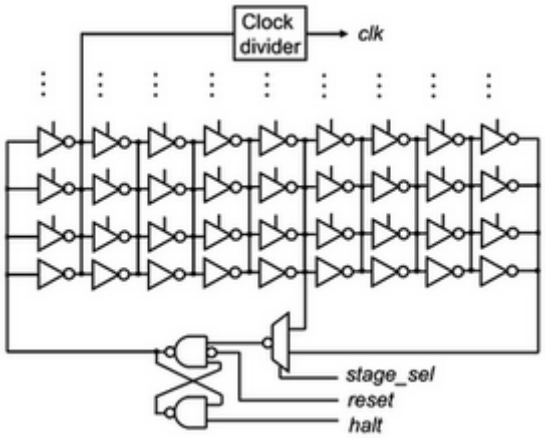


Figura 8.8 – Oscillatore clock programmabile.

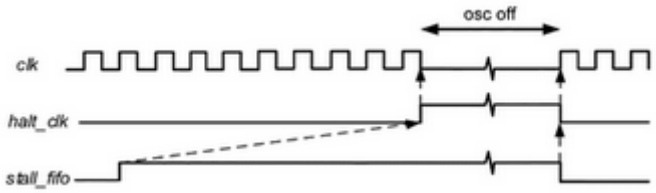


Figura 8.9 – Esempio di una forma d'onda relativa all'arresto e ripartenza di un clock.

più ampio essendo $0.06MHz$.

Nonostante il layout sia lo stesso per ogni processore, le variazioni di processo fanno sì che diversi processori sul medesimo chip abbiano prestazioni differenti. La figura 8.11 mostra una misura delle frequenze degli oscillatori eseguita su processori appartenenti allo stesso chip e aventi la medesima configurazione. L'oscillatore del processore in basso a destra ha una frequenza maggiore di $540MHZ$, mentre diversi altri nella riga in alto hanno frequenze inferiori ai $500MHZ$. Essendo AsAP un vero array di processori GALS, il suo design è scevro di qualsiasi nuova problematica che non sia presente in sistemi digitali standard, dotati di clock, a eccezione di possibili metastabilità negli attraversamenti tra domini di clock, che possono comunque essere ridotte a livelli estremamente bassi con l'uso di registri di sincronizzazione configurabili. Lo stesso vale per l'hardware (buffer FIFO inclusi) e per il software, per via delle variazioni, nelle frequenze dei clock dei processori, dovute a effetti come *jitter*, *skew*, *halt*, *restart* e a variazioni di frequenza o di processo. L'unico elemento che può subire l'impatto delle suddette variazioni è il throughput.

8.3.2. Design della comunicazione tra processori

L'architettura AsAP connette tra loro i processori tramite una mesh 2-D configurabile, come mostrato in figura 8.12. Per mantenere la comunicazione da un processore all'altro alla massima frequenza di clock, le connessioni sono create solo tra processori adiacenti. Ciascun processore possiede due porte asincrone di input e può connetterle a ciascuno dei suoi (al più) 4 vicini. Le connessioni in input di ogni processore nor-

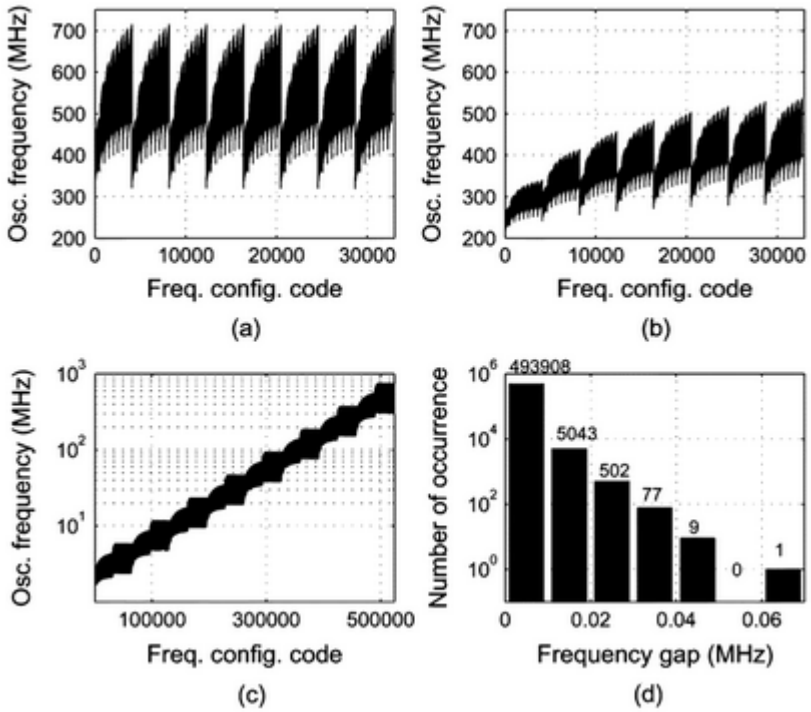


Figura 8.10 – Misura dei dati relativi a un oscillatore per un singolo processore: frequenze per l'anello con (a) 5 e (b) 9 inverter; (c) frequenze per tutte le possibili configurazioni; (d) numero di occorrenze a diversi salti di frequenza.

malmente vengono definite nella sequenza di configurazione dopo l'accensione. Le porte di output possono essere connesse impiegando una qualsiasi combinazione dei 4 processori adiacenti e questo setup è modificabile in ogni momento via software. Le porte di input vengono lette (e quelle di output, scritte) utilizzando variabili dedicate e la temporizzazione inter-processore è di fatto invisibile ai programmi che non sono dotati di esplicita sincronizzazione software. Le connessioni di tipo *nearest neighbor* dell'AsAP risultano quindi essere linee ad alta velocità locali (cioè non globali) di lunghezza inferiore alla dimensione lineare di un processore e, di conseguenza, la comunicazione tra un processore e uno adiacente presenta ritardi molto piccoli e permette alle frequenze dei clock di scalare verso l'alto senza alcun problema. Dato che i collegamenti fisici sono presenti solamente tra processori adiacenti, il trasferimento di dati da un processore a uno non adiacente avviene mediante tecniche di instradamento (routing) che sfruttano i processori intermedi. E' possibile quindi collegare virtualmente una qualsiasi coppia.

L'affidabilità del trasferimento dei dati attraverso domini di clock incorrelati è garantita da strutture FIFO di tipo *mixed-clock-domain*, studiate per gestire questo tipo di comunicazioni. Le *dual-clock* FIFO leggono e scrivono dati in domini di clock completamente indipendenti (e arrestabili), senza alcuna restrizione oltre al rispetto del numero minimo di cicli per effettuare una lettura/scrittura. Le strutture FIFO sono state create interamente mediante sintesi, e disposte automaticamente.

Un diagramma a blocchi dei principali componenti di una FIFO è mostrato in figura 8.13. In AsAP, la *dual-clock* FIFO risiede nel processore dal quale provengono i dati, perché ciò semplifica il design. I valori vengono letti solamente dalla testa della

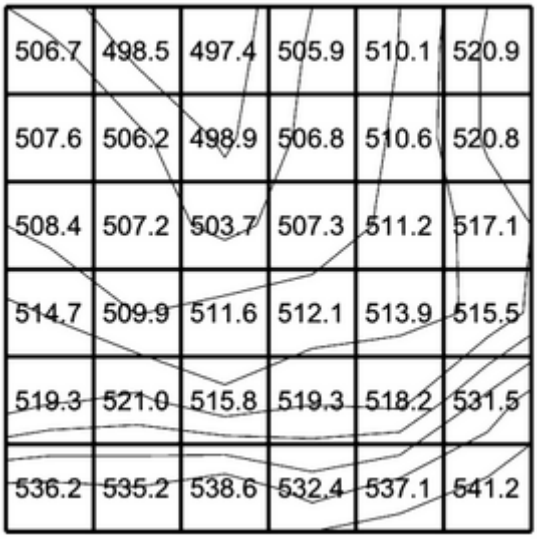


Figura 8.11 – Distribuzione fisica delle frequenze dell'oscillatore misurate tra processori differenti con la stessa configurazione. I dati sono espressi in MHz, con linee di contorno tra $500MHz$ e $540MHz$, con scatti di $5MHz$.

FIFO e ciò si traduce, nel software, in un singolo operando sorgente. Gli indirizzi *read* e *write* sono trasferiti attraverso l'interfaccia asincrona e sono utilizzati per decidere se la FIFO sia piena o vuota. Registri di sincronizzazione configurabili sono inseriti nell'interfaccia asincrona per evitare metastabilità. Inoltre, gli indirizzi sono codificati (con codice gray) quando vengono trasferiti tra diversi domini [101].

8.3.3. Implementazione dell'AsAP

Il processore AsAP è stato implementato con tecnologia TSMC standard CMOS a $0.18\mu m$ utilizzando celle standard Artisan. Il chip è stato interamente sintetizzato da una descrizione in Verilog, eccezion fatta per gli oscillatori che invece sono stati progettati a mano utilizzando celle standard. Le IMem, DMem e altre due memorie FIFO sono state costruite utilizzando macro-blocchi di memoria.

Il processo di design è stato condotto in due fasi: (1) un singolo processore è stato posizionato e collegato automaticamente utilizzando tecniche per il design *power-grid* [46, 47], le tecniche di inserzione di clock in strutture ad albero e diversi passaggi per l'ottimizzazione *in loco* per aumentarne la velocità e ridurre la congestione delle linee. (2) I processori sono stati disposti lungo il chip e il piccolo quantitativo di circuiteria globale è stato posizionato automaticamente intorno alla matrice.

8.4. Test e misurazioni

8.4.1. Area

Per via delle ridotte dimensioni delle memorie e del semplice schema di comunicazione, ciascun processore AsAP dedica gran parte della propria area all'esecuzione e quindi raggiunge in questi termini una notevole efficienza.

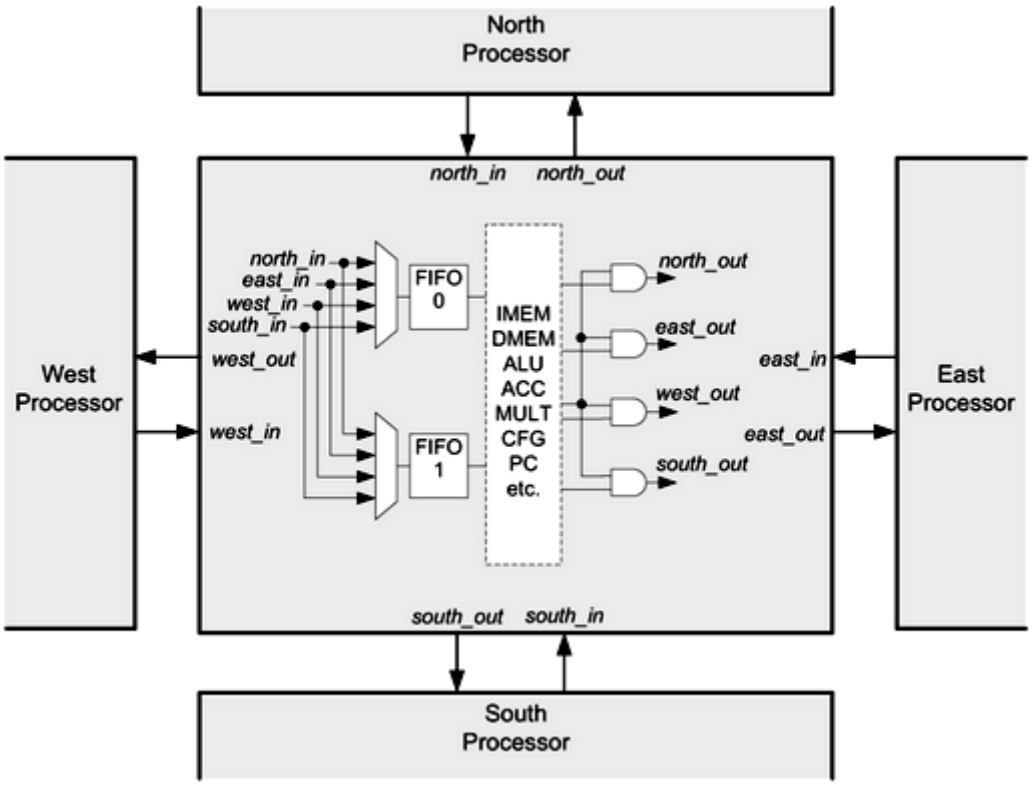


Figura 8.12 – Diagramma di comunicazione inter-processore tra primi vicini.

La tabella 8.2 mostra il dettaglio dell'area per ciascuno dei processori dell'AsAP. L'8% è dedicato ai circuiti di comunicazione, il 26% alla memoria e il 66% al nucleo del processore (dato estremamente rilevante). La bontà di questi dati è evidente dal confronto con altri processori, come mostrato in figura 8.14(a), poiché gli altri processori usano solo tra il 20% e il 45% della propria area per il nucleo [102, 1, 78, 7, 25, 98].

Ciascun processore AsAP occupa $0.66mm^2$ e la matrice 6×6 occupa $32.1mm^2$ includendo gli spaziatori, gli anelli globali di alimentazione e un piccolo contributo dovuto a circuiteria a livello chip. La figura 8.14(b) confronta l'area di alcuni processori scalati a tecnologia $0.13\mu m$, assumendo che l'area si riduca proporzionalmente col quadrato della riduzione nella misura della tecnologia. Il processore AsAP è dalle 20 alle 210 volte più piccolo rispetto a questi altri processori.

8.4.2. Consumo e prestazioni

Il processore realizzato lavora a circa $520 - 540MHz$, alimentato da una tensione di $1.8V$, e a oltre $600MHz$ se alimentato con una tensione di $2.0V$. Dato che i processori AsAP non dissipano potenza attiva quando sono in stato di inattività (idle) anche solo per brevi istanti, e poiché istruzioni differenti dissipano quantità di potenza alquanto diverse, è utile considerare diverse misurazioni per la potenza. La potenza media consumata per processore con applicazioni JPEG *encoder* e trasmettitore LAN wireless 802.11a/g, è di $32mW$ a $475MHz$. I processori che sono attivi al 100% mentre eseguono un insieme tipico di istruzioni, consumano ciascuno $84mW$ a $475MHz$. Il

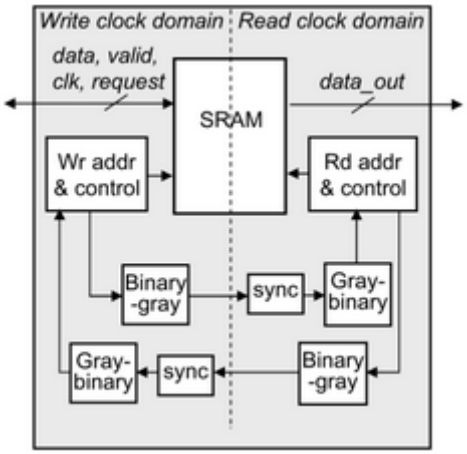


Figura 8.13 – Diagramma di una dual-clock FIFO usata per la comunicazione asincrona su frontiera.

	Area (μm^2)	**Area (%)**
Unità funzionali	433'300	66.0%
Data Memory	115'000	17.5%
Instruction Memory	56'500	8.6%
Two FIFOs	48'000	7.4%
Oscillatore	3'300	0.5%
Singolo processore	656'100	100.0%

Tabella 8.2 – Dettaglio dell'area in un singolo processore.

caso peggiore in assoluto per processore è stato rilevato in $144mW$ a $475MHz$ e si è verificato usando istruzioni MAC con tutte le memorie attive in ogni ciclo.

Alimentati con una tensione di $0.9V$, i processori girano a $116MHz$ e il consumo di potenza per una applicazione tipica è di soli $2.4mW$.

Complessivamente, circa i $2/3$ del consumo dell'AsAP sono imputabili al sistema di clock. Ciò è dovuto al fatto che non è implementato il meccanismo di *clock-gating* (in questo primo design) e quindi si presume che implementazioni future, dotate di *clock-gating* (che, ricordiamo, è distinto dal meccanismo di arresto dei clock) anche di basso livello, possano risparmiare un ulteriore, significativo, quantitativo di potenza.

Anche in questo caso, se paragonato a diversi altri processori, l'AsAP vanta prestazioni tra le 7 e le 30 volte maggiori rispetto agli altri (in particolare il consumo per operazione è dalle 5 alle 15 volte più piccolo).

Se il design AsAP qui presentato fosse scalato fino ai $90nm$, un chip delle dimensioni di $13mm \times 13mm$ riuscirebbe a contenere 1000 processori, con un picco computazionale di 1 TeraOp/s, e consumando solamente $10W$ in applicazioni tipiche (in aggiunta al consumo dovuto al *leakage*).

La struttura AsAP può fornire opportunità per ridurre le perdite (*leakage*) e il consumo di potenza attiva per i casi tipici quando i carichi di lavoro sono sbilanciati rispetto alla struttura dei processori e inoltre è in grado di sfruttare tecniche già adottate in processori molto più grandi, come ad esempio la riduzione della tensione di alimentazione nelle zone a bassa densità di processori attivi. Nonostante questo

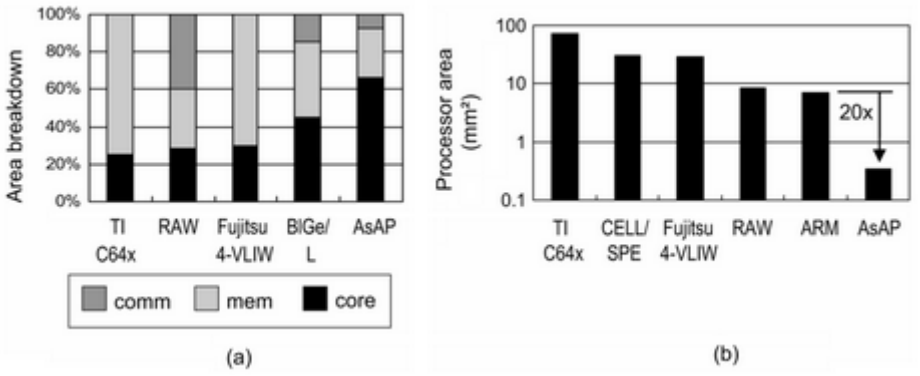


Figura 8.14 – Calcolo dell'area di un processore AsAP e diversi altri con tecnologia scalata su $0.13\mu m$.

design non contenga circuiti speciali o funzionalità architetturali per ridurre il *leakage*, assumendo una densità di transistor costante e tecnologia di implementazione di tipo *deep-submicron*, il consumo (di leakage) dell'AsAP scala circa come l'area, cioè è tra le 20 e le 210 volte inferiore rispetto ai processori mostrati in figura 8.14.

8.4.3. Software e implementazioni

I precedenti risultati sono utili, ma la maggior parte è basata su metriche semplici e non tiene conto delle prestazioni specifiche per una data applicazione. In aggiunta ai task elencati in tabella 8.1 [125], sono state implementate e analizzate anche alcune particolari applicazioni, come un encoder JPEG e la parte in banda base di un trasmettitore LAN wireless 802.11a/b, entrambe scritte a mano in codice assembly e parallelizzate in modo che potessero sfruttare la matrice di processori. Concludiamo il capitolo presentando alcuni risultati per queste due applicazioni.

Encoder JPEG

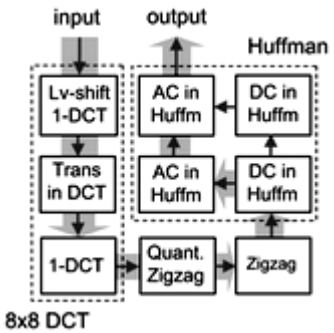


Figura 8.15 – Nucleo di un encoder JPEG con 9 processori: frecce sottili indicano tutti i percorsi mentre quelle spesse indicano il flusso primario dei dati.

La figura 8.15 mostra un encoder JPEG (nucleo) che utilizza 9 processori. 3 processori calcolano il *Level Shift* e una 8×8 DCT[(4)], mentre 4 processori implementa-

[(4)]DCT: Discrete Cosine Transform.

no un Huffman encoder. Insieme, consumano $224mW$ a una frequenza di $300MHz$. Processare tutti gli 8×8 blocchi richiede circa 1400 cicli di clock.

Paragonato un'implementazione su un processore TI C62x 8-way VLIW DSP, AsAP registra prestazioni simili, ma consuma circa 11 volte di meno [112, 16].

Trasmettitore 802.11a/g

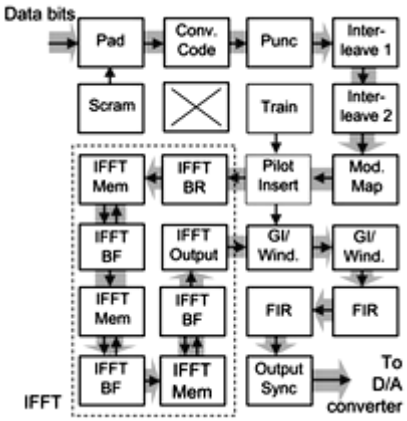


Figura 8.16 – Trasmettitore 802.11a/g che utilizza 22 processori: frecce sottili indicano tutti i percorsi, mentre quelle spesse indicano il flusso primario dei dati.

La figura 8.16 mostra la parte in bada base di un trasmettitore wireless, full-compliant IEEE 802.11a/g che utilizza 22 processori [70]. I dati entrano nel trasmettitore nella parte alta a sinistra, scorrono attraverso un certo numero di task nel dominio della frequenza, passano nella OFDM IFFT complessa a 64-punti, percorrono un certo numero di task nel dominio del tempo e alla fine entrano in un processore di sincronizzazione il cui solo scopo è quello di abilitare la connessione tra l'array asincrono e un convertitore D/A (Digitale/Analogico) senza alcuna altra logica necessaria.

	TI C62x		*AsAP*	
	Prestazioni	**Energia**	**Prestazioni**	**Energia**
JPEG encoder	$3.90\mu sec/blk$	$12.4\mu J/blk$	$4.65\mu sec/blk$	$1.04\mu J/blk$
IEEE 802.11a/g tx	$1.7Mbps$	$1880nJ/bit$	$16.2Mbps$	$25.1nJ/bit$

Tabella 8.3 – Confronto di JPEG encoder (9 processori, AsAP) e trasmettitore IEEE 802.11a/g (22 processori, AsAP) con implementazioni su TI C62x VLIW DSP.

L'implementazione consuma $407mW$ a $300MHz$ e raggiunge i $54Mb/s$ al 30% della velocità massima, nonostante il fatto che il codice sia poco ottimizzato e non schedulato. Questi risultati dimostrano la bontà di questa implementazione, se comparati con una precedente implementazione (incompleta) su un TI C62x 8-way VLIW DSP [80, 16].

AsAP raggiunge prestazioni tra le 5 e le 10 volte superiori consumando tra le 35 e le 75 volte meno potenza, a seconda dei dettagli dell'implementazione. I dati sono riassunti in tabella 8.3.

9. BitSNAP

In questo capitolo presentiamo BitSNAP, un processore asincrono pensato per lavorare in una rete di sensori, che sfrutta datapath di tipo *bit-serial* e realizza un'efficiente versione della *dynamic significance data compression* (la spiegazione più avanti nel capitolo).

9.1. Introduzione

La maggior parte delle piattaforme per reti di sensori basate su micro-controller COTS (*Commodity-Off-The-Shelf*) consumano enormi quantitativi di energia ed esauriscono le batterie in tempi che vanno da qualche ora a pochi giorni. In applicazioni di questo tipo, poter disporre di processori che sfruttino in modo efficiente le risorse di energia è essenziale se si vuole acquisire un'autonomia maggiore.

Recentemente, è stato proposto un innovativo *Instruction Set Architecture* (ISA) asincrono, progettato espressamente per reti di sensori, chiamato SNAP (Sensor Network Asynchronous Processor) ISA. SNAP è stato progettato per due scopi principali: (1) per funzionare come processore all'interno di un chip multi-processore per un simulatore hardware di una rete di sensori, chiamato *Network-On-a-Chip* (NoC) [15] e, (2) per essere impiegato come processore in un nodo a basso consumo di una rete di sensori wireless chiamata SNAP/LE (*SNAP Low Energy*) [115].

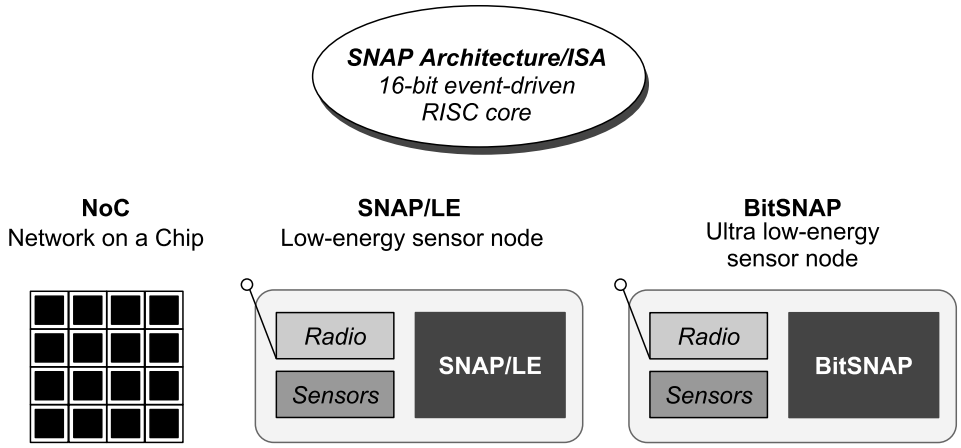


Figura 9.1 – Famiglia di processori SNAP.

Le varianti del processore sono mostrate in figura 9.1.

Il processore SNAP/LE contiene un nucleo RISC 16 bit di tipo *event-driven* (pilotato da eventi) e può operare entro un ampio range di tensioni di alimentazione, eseguendo istruzioni da un tasso di $23MIPS$ (a $0.6V$) fino a $200MIPS$ (a $1.8V$) e consumando solamente $24pJ$ per istruzione: un consumo inferiore per diversi ordini di grandezza rispetto a qualunque micro-controllore COTS.

Per raggiungere un così ridotto consumo di potenza dinamica pur mantenendo un throughput relativamente alto, lo SNAP/LE si basa su un semplice ISA, su un modello di esecuzione di tipo event-driven, su hardware specializzato per le reti di sensori e su tecniche per circuiti asincroni a basso consumo. Con livelli di attività estremamente bassi, come quelli riscontrati nelle applicazioni tipiche per una rete di sensori (10 eventi al secondo o meno), SNAP/LE consuma dai $16nW$ ai $58nW$ alimentato a $0.6V$.

Un'osservazione chiave, derivante dai risultati prodotti dalle piattaforme di test per reti di sensori, è che necessitano di prestazioni relativamente basse: i nodi si limitano a estrarre informazioni dai sensori, processano e/o aggregano i dati in modo molto rudimentale e infine spediscono il tutto, attraverso collegamenti radio abbastanza lenti (nell'ordine dei Kbit/s), a delle stazioni di calcolo molto più potenti [35].

La maggior parte delle reti di sensori attuali impiega processori che operano tra i 4 e i $12MIPS$ [35, 33], mentre SNAP/LE, anche se ai limiti del *voltage-scaling* e alimentato con una tensione vicina alla tensione di soglia dei transistor, opera ben oltre i $20MIPS$, che è molto più di quanto necessario.

Nel resto del capitolo, illustreremo un nuovo processore, BitSNAP [117], che sacrifica una parte delle (comunque abbondanti) prestazioni per ottenere un ulteriore guadagno nei consumi. BitSNAP è la logica estensione di SNAP/LE, progettato per sfruttare il comportamento tipico di una rete di sensori. Ciò deriva dalla naturale compressibilità dei valori degli operandi che si propagano nel datapath del processore: sfruttando questa caratteristica si riducono le commutazioni dei circuiti, risparmiando potenza.

Mentre SNAP/LE è stato progettato con un datapath parallelo a 16 bit, BitSNAP utilizza un bit-serial datapath (con il quale il design asincrono si accoppia favorevolmente) per calcolare le lunghezze degli operandi in modo efficiente. L'uso di un bit-serial datapath inoltre riduce la quantità dell'hardware impiegato nel circuito e, di conseguenza, vi è una riduzione della dispersione di potenza statica, argomento questo molto importante e i cui effetti riguardano in particolare i processi di tipo *deep sub-micron*.

9.2. Significance compression

9.2.1. Architetture parallel-word

Studi architetturali precedenti [96, 73] hanno mostrato che la maggior parte dei valori degli operandi nei datapath necessitano di un numero di bit significativi inferiore di quello fornito da un datapath a 16, 32 o 64 bit. Comprimendo in modo dinamico il valore di un dato (sopprimendo gli zeri o uni iniziali) e trasmettendo solo i bit necessari, si può risparmiare dal 30 all'80% dell'energia impiegata per le commutazioni dei circuiti (*switch*) che avvengono normalmente in un datapath.

Precedenti lavori di Brooks e Martonosi nell'ambito dei circuiti sincroni, sono focalizzati su approcci di *clock-gating* delle unità funzionali in base agli operandi [23]; a ogni ciclo, viene eseguito un controllo sugli operandi per individuare zeri iniziali e un appropriato segmento di unità funzionale viene spento. Naturalmente, questo approccio introduce una latenza e un consumo energetico aggiuntivi, per via del circuito dedicato all'individuazione degli zeri. Canal et al. in [96] hanno presentato un

approccio in cui ciascun operando viene marcato tramite bit aggiuntivi che indicano quali byte della parola di codice siano compressibili. I datapath di tipo *bit-parallel* e *bit-serial* studiati in questi lavori subiscono pesantemente l'impatto del controllo per la propagazione verticale del dato e dei meccanismi di bypass per i processi multi-ciclo seriali.

Al contrario, precedenti lavori sulla "*compressione dinamica della significance*"[(1)], nel contesto dei circuiti asincroni, si sono focalizzati su architetture chiamate *Width-Adaptive Data* (WAD) dove la rappresentazione numerica è di per sè limitata. I numeri WAD come furono proposti da Manohar [73] utilizzano una speciale rappresentazione numerica che impiega le cifre $\{0, 1, \underline{0}, \underline{1}\}$. Le cifre $\underline{0}$ e $\underline{1}$ rappresentano delimitatori che terminano una data parola di codice binario: ogni bit che sia più significativo del bit delimitatore assume il suo stesso valore. Per fare un esempio, il numero WAD $\underline{1}011$ rappresenta, per un datapath a 8 bit, il numero $1111\,1011$ in normale rappresentazione binaria. Ciò ci permette di comprimere un numero con zeri/uni iniziali utilizzando una rappresentazione più compatta che ben si adatta al controllo verticale su pipeline nel datapath.

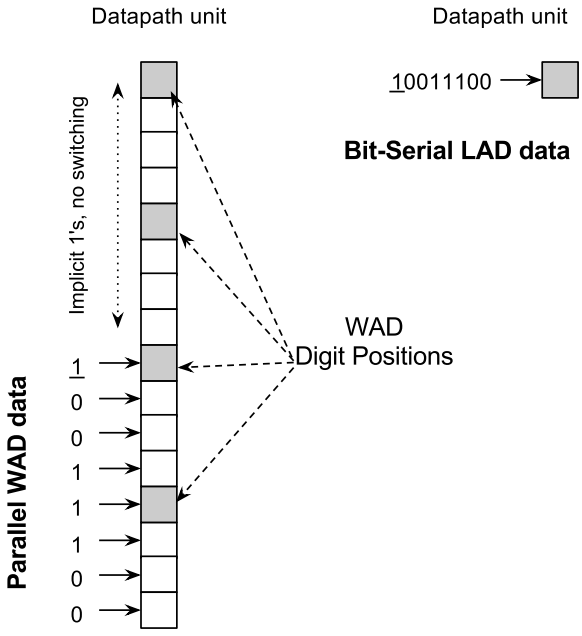


Figura 9.2 – (sx) Width Adaptive Datapath (WAD) a 16 bit con una cifra WAD ogni 4 bit. (dx) Length Adaptive Datapath (LAD) a 16 bit con una cifra LAD ogni bit.

Per ragioni di efficienza, nelle architetture è necessario limitare l'uso delle cifre WAD in base ai blocchi (fig. 9.2). Un simile schema ibrido, dove alcuni bit sono normali e altri sono di tipo WAD, è stato usato nel register file del QDI 32 bit presentato in [24]. In quel caso, ciascun blocco consiste di 3 bit normali più un bit WAD nella posizione più significativa (dei 4 bit del blocco) e quindi fornisce una compressione dei dati con granularità a blocchi di 4 bit. D'altro canto, utilizzare uno schema ibrido

[(1)]Dynamic Significance Compression: Il termine *significance* in questo contesto rappresenta il numero di bit che rimangono, in una parola di codice binaria, dopo la compressione degli zeri/uni iniziali. Per una parola quindi, quanti più sono i bit significativi (dopo la compressione), tanto più è alta la sua *significance*. Di conseguenza, abbiamo scelto di non tradurre il termine *significance*: in questo contesto la traduzione altera il significato.

complica il processing dei dati, perché il progettista deve disporre i bit WAD oltre ai bit normali.

9.2.2. Architetture bit-serial

Al contrario delle architetture di tipo *parallel-word*, BitSNAP utilizza la compressione dinamica della significance su un flusso dati di tipo bit-serial. In altre parole, BitSNAP è un processore di tipo *Length-Adaptive Datapath* (LAD) e realizza la compressione dei dati con granularità di un singolo bit, come mostrato in figura 9.2. Invece di spedire tutti i 16 bit attraverso il datapath, per ogni parola vengono spediti solo i bit significativi, incluso il delimitatore. Un evidente vantaggio rispetto all'altro approccio (oltre al ridotto numero di commutazioni), è che il throughput relativo alle operazioni su *data words* non risulta più strettamente lineare nella lunghezza dei dati (es. 16 volte il ciclo per un bit individuale in un'architettura a 16 bit) poiché gli operandi compressi possono essere completati in un tempo proporzionale alla porzione di bit significativi, che varia da operando a operando. Un ulteriore vantaggio per il progettista è che, poiché ciascun bit è nella rappresentazione LAD, le unità nel datapath non devono gestire la differenza tra bit normali e bit speciali.

Handler	**Function**
Packet TX	Trasmissione di pacchetti radio via 802.11 basato su MAC layer
Packet RX	Ricezione pacchetti radio
AODV RR	Ricerca percorso e risposta del protocollo di routing ad-hoc
AODV PF	Invio pacchetti del protocollo di routing ad-hoc
TR1000 Radio Stack	Codifica e CRC dei pacchetti dati
Data logger	Tracciamento sensori e calcolo medie a regime

Tabella 9.1 – Carico di lavoro del software per la rete di sensori BitSNAP.

Le architetture di tipo bit-serial sincrone sono state esplorate a fondo, dal *bit-serial signal processing* [89] fino al design di compilatori come il Parsifal della GE [100]. In ogni caso, anche la più semplice architettura richiede clock multipli: uno per l'unità di calcolo sui bit e un altro per il calcolo dei confini sulle parole di bit. Nel caso della dynamic significance compression, dove il confine di una parola può essere in qualsiasi posizione, gli schemi di clock e bypass diventano rapidamente ingestibili anche per parole di soli 16 bit. Questo problema può essere ridotto attraverso l'uso di bit speciali, adibiti alla segnalazione della posizione di fine parola, ma presenta un costo aggiuntivo in termini energetici.

Per portare un esempio del potenziale risparmio di energia relativo al carico di lavoro dei processori, e specialmente nel contesto dei processori per la rete di sensori BitSNAP, è stato eseguito uno studio sulla significance degli operandi, nei datapath dei processori di una rete gravata da un tipico carico di lavoro (mostrato in tabella 9.1). La figura 9.3 illustra invece la distribuzione media della significance degli operandi rispetto ai vari carichi di lavoro. Da notare che il 90% dei dati che fluiscono nel

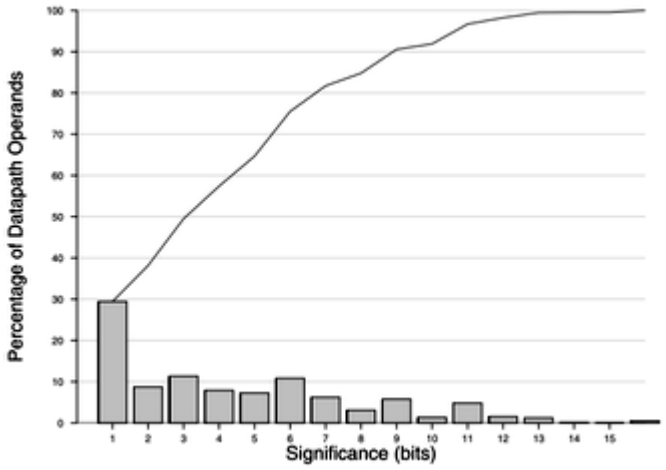


Figura 9.3 – Stima dei valori medi della *significance* degli operandi rispetto a un carico di lavoro tipico per una rete di sensori.

datapath del processore presenta una significance inferiore a 10 bit e che circa il 30% può essere compresso addirittura sino a un singolo bit.

9.2.3. Rappresentazione Length-Adaptive Data

Per la codifica delle cifre LAD nelle pipeline Delay-Insensitive sono state considerate due opzioni:

1. Per ogni bit, utilizzare un codice di tipo 1-di-2 (due bit con diversi significati) dove uno esprima il dato e l'altro indichi se è un delimitatore o no. Quindi ad esempio, "00" indicherebbe uno 0 mentre "10" indicherebbe uno $\underline{0}$.

2. Per ogni bit, utilizzare 4 linee, ciascuna indicante uno dei 4 possibili codici WAD $0, 1, \underline{0}, \underline{1}$.

Nonostante la prima sia una buona opzione per datapath WAD ibridi (come quelli usati nei register file WAD) poiché mantiene l'informazione dei dati distinta dall'informazione dei delimitatori, è preferibile il secondo approccio per i bit-serial datapath. La ragione è che, poiché solo una della 4 linee commuta quando viene trasmesso un bit, con questo sistema viene speso il minimo quantitativo di energia per trasmettere dati LAD.

9.3. Architettura del processore BitSNAP

In questa sezione forniamo un riassunto dello SNAP ISA e delle decisioni architetturali prese durante l'implementazione delle caratteristiche LAD bit-serial di BitSNAP.

9.3.1. Nucleo dell'architettura

Un diagramma a blocchi di alto livello del processore BitSNAP è illustrato in figura 9.4. BitSNAP implementa un nucleo di tipo *single-issue* (si veda la nota a pagina 93),

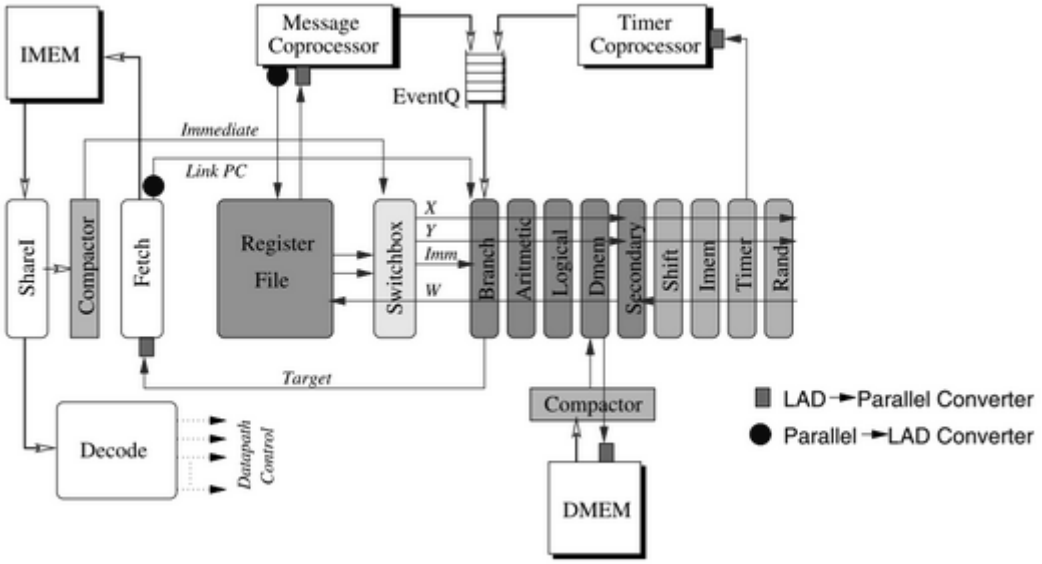


Figura 9.4 – Diagramma a blocchi di alto livello del BitSNAP.

in-order, privo di speculazione. L'ISA dello SNAP è basato sul design MIPS e specifica una parola *architetturale* a 16 bit con 16 registri. Le istruzioni possono consistere di una o due parole (queste ultime perché richiedono un valore immediato a 16 bit). Le istruzioni e la memoria dati (DMEM in figura) consistono di 2 banchi da $4KB$, con una memoria istruzioni (IMEM in figura), la cui scrittura è permessa all'utente, che consente di eseguire il boot-strap del processore con programmi esterni forniti in fase di accensione.

La differenza fondamentale tra l'architettura SNAP e altri processori convenzionali è che SNAP è completamente pilotata da eventi (*event-driven*): SNAP rimane in stato di inattività (idle), privo di commutazioni dinamiche, finché non si verifica un evento, o interno (dovuto a un timer che scade e lo scatena), o esterno (un sensore che legge dei dati, un messaggio che arriva via radio, etc.). A questo punto, un token[(2)] che corrisponde all'evento viene propagato attraverso una coda di eventi hardware finendo per risvegliare il processore mediante l'unità di controllo *PC Control*.

In questa unità, il token relativo all'evento viene impiegato come indice in una tabella di eventi hardware, per recuperare l'indirizzo di una precisa porzione di codice atta alla gestione di quel tipo di evento (*event handler*). L'indirizzo dell'event handler viene a sua volta passato al *Fetch*, e a questo punto il processore può cominciare a eseguire il codice. Una volta completata l'esecuzione, l'event handler emette un'istruzione *DONE* che arresta il processore finché un altro evento non si presenterà in coda e ricomincerà il tutto.

Gli eventi radio sono generati dal *message coprocessor*, il quale comunica con un chip radio esterno. Il message coprocessor inoltre genera eventi che corrispondono a interrupt da sensori esterni. Tutta la comunicazione tra il message coprocessor e il radio chip o i sensori, è gestita con un sistema di I/O *general-purpose*, *register-mapped* e quindi non richiede nuove istruzioni. Per maggiori dettagli sull'ISA dello SNAP e sull'implementazione del coprocessore, si veda [115].

[(2)]Token: Un'istruzione, assieme ai suoi operandi (quando necessario), viene tramessa a un'unità di processo in un pacchetto che viene chiamato *instruction token*.

BitSNAP utilizza per il bus una struttura a due livelli per sfruttare al meglio la flessibilità fornita dal design asincrono. C'è un bus principale che si occupa delle unità sul datapath che vengono utilizzate più spesso: l'unità aritmetica, logica, di branch e quella relativa al data memory. Un bus secondario invece viene collegato a quello principale come unità aggiuntiva e contiene l'unità di shift, quella per l'accesso all'instruction memory e al coprocessore *timer*, e quella che genera i numeri pseudo-casuali. Nella progettazione di BitSNAP, sono state utilizzate tecniche QDI che sono un ibrido tra gli stili di design utilizzati per il MiniMIPS [10] e quelli utilizzati per il microprocessore Caltech [11].

Mentre il microprocessore Caltech utilizza datapath allineati, progettati con tecniche di *control/data decomposition* e il MiniMIPS utilizza stadi pipeline con *pipeline completion*, BitSNAP alterna l'uso di entrambi gli stili in base all'implementazione realizzata per ciascun blocco hardware.

9.3.2. Supporto per il bit-serial

L'architettura discussa fin qui ha molto in comune con lo SNAP/LE, quindi ora vediamo le funzionalità aggiuntive che la rendono innovativa e che le permettono di realizzare la significance data compression grazie alla conversione dal datapath di tipo bit-parallel a quello bit-serial.

I blocchi ombreggiati in figura 9.4 rappresentano le parti del datapath che processano dati di tipo bit-serial. Si noti che l'intero register file, tutte le unità funzionali e ogni *split* e *merge* nel bus del datapath, sono implementati come unità bit-serial che operano su cifre LAD.

Le linee sottili rappresentano bus bit-serial che trasportano dati LAD. Poiché lo sforzo è stato concentrato nella progettazione del nucleo principale del processore, l'architettura utilizza lo stesso timer bit-parallel e message coprocessor del processore SNAP/LE ma, comunque, è sicuramente possibile prevedere l'implementazione di questi ultimi in modo che sfruttino dei datapath LAD, in futuro.

Le memorie operano anch'esse su parole a 16 bit di tipo *parallel-words*, per ragioni di densità ed efficienza. L'unità di fetch, poiché opera in ogni ciclo e si interfaccia direttamente con l'instruction memory, è anch'essa implementata utilizzando circuiti bit-parallel, sempre per ragioni di efficienza.

Se così non fosse infatti, il program counter (PC) dovrebbe passare per la conversione tra i due formati in ogni ciclo, sprecando energia per eseguire anche solo delle semplici istruzioni di incremento (del PC) che non interagiscono con il datapath.

Allo stesso modo, il processo *Sharel*, che condivide la parola in ingresso o con il decoder (per una *instruction word*), o con il datapath (per operandi immediati), utilizza circuiti di tipo bit-parallel.

La conclusione è che, per non dover affrontare la completa conversione dell'architettura da bit-parallel a bit-serial, la soluzione ibrida qui proposta lascia inalterate quelle parti che richiederebbero l'introduzione di meccanismi di conversione troppo onerosi in termini di consumo energetico per via della loro frequenza di utilizzo.

Per una descrizione dettagliata di alcune delle unità funzionali del BitSNAP (e del codice CHP[3] relativo), si veda [117].

[3] Per una completa descrizione della notazione CHP e della sua semantica, si veda [75].

9.4. Gestione dei Length-Adaptive-Data

In questa sezione descriveremo il funzionamento dell'hardware speciale che viene impiegato da BitSNAP per gestire l'interfacciamento (attraverso la conversione del formato dei dati) tra le unità funzionali LAD e i componenti di tipo bit-parallel. Vedremo anche alcuni circuiti speciali che servono per ottimizzare la gestione dei dati LAD. L'efficienza di BitSNAP proviene dalla sua capacità di mantenere gli operandi alla massima compressione, risparmiando energia, come mostrato in figura 9.3 (ad esempio, gli operandi che provengono dalla memoria, necessitano di essere compressi prima di poter entrare nel datapath).

Un aspetto di cui tenere conto è che l'hardware utilizzato per implementare il processore BitSNAP non è perfetto, nel senso che non produce sempre la rappresentazione ottimale: per chiarire questo concetto con un esempio, basta eseguire la somma dei due operandi $\underline{0}111$ e $\underline{1}000$. Il risultato è $\underline{1}111$, che chiaramente non è ottimale. Idealmente, il datapath dovrebbe comprimere il risultato in $\underline{1}$, ma applicare la compressione massima su ogni operando in questo modo, introdurrebbe elevati costi in termini di consumo e throughput. Vedremo che è possibile, con l'uso di alcune pratiche tecniche di compressione, alleviare in parte questo fenomeno indesiderato.

9.4.1. Conversione LAD/bit-parallel

I punti di frontiera tra blocchi LAD e bit-parallel (fig. 9.4) richiedono dei blocchi per la conversione da una struttura all'altra. In questa sezione discutiamo i due convertitori principali: da LAD a bit-parallel e il blocco *compactor* per la conversione da bit-parallel a LAD.

Conversione LAD $\rightarrow$ bit-parallel

Quando spediamo un indirizzo alla data memory, oppure un valore al message coprocessor, il bus impiegato nella comunicazione deve convertire il dato in questione da flusso dati LAD (che è di lunghezza variabile) a un valore a 16 bit. Per eseguire questo compito, viene impiegato un contatore, in accoppiata con un insieme di 16 registri da 1 bit, controllati da un demultiplexer pilotato dal contatore stesso. Il flusso dati LAD viene scritto nei registri in modo sequenziale; quando arriva un delimitatore, il bit LAD rimane inalterato finché il contatore non arriva al termine del conteggio. In questo modo, se entra un $\underline{1}$, tutti i registri che trattengono bit più significativi di quella posizione, vengono riempiti con il valore 1, fino al termine del conteggio.

Blocco *compactor* per conversione bit-parallel$\rightarrow$LAD

BitSNAP utilizza questo tipo di conversione principalmente sulle parole che provengono dalla memoria (*immediate* e *load values*). Si noti che non vi è alcuna informazione, riguardo alla significance di un dato, registrata in memoria; le parole a 16 bit possiedono tutte il massimo valore di significance, 16. Di conseguenza, è necessario comprimere questi dati eliminando gli zeri/uni iniziali.

La figura 9.5 mostra gli effetti di diversi metodi di compressione volti a raggiungere il massimo livello di compattezza. La linea *maximum compression* (equivalente a quella in figura 9.3) mostra il massimo livello di compressione cui ciascun operando può arrivare (cioè la sua reale significance), e rappresenta il massimo che ciascun

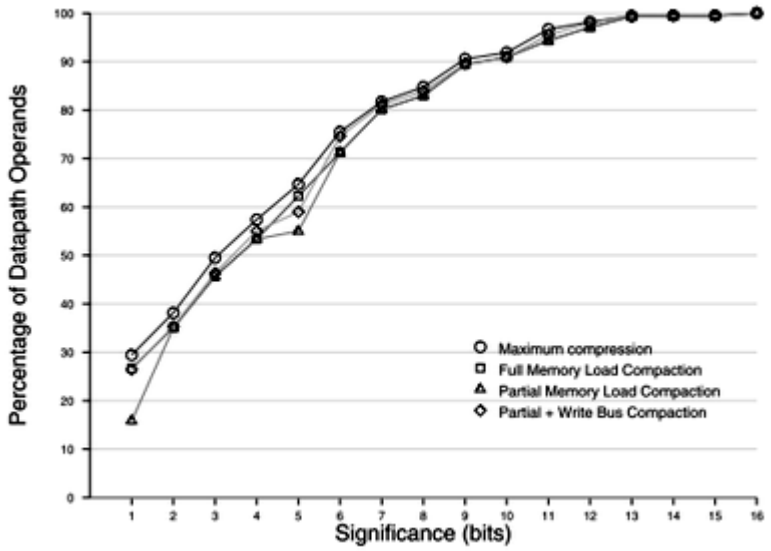


Figura 9.5 – Significance degli operandi, utilizzando opzioni di compressione differenti.

diverso metodo può raggiungere. La linea *full memory load compaction* mostra cosa succede alla significance dell'operando se operiamo una compressione massima (*full*) sui dati in uscita dalla memoria. Si può raggiungere un guadagno del 5% per operandi fino a 8 bit e dell'1% per operandi con un numero di bit maggiore (effetti collaterali, come la decompressione dovuta a operazioni come l'addizione o la sottrazione ci impediscono di raggiungere il massimo).

Sfortunatamente, un circuito che realizza la compressione massima è dispendioso in termini di energia e throughput [73]. In base alle simulazioni eseguite, la scelta è ricaduta sul seguente schema, che fornisce buoni risultati utilizzando circuiti poco dispendiosi: per ogni parola, si cerca di comprimere solamente i 12 bit più significativi. Si controlla che (1) gli 8 bit più significativi siano o tutti 1, o tutti 0, mentre in parallelo (2), si fa lo stesso con i 4 bit meno significativi. Se (3) entrambi i test risultano verificati e se i due blocchi compressi presentano lo stesso valore (es: $\underline{0}, \underline{0}$ o $\underline{1}, \underline{1}$), i 12 bit vengono ridotti in un unico simbolo LAD, $\underline{0}$ o $\underline{1}$, altrimenti vengono compressi solamente gli 8 bit più significativi.

Questo metodo presenta il vantaggio che i 4 bit meno significativi del gruppo di 12 bit possono essere inviati in modalità bit-serial mentre viene eseguita la compressione degli altri 8. Oltretutto, tra lo stadio di compressione e il processo sul datapath che andrà a utilizzare il dato compresso, vengono inseriti diversi blocchi *one-step compaction* (descritti in dettaglio nella prossima sezione), ciascuno dei quali permette di guadagnare un ulteriore bit (es. $\underline{0}001 \rightarrow \underline{0}01$). L'inserimento di tre di questi blocchi costituisce il miglior compromesso tra massima compressione e latenza (per il trasporto). I risultati di questo metodo sono illustrati in figura 9.5 dalla linea *partial memory load compaction*, che si avvicina molto ai risultati relativi alla compressione massima eccezion fatta per un abbondante 10% nel caso di un singolo bit.

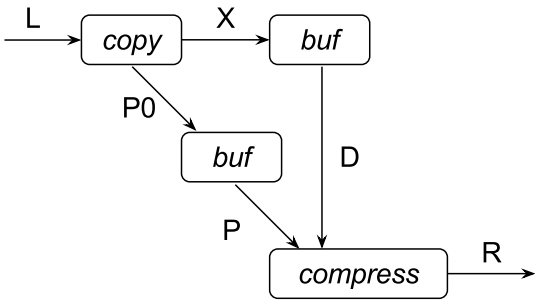


Figura 9.6 – Blocco *one-place compaction*.

9.4.2. One-Step Compaction

Questo processo viene utilizzato sia nella compressione (coi metodi illustrati nella sezione precedente) degli *immediate values*, che in quella dei *load values,* e anche sui dati che scorrono nel bus (in scrittura) che porta al register file, per rimediare al problema di decompressione che si verifica quando si eseguono operazioni su dati LAD.

Un diagramma dei processi coinvolti è illustrato in figura 9.6. L'input arriva sul canale L e l'output, compresso, esce dal canale R; i processi *copy* e *buf* sono dei semplici *copy* e *buffer* di tipo PCEVFB[4] mentre il processo *compress* è dato dal seguente codice in notazione CHP (D si assume inizializzato con un token nullo):

$$
\begin{array}{l}
*[D?d; P?p; \\
\quad [\hat{d} \rightarrow R!d \\
\quad []\neg\hat{d} \wedge \hat{p} \wedge p = d \rightarrow \textbf{skip} \\
\quad []\neg\hat{d} \wedge \hat{p} \wedge p \neq d \rightarrow R!d \\
\quad []d = null \rightarrow \textbf{skip} \\
\quad []\textbf{else} \rightarrow R!d \\
]]
\end{array}
$$

Figura 9.7 – Processo *compress* in notazione CHP.

L'efficacia di utilizzare un singolo stadio *one-step compaction* sul bus in scrittura (combinato con la compressione parziale della memoria, come descritto in precedenza), è illustrata in figura 9.5 dalla linea *Partial + Write Bus Compaction*.

Il livello di compressione raggiunto con questi accorgimenti è davvero molto vicino al massimo ottenibile, e i risultati sono addirittura migliori di quelli che si otterrebbero utilizzando solamente la compressione massima sui bus della memoria.

9.5. Valutazione

E' stato effettuato un confronto, in termini di prestazioni e consumo energetico, tra una comune implementazione con unità datapath LAD del BitSNAP e il bit-parallel SNAP/LE. Per quest'ultimo, per il quale si ha a disposizione un layout di datapath di qualità di produzione, si è usato Nanosim[5] per stimare le cifre relative alle presta-

[4]PCEVFB: Pre-Charge Enable-Valid Full-Buffer (si veda [24]).
[5]Nanosim: un veloce simulatore di circuiti prodotto dalla Synopsys, simile a Hspice in accuratezza.

zioni e al consumo energetico dalle netlist estratte (con effetti parassiti post-layout). Anche per BitSNAP è stato usato Nanosim, ma in questo caso le netlist sono state estratte in pre-layout e gli effetti parassiti sono stati aggiunti in seguito (in base a delle stime).

La netlist inoltre non è ottimizzata: la maggior parte delle porte utilizza una lunghezza fissa di 4λ per l'nfet e di 8λ per il pfet. Si è assunto inoltre un processo TSMC a $180nm$ utilizzando le regole SCMOS.

Structure	**SNAP/LE**		**BitSNAP**		
	Energy per word (pJ)	Total gate width μm	Energy per bit (pJ)	Freq. (MHz)	Total gate width μm
Bus Interface	8.1	473	Reader/Writer 0.43 1-step Comp. 0.71 Align 0.51	334 235 257	46 158 77.3
Arithmetic	11.7	1559	1.3	248	359
Logical	5.4	788	0.37	322	168
Register File	31.2	16.9k	Core (20-bits) 10.5 1-bit Ports 1.1	142 263	10.6k
Shifter	11.6	6k	1.4	238	515

Tabella 9.2 – Risultati per energia/prestazioni del confronto tra BitSNAP e SNAP/LE.

I risultati per le prestazioni e per il consumo di energia sono mostrati in tabella 9.2 per il BitSNAP e lo SNAP/LE, alimentati con una tensione di $1.8V$. In termini di consumo energetico, come previsto, il *bit processing LAD* è in qualche modo più costoso rispetto al consumo per bit del bit-parallel SNAP/LE, e ciò si deve all'impatto del circuito di controllo per ciascun ciclo. Per le interfacce bus si specificano i costi per: (1) *read*/*write bus* (la media dei due), (2) il processo di *one-step compaction* sul *write bus* e (3) il processo di allineamento usato per le operazioni che coinvolgono 2 operandi. Quindi, per una addizione, il consumo totale di energia dei bus risulta dalla somma di $0.43 \times 3pJ$ (2 operazioni *read*/*write*), $0.71pJ$ (*write bus compaction*) e $0.51pJ$ (allineamento), per un totale di $2.51pJ$. L'energia per l'operazione equivalente, consumata dallo SNAP/LE, è invece di $8.1 \times 3pJ$ per ciascuna operazione a 16 bit. Il consumo energetico del register file nel BitSNAP consiste di un costo fisso per accedere alla locazione del delimitatore a (16+4)-bit, mediato sui valori di lettura/scrittura, e di un costo per accedere a ciascun bit nel *1-bit input*/*output bus*. Si noti che, comunque, il costo fisso per accedere al register file di BitSNAP è notevolmente inferiore rispetto all'equivalente per SNAP/LE, poiché il throughput del register file dello SNAP/LE deve adeguarsi al throughput complessivo del processore, che è nell'ordine dei $200MHz$. Per BitSNAP invece, ciò che conta è il tempo di ciclo necessario per inserire o estrarre i bit dal bus, i cui cicli multipli possono *nascondere* il (più lento) tempo di ciclo della lettura/scrittura in parallelo verso il nucleo del registro.

In media, si è trovato che un bit LAD consuma circa il 10% in più dell'energia consumata da un bit per l'architettura parallela a 16 bit e quindi ogni operazione che può essere compressa fino a un massimo di 10 bit consumerà meno nel BitSNAP, mentre oltre i 10 bit, il consumo sarà minore nello SNAP/LE.

Per analizzare il risparmio effettivo rispetto allo SNAP/LE, sono state usate le stime energetiche prese dal simulatore dell'architettura BitSNAP e il risultato è che,

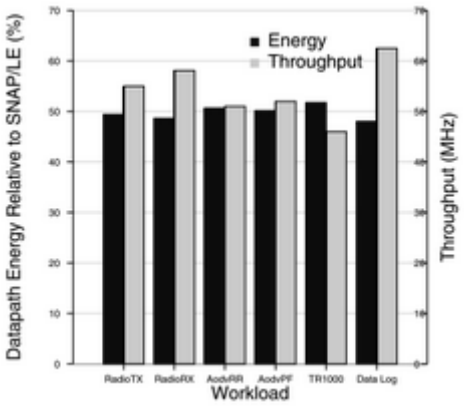


Figura 9.8 – BitSNAP: risparmio energetico e throughput per diversi carichi di lavoro (con tensione di alimentazione a $1.8V$).

con un carico di lavoro distribuito sulla rete di sensori come quello descritto in precedenza, il BitSNAP consuma circa il 48-52% di quanto consuma lo SNAP/LE, cioè circa la metà (figura 9.8).

Per quanto riguarda le prestazioni, il throughput della maggior parte delle unità del BitSNAP è maggiore rispetto a quello dello SNAP/LE, principalmente grazie alla mancanza di ampi *completion trees*. In un design convenzionale per un processore bit-serial, il tempo di esecuzione complessivo risulterebbe circa pari a 16 volte il tempo impiegato da un equivalente bit-parallel, ma grazie alla significance compression, il rallentamento è di fatto proporzionale alla quantità di bit che vengono processati, e quindi decisamente inferiore. In figura 9.8 è illustrato il throughput atteso per il BitSNAP rispetto a ciascuno dei test di valutazione, assieme alla stima del risparmio sui datapath rispetto allo SNAP/LE.

Per quanto riguarda infine il consumo complessivo di energia, tenendo in conto i consumi dovuti alle memorie, al *Fetch* e al *Decode*, le stime indicano che il BitSNAP consuma circa il 70% dell'energia dinamica consumata dallo SNAP/LE. Ciò significa circa $152pJ$ per istruzione a $1.8V$, e $17pJ$ per istruzione a $0.6V$, che, in termini complessivi, con un carico di lavoro tipico per una rete di sensori, si traduce in un consumo di potenza attiva in un range di $12 - 43nW$ a $0.6V$.

10. Il Vortex

10.1. Introduzione

Illustrare il progetto Vortex nella sua interezza sarebbe impossibile, date le enormi dimensioni, per cui in questo capitolo ci limitiamo a descriverne l'architettura e l'instruction set.

Il design di alto livello del Vortex è stato specificato e simulato in Java, con estensioni per abilitare il message passing (si veda sezione 4.2 a pagina 46). La decomposizione *top-down* è stata impiegata per creare una gerarchia di celle dapprima CSP, e PRS alla fine [75]. La circuiteria di basso livello è stata basata su alcuni modelli di pipeline integrate (WCHB, PCHB, PCFB) [63] e sul modello temporale Quasi Delay Insensitive (si veda sezione 2.4.2 a pagina 25) [74]. Il flusso di progettazione complessivo e alcune parti dell'implementazione sono simili a quelle del suo predecessore, il Caltech MiniMIPS [10]. L'obiettivo per il tempo di ciclo per la maggior parte dei circuiti era di 18 transizioni/ciclo (paragonabile alle moderne CPU *full-custom*).

Il layout invece è stato creato interamente a mano utilizzando *Magic*, mentre altri strumenti, in gran parte proprietari, sono stati impiegati per la simulazione e la verifica.

Il processore Vortex è stato progettato come parte del "TestChip 2" (*Fulcrum Microsystems*) che include anche un'interfaccia DDR-SDRAM e un'interfaccia Ethernet a $10Gb$ connessa alla CPU per mezzo di una crossbar di tipo SoC (*System On Chip*), chiamata "Nexus". Il sistema è stato progettato perché potesse scalare fino a 16 CPU e interfacce di I/O in totale. Una versione più recente del design Nexus è stata pubblicata in [64] e in questo capitolo non ci occuperemo della DDR-SDRAM e dell'interfaccia Ethernet.

Passiamo quindi a introdurre l'architettura del Vortex.

10.2. Architettura

L'architettura del Vortex consiste di un *dispatcher*, di una *crossbar*, e di una varietà di unità funzionali che comunicano mediante canali di tipo *flow-controlled*.

In figura 10.1 è illustrato un modello di esempio. Il dispatcher decodifica un flusso di istruzioni per fornire una sequenza di controllo alla crossbar e alle unità funzionali. Queste ultime hanno porte per operandi e risultati, connesse alla crossbar tramite i canali. Ciascun componente progredisce ogni volta che ottiene (1) controllo, (2) dati in input e (3) spazio a sufficienza nel buffer associato al canale di output, altrimenti stalla. Il controllo di flusso locale permette alle unità funzionali di entrare in esecuzione non appena le dipendenze dei dati sono soddisfatte. Questo comportamento di tipo *data-flow* (flusso dati) è implementato elegantemente con circuiteria asincrona.

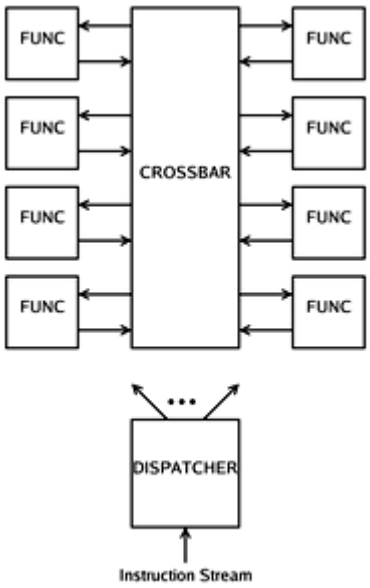


Figura 10.1 – Architettura generica per il Vortex.

10.2.1. Dispatcher

Il dispatcher riceve da un canale di input un flusso ordinato di token[(1)] di istruzioni, che può provenire dall'ambiente (*environment*) o da una cache di istruzioni. Ci sono due classi di istruzioni: (1) istruzioni *function* indicano cosa fare alle unità funzionali, mentre (2) istruzioni *move* indicano alla crossbar di spostare un token dalla porta *result* di un'unità funzionale alla porta *operand* di un'altra unità. Le istruzioni function possiedono un prefisso che indica quale unità funzionale debba ricevere i bit rimanenti (cioè quelli contenuti nel token, esclusa l'istruzione stessa). Le istruzioni move invece, codificano due porte della crossbar, sorgente e destinazione, più alcune informazioni aggiuntive. L'indice della porta destinazione viene spedito alla porta sorgente perché controlli la parte *SPLIT* della crossbar, mentre quello della porta sorgente viene spedito alla porta destinazione perché controlli la parte *MERGE*.

Per incrementare il throughput del dispatcher vengono ricevute molteplici istruzioni su canali di input in parallelo, in ordine ciclico. Le istruzioni rimangono in ordine logico, ma vengono spedite in parallelo appena è possibile. Le istruzioni su canali di output separati non necessitano di essere rimesse in sequenza poiché controllano diverse unità funzionali. Il dispatcher, tipicamente, presenta dei tempi morti nei suoi output e ciò causa l'esecuzione non ordinata del datapath (quando ciò non porti a errori).

10.2.2. Crossbar

La crossbar è dotata di N input e M output; è di tipo *non-blocking* e internamente possiede tutte le $M \times N$ possibili connessioni tra input e output. In un singolo trasferimento, la crossbar riceve un token da un canale di input e lo inoltra su un canale di output. Le informazioni per il controllo di ciascun input/output giungono dal dispatcher. Ogni qual volta si verifichi una situazione in cui una coppia di porte (una

[(1)]Si veda nota (2) a pagina 110.

input e una output) possiedono istruzioni complementari, la crossbar apre il cammino che ha la precedenza rispetto al flusso di esecuzione del programma. In questo modo istruzioni in sequenza mantengono l'ordinamento corretto, mentre istruzioni non in sequenza, relative a coppie di porte distinte, possono essere completate in modo aleatorio, senza necessità di sincronizzazione. La crossbar può instradare al massimo $min(M, N)$ token contemporaneamente e, quando il dato in input o le istruzioni tardano ad arrivare, o quando il canale di output è pieno, quel particolare trasferimento stalla senza influenzare i trasferimenti che avvengono tra le altre porte.

10.2.3. Unità funzionali

Le unità funzionali ricevono le istruzioni dal dispatcher o dalla crossbar e inviano istruzioni alla crossbar, attraverso canali di comunicazione. Istruzioni dirette a differenti unità funzionali sono o incorrelate, oppure i loro dati vengono messi in sequenza mentre fluiscono nella crossbar. Non è necessario un meccanismo di sincronizzazione globale e ciascuna unità funzionale può possedere un qualsiasi numero di porte di input e output. Comunque, tutti i dati comunicati tra due unità funzionali devono obbligatoriamente transitare nella crossbar, e ciascuna unità decide come operare sui propri utilizzando le sue istruzioni.

Inoltre, le unità funzionali possono memorizzare lo stato internamente, ma possono anche connettersi a canali esterni (cioè non diretti alla crossbar), come quelli dedicati all'interfaccia per la memoria e per il *message-passing*.

10.2.4. Instruction Set (IS)

Un'istruzione Vortex consiste di due parti: (1) un codice prefisso e (2) il corpo vero e proprio dell'istruzione. Il codice prefisso viene utilizzato dal dispatcher per decidere a quale unità funzionale debba essere spedita l'istruzione e, una volta giunta a destinazione, l'istruzione (il corpo) viene utilizzata dall'unità funzionale che l'ha ricevuta.

Il pregio di questo tipo di decodifica in due stadi è quello di permettere l'implementazione, con circuiteria uniforme, di un instradamento altamente parallelizzato, senza necessità di curarsi di quanto è contenuto nel corpo delle varie istruzioni. Le istruzioni Vortex possiedono una lunghezza fissa, che dipende dal tipo di design, anche se in realtà la porzione compresa tra il prefisso e il corpo dell'istruzione può essere regolata per ciascuna unità. Per inviare una lunga istruzione singola a un'unità funzionale, vengono spedite numerose istruzioni Vortex più corte.

In questo senso, le istruzioni Vortex sono molto differenti dalle istruzioni RISC, le quali specificano implicitamente operazioni su registri, trasferimenti dati e operazioni funzionali. Un'istruzione Vortex invece gestisce solamente uno di questi compiti e quindi è possibile dover impiegare diverse istruzioni Vortex per operare un'equivalente istruzione RISC.

Comunque, le istruzioni Vortex possono anche essere più piccole e inoltre, poiché non ci sono operazioni implicite su registri, questi non sono necessari per memorizzare variabili temporanee prodotte da un'unità e destinate a essere consumate immediatamente da un'altra. La granularità delle istruzioni Vortex quindi permette un maggiore livello di vettorizzazione rispetto alle RISC. Un'istruzione vettorizzata include una ripetizione con conteggio a N nel proprio corpo e ciò fa sì che l'ultimo sta-

dio del dispatcher la ripeta N volte. Una tipica sequenza di istruzioni RISC, invece, difficilmente prevede la ripetizione di una istruzione identica anche se ciò in effetti si verifica per alcune parti di certe istruzioni, come ad esempio nel movimento dei dati dall'*adder* al *multiplier*.

Nell'architettura del Vortex, parti non uniche di un'istruzione possono essere vettorizzate anche se, prese tutte assieme, comprendono un'istruzione RISC equivalente. La vettorizzazione è essenzialmente una compressione del flusso dati e riduce notevolmente la dimensione del codice e l'occupazione di banda per il dispatcher.

10.3. Implementazione di un prototipo Vortex

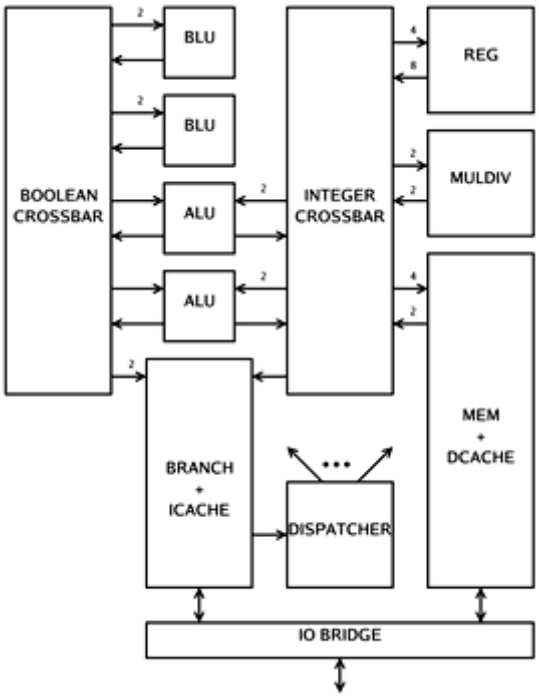


Figura 10.2 – Prototipo di un'implementazione del Vortex. I numeri sopra le frecce indicano operandi/canali multipli.

Ora che abbiamo descritto i concetti base dell'architettura del Vortex, presentiamo l'implementazione (notevolmente più complicata) del primo prototipo di CPU Vortex (figura 10.2).

Il prototipo gestisce due tipi di dato: un intero a 32 bit e un booleano a 1 bit. Come si può vedere in figura, ciascun tipo ha la propria crossbar dedicata e le unità funzionali sono in contatto con una di queste o con entrambe. Le unità funzionali sono:

- 2×BLU: Logica booleana
- 2×ALU: Logica aritmetica
- MULDIV: *Multiply/Divide*
- MEM: Memoria

- BRANCH: *Branch*
- REG: *Integer register file*

10.3.1. DISPATCHER

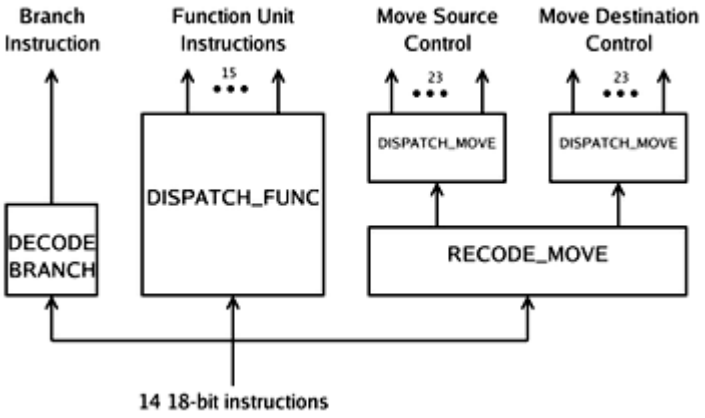


Figura 10.3 – Dispatcher.

Le istruzioni vengono lette (*fetch*) dall'*instruction cache* su linee "cache" da 8 parole, all'interno dell'unità *Branch*. A eccezione delle istruzioni di branch, che sono di lunghezza variabile, ci sono 14 istruzioni da 18 bit in ciascuna linea "cache". Il dispatcher le decodifica in flussi di controllo sequenziali per tutte le altre unità.

Nella CPU prototipo, ciò viene fatto copiando prima la linea su 3 percorsi: la prima copia è decodificata in un'istruzione di branch, se presente; la seconda passa attraverso una crossbar 14×15 che estrae il controllo per tutte le unità funzionali in ordine di programma; la terza gestisce tutte le istruzioni di spostamento dati più quelle per il register file, REG.

Tutte queste istruzioni contengono l'indirizzo di una porta sorgente e di una porta destinazione, più qualche informazione condivisa, come ad esempio la ripetizione con conteggio a N. Alla fine, uno stadio di ricodifica copia le informazioni rilevanti su due crossbar 14×23 per ciascuna delle due metà dell'istruzione (*source* e *destination*). I canali di output presentano tempi morti e ciò fa si che il flusso di esecuzione perda l'ordinamento (in realtà l'esecuzione resta comunque ordinata quando ciò sia necessario, come quando ci sono dipendenze tra dati o tra operazioni).

Alla fine di ogni unità funzionale e di ogni canale di controllo, c'è un'unità di ripetizione vettoriale che può ripetere la stessa istruzione da 1 a 8 volte. In figura 10.3 possiamo vedere il dispatcher.

L'implementazione dei circuiti *DISPATCH_FUNC* e *DISPATCH_MOVE* è stata una delle innovazioni più importanti apportate dal prototipo del Vortex. L'implementazione utilizza una crossbar per instradare il carico di N istruzioni su M canali di output, con ordinamento deterministico. Lato input, il controllo *split* S[i] deriva direttamente dai prefissi di ciascuna istruzione, e quindi realizzarlo è un compito semplice. Lato output invece, l'implementazione del controllo *merge* M[i] è molto delicata. In un sistema con crossbar ad arbitraggio, come ad esempio il Nexus [64], le celle di input inviano una richiesta (più propriamente, un *request token*) alla porta

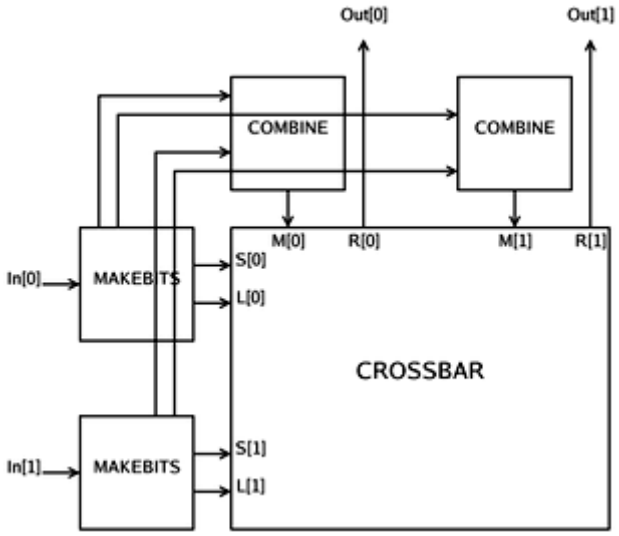


Figura 10.4 – Dispatch crossbar con controllo.

di output desiderata, che esegue un *arbitraggio* tra di esse per decidere quale sarà la prossima su cui eseguirà il *merge*.

Per controllare il *merge* in modo da mantenere l'ordinamento (deterministico) del programma, ciascun input possiede una cella *MAKEBITS* che invia un bit a ciascun output. Tutti questi bit inviati sono a 0, tranne quello relativo alla porta output desiderata, che è a 1. Le unità *COMBINE* su ciascun output raccolgono un bit da ciascun input e successivamente leggono il valore di questi bit in sequenza, attivando gli input in ordine ogni volta che incontrano un 1. In questo modo quindi, se ogni blocco di istruzioni invia solamente un'istruzione a ciascuna unità, esse possono essere consegnate in parallelo, mentre se ci sono due o più istruzioni in attesa sulla stessa unità, vengono eseguite in ordine.

La figura 10.4 mostra un semplice esempio per il caso 2×2.

10.3.2. BLU

Con riferimento alla figura 10.2, ciascuna delle due unità BLU riceve due operandi booleani e ne produce uno come risultato. Un campo di controllo a 4 bit permette a queste unità di calcolare tutte le 16 funzioni booleane a 2 input. Inoltre, 2 bit aggiuntivi vengono usati per indicare un aknowledge su ciascun operando. In questo modo, ciascun input può essere utilizzato più volte prima che venga emesso il segnale di acknowledge, e ciò dona all'unità BLU la caratteristica di operare come un rudimentale registro.

10.3.3. ALU

Ciascuna delle due ALU ha due operandi interi (nel senso di numeri *integer*) e uno booleano e produce due risultati: un intero e un booleano. L'ALU è composta internamente da istruzioni *add*/*subtract*/*negate*, un blocco logico booleano bit-a-bit, un

barrel shifter[2] e un comparatore. La maggior parte di queste operazioni sono esattamente ciò che ci aspetterebbe da una qualsiasi CPU, con alcune generalizzazioni e piccole varianti.

Ciascuna ALU possiede un'unità IMM (*immediate*) di supporto, che riceve dal dispatcher un canale di controllo separato e costruisce un valore immediato a 32 bit, 8 bit alla volta dall'LSB al MSB[3], con estensione di segno dopo ciascun byte. Questo operando immediato può essere utilizzato al posto dell'operando Y nell'ALU. Gli operandi immediati solitamente sono piccoli, anche se servono 4 cicli per costruirli; possono essere costruiti anche per le unità MEM e possono essere riutilizzati richiedendone semplicemente il MSB.

All'interno dell'ALU, l'operando Y può essere sostituito anche da uno 0, mentre X può essere sostituito sia da uno 0 che da un 1, senza alcun bisogno di utilizzare l'unità IMM.

Le istruzioni *add* / *subtract* / *negate* possono usare valori booleani come carry-in/out opzionali. Il comparatore produce solamente un risultato booleano e può essere utilizzato per eseguire tutti i 6 tipi di funzioni di confronto, con e senza segno. Il blocco logico invece può eseguire tutte le 12 possibili funzioni a 2 input e può anche ricevere un valore booleano verificando il quale può decidere se scambiare tra di loro i valori degli operandi. In questo modo implementa di fatto operazioni come select condizionati e simili. Inoltre, poiché è possibile sostituire 1 a X e 0 a Y, quest'unità è quindi in grado di convertire un booleano in un intero.

10.3.4. MULDIV

L'unità MULDIV esegue moltiplicazioni e divisioni tra interi; possiede due porte di input e due di output e gestisce dati con e senza segno. Le porte di input sono condivise tra il *multiplier* e il *divider*, mentre invece le porte di output sono private, una dedicata al multiplier e l'altra dedicata al divider. Porte separate per gli output sono estremamente utili per via della maggiore latenza della divisione rispetto alla moltiplicazione.

Il risultato di una moltiplicazione di due interi a 32 bit è contenuto in un intero a 64 bit le cui due metà (32 bit più/meno significativi) possono essere fornite in uscita sia singolarmente che assieme, in modo sequenziale. Grazie a questa caratteristica e al supporto per operandi con e senza segno, è possibile eseguire moltiplicazioni su operandi più larghi di 32 bit in cicli multipli. Il multiplier inoltre ha anche un accumulatore a 64 bit e la sua micro-architettura è un design di tipo *wallace-tree*, *booth-encoded carry-save*, che produce 4 bit del risultato per ogni stadio domino. La catena *carry-save* termina in un adder *carry-select* KPG ibrido. Questo design si basa direttamente sul multiplier/adder del MiniMIPS (si veda [10] per approfondire).

Il divider invece implementa la divisione intera standard. Il numeratore inoltre può subire uno shift verso sinistra con un offset costante fino a 32 bit, producendo un quoziente a 64 bit con un resto a 32 bit. Il risultato può consistere di una metà (o entrambe) del quoziente, il resto, o tutto assieme, in 3 blocchi esposti in sequenza. Grazie alla capacità di shift inoltre, il divider supporta anche numeri reali a virgola fissa (anche se operare con questo tipo di dati è molto dispendioso).

[2] Un *barrel shifter* è un circuito digitale che può eseguire uno *shift* su una parola, di un specifico numero di bit, in un unico ciclo di clock.

[3] LSB, MSB: Least/Most Significant Byte, cioè Byte Meno/Più significativo.

Nell'implementazione il numeratore viene normalizzato con uno shift a sinistra, poi viene applicato iterativamente il nucleo dell'algoritmo di divisione che presenta alcune variazioni rispetto a un algoritmo tradizionale (utilizza uno *skip-ahead shifter* cercando di normalizzare il resto intermedio su un massimo di 3 bit per ciclo e, successivamente, prova a sottrarre il divisore o il suo doppio). Questo procedimento può prendere da 1 a 16 iterazioni, ma tipicamente impiega solo pochi cicli grazie alla normalizzazione iniziale, molto aggressiva, e alle piccole variazioni descritte in parentesi. Il costo in termini di area è relativamente alto, circa lo stesso di un moltiplicatore *fully pipelined*, ma dato che la matematica a virgola fissa è una delle applicazioni che il Vortex si è posto come obiettivo, un divider rapido è stato ritenuto essere molto importante.

10.3.5. MEM

L'unità funzionale MEM è la più larga: supporta 2 *load* e 1 *store* in parallelo. Possiede 3 porte per gli indirizzi degli operandi, una per la memorizzazione dei dati e due per il caricamento, più 3 canali di controllo per le istruzioni e altri 3 per generare valori immediati. Inoltre, supporta load/store di parole complete e *prefetch*/*flush* su linee cache da 8 parole.

Gli indirizzi sono calcolati come somma di una combinazione di queste 4 sorgenti: una variabile, un valore immediato, 4 (il valore 4) o l'ultimo indirizzo utilizzato nel processo. Ciò permette accessi vettoriali molto efficienti dato che è possibile fornire un indirizzo base, e in seguito utilizzare *last_address+4* per gli accessi successivi.

L'unità MEM contiene 8 banchi cache di tipo *address-mapped*, per un totale di $32KB$, dai quali diramano linee cache. Ciascun banco possiede una singola porta, ma i 3 processi della memoria possono operare in parallelo se utilizzano banchi differenti. Ciascun banco include sia un array di dati da $4KB$ che il corrispondente *tag*[4]. I tag tengono traccia dell'indirizzo della linea cache, della validità dei dati e di quali parole siano state modificate; in questo modo, solo i dati che subiscono modifiche vengono aggiornati in memoria.

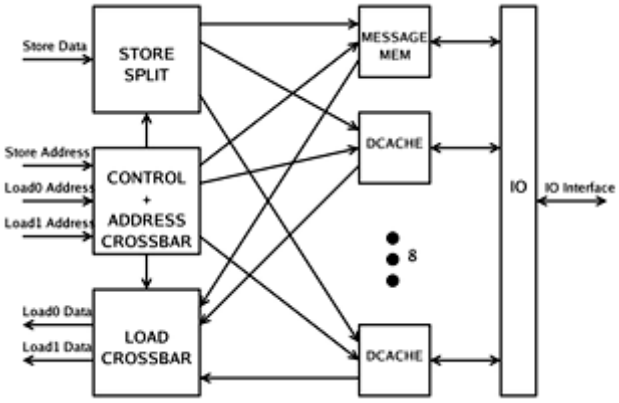


Figura 10.5 – MEM.

Gli 8 banchi sono connessi agli indirizzi e alle porte dati dei 3 processi dalla crossbar, come mostrato in figura 10.5.

[4]Un *tag* consiste di un insieme di informazioni relative a un oggetto.

In aggiunta agli 8 banchi cache, c'è un nono banco *SRAM* da $4KB$ privo di cache che può essere utilizzato come memoria locale temporanea. Questo banco ha la caratteristica di poter essere scritto anche da altri moduli della CPU o del sistema I/O e può essere utilizzato per il passaggio diretto di messaggi da moduli della CPU a quelli del sistema I/O, e viceversa. Messaggi in ingresso generano degli *interrupt*, dei quali si occupa l'unita BRANCH.

L'ordinamento relativo tra i due processi *load* e lo *store* è uno degli aspetti delicati di questo circuito. Solitamente i processi utilizzano degli *arbitri*[5] per accedere agli 8 banchi di memoria e quindi l'ordinamento può venire a mancare ma, quando ci sono delle dipendenze tra i dati, è necessario che ciò venga impedito. La soluzione a questo problema consiste nell'introduzione di canali di sincronizzazione tra i 3 processi. Le istruzioni indicano quando spedire o ricevere token sui canali di sincronizzazione, forzando in questo modo l'ordinamento corretto nell'esecuzione delle operazioni.

10.3.6. BRANCH

L'unità BRANCH possiede un operando di tipo intero e due booleani. Non dispone di una porta per i risultati (perché un salto non ne produce) e impiega un canale di fetch verso l'*instruction cache* da $32KB$ e due canali (*read*/*write*) verso la memoria esterna. L'insieme di istruzioni e funzionalità dell'unità BRANCH è abbastanza diverso da quello di una tipica CPU RISC. La differenza principale è che il supporto è esteso a istruzioni di alto livello per il controllo di flusso in stile C (come *if, do, while, for, call, return*) e non si limita solamente a primitive *branch*/*jump*.

La funzionalità più singolare è uno stack (built-in) per il branch, che gestisce chiamate a *subroutine* e cicli. Il segnale *settos* inizializza questo stack in modo che punti verso un indirizzo nella memoria principale. Lo stack si comporta in effetti come una cache di 32 elementi; quando la sua dimensione supera questa quantità, viene svuotato e ricostruito da zero dalla memoria principale. In aggiunta alle chiamate a subroutine, lo stack viene utilizzato nei costrutti di cicli hardware.

Le istruzioni di branch compaiono sempre all'inizio di un blocco base di istruzioni e diventano effettive alla fine; sono di lunghezza variabile e supportano fino a 2 indirizzi immediati a 32 bit. Le istruzioni nelle linee cache sono completate (da un lato) con istruzioni *NOP*[6] in modo che i punti di ingresso dei branch siano allineati all'inizio di ogni linea cache da 8 parole. Gli indirizzi delle istruzioni includono la lunghezza del blocco base (codificata nei 3 bit meno significativi), per un numero di linee cache che può variare da 1 a 8. In questo modo, quando si salta verso un blocco base, l'unità di BRANCH può immediatamente eseguire il fetch di al massimo 8 linee cache senza dover attendere l'ispezione di nuovo codice per individuare nuove istruzioni di salto. A causa di questa modalità operativa, spesso le istruzioni di salto stallano in attesa che arrivi un operando e, non appena questo diventa disponibile, partono immediatamente. Il lettore interessato all'intero insieme di istruzioni dell'unità BRANCH, può trovarlo in [66].

[5] Un *arbitro* è un dispositivo che gestisce l'accesso a risorse condivise.

[6] NOP: No Operation. E' un'istruzione che non esegue nulla e che spesso viene usata come segnaposto per l'allineamento di altre istruzioni.

10.3.7. REG

Vediamo infine l'unità REG, che trattiene variabili temporanee. Va notato che l'impiego di quest'unità è opzionale, in quanto è sempre possibile caricare e registrare variabili nell'unità MEM muovendole direttamente da un'unità all'altra tramite la crossbar.

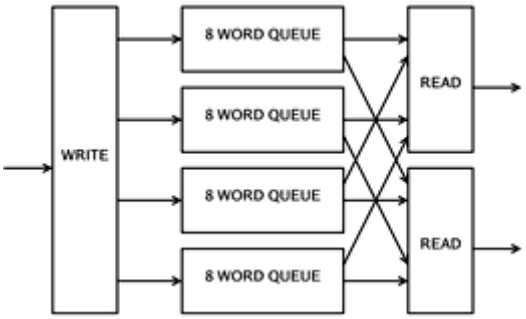


Figura 10.6 – REG: $\frac{1}{4}$ del totale register file.

A differenza di una CPU RISC, il register file del Vortex consiste di 16 code, ciascuna avente profondità di 8 parole. Queste code sono associate a diversi canali per operandi: 4 in scrittura e 8 in lettura. Ogni canale è collegato a 4 code in uscita e a 4 in ingresso. Quando viene letto un token da una coda, l'istruzione può specificare se il token vada consumato o se debba esserne mantenuta una copia. Durante l'esecuzione di un'istruzione vettoriale, la lettura si spinge sempre alla massima profondità nella coda, non limitandosi alla sola testa. E' anche possibile leggere lo stesso dato da due porte differenti in modo da duplicarlo in parallelo (e quindi evitando di doverlo leggere due volte in sequenza).

L'unità REG è illustrata in figura 10.6.

Il controllo dell'unità non viene codificato come accade per le normali unità funzionali, ma viaggia incorporato in istruzioni *move* che controllano la crossbar (la tecnica è quella del *piggy-backing*).

Il design di questa unità rappresenta probabilmente il problema più arduo riscontrato nel prototipo Vortex. Infatti, nonostante il REG possa essere pilotato da codice macchina altamente ottimizzato (manualmente), l'assemblatore e il compilatore di alto livello non hanno la capacità di sfruttarlo in modo efficace. Lo spazio per lo *storage* di 128 parole (complessivamente) è una sua buona qualità, ma i limiti nell'accessibilità costituiscono dei grossi inconvenienti. Per via della natura delle variabili (viaggiano in token) è spesso necessario caricare [svuotare] il register file dalla [in] memoria nelle zone di confine tra i blocchi base, e ciò comporta severe penalizzazioni nelle prestazioni. Un'alternativa migliore sarebbe quella di avere semplicemente 16 variabili che possano essere lette/scritte in ordine (da programma) come avviene nelle tradizionali CPU.

10.4. Dinamiche di esecuzione

La natura del Vortex, distribuita e *event-driven*, gli permette di disporre di molte utili dinamiche di esecuzione. In una macchina che gestisce un flusso dati in modo ideale, ogni operazione i cui operandi siano stati definiti può essere eseguita. Invece,

quando questi non sono ancora disponibili, l'esecuzione deve stallare. Questo tipo di sequenzialità è dovuta alle dipendenze tra i dati. Per dipendenze tra i dati si intende quella situazione in cui un'istruzione, per poter essere eseguita, necessita dei risultati di altre operazioni. Quando i risultati da cui un'istruzione dipende non sono ancora disponibili, l'esecuzione di quest'ultima deve entrare in stallo. Gli stalli per via delle dipendenze tra i dati sono funzione delle latenze delle unità funzionali e delle interconnessioni, ma anche della struttura del programma.

Qualsiasi altro tipo di sequenzialità delle operazioni, nel Vortex, è dovuto a dipendenze strutturali, le quali vengono classificate come dipendenze di canale, funzionali e di controllo, a seconda del tipo di risorsa condivisa in gioco.

Le dipendenze funzionali riguardano la sequenzialità nell'uso delle unità funzionali. Supponiamo che due istruzioni I_1 e I_2 abbiano bisogno di utilizzare la medesima unità funzionale U. I_1 è la prima a farne richiesta, e quindi riceve l'accesso e impegna U in un'elaborazione. Subito dopo, I_2 richiede a sua volta l'accesso che però non le viene accordato, poiché U è ancora impegnata. A questo punto I_2 entra in stallo, e attende finché U non abbia completato le operazioni di I_1. Questo tipo di stalli è causato principalmente dalla latenza con cui i dati vengono instradati da e verso un'unità funzionale, e dal (basso) throughput di quest'ultima. Per eliminarli quindi, è necessario impiegare circuiti con latenze più brevi e maggiore throughput, non essendo sufficiente modificare solamente l'architettura.

Le dipendenze di canale si incontrano ogni qual volta uno stesso canale sia richiesto da due operazioni per lo spostamento di un dato. Nel caso del Vortex, poiché la crossbar è completamente connessa e indipendente, e fornisce una specie di canale "privato" tra ciascuna coppia di unità funzionali, dipendenze di canale non costituiscono mai un collo di bottiglia. Volendo fare un paragone con una macchina RISC superscalare, questa tipicamente presenta molte unità funzionali distribuite su un ridotto numero di pipeline e ciò fa si che le unità che sfruttano una stessa pipeline debbano lavorare in modo mutuamente esclusivo. Per esempio, se sommatore e moltiplicatore fossero sulla stessa pipeline RISC, allora addizioni e moltiplicazioni non verrebbero mai eseguite in parallelo.

Le dipendenze di controllo invece sorgono quando i dati di un'operazione arrivano a un'unità prima delle relative istruzioni, ovvero quando l'unità non riceve le istruzioni con adeguata velocità (prima dei dati). Il dispatcher del Vortex è implementato con un elevato livello di parallelismo, in modo che molte istruzioni possano essere inviate simultaneamente, e in modo che istruzioni destinate a diverse unità non siano stallate da quelle destinate ad altre unità. L'introduzione di tempi morti nei canali del dispatcher (o in quelli subito dopo il dispatcher) fa sì che istruzioni lette nello stesso istante vengano eseguite di fatto in tempi molto diversi. Ciò, combinato con l'efficiente connettività della crossbar, permette il riordino dinamico dell'esecuzione delle istruzioni in differenti unità. Inoltre, poiché i tempi morti del dispatcher sono quantità finite, le istruzioni non possono disordinarsi in modo completamente aleatorio perché, se così fosse, il dispatcher si intaserebbe finendo per stallare istruzioni incorrelate. Le istruzioni destinate a un'unità vengono eseguite in ordine di programma e ciò fa sì che vengano rispettate le dipendenze tra i dati. In effetti, sarebbe possibile eseguire un ordinamento tra le istruzioni che giungono a un'unità, ma questo richiederebbe tempi e circuiti supplementari che non farebbero altro che degradare le prestazioni del sistema e aumentare la complessità della logica di controllo. Le istruzioni inoltre dovrebbero essere riordinate durante la compilazione in modo che

arrivino con lo stesso ordine con cui arrivano gli operandi su cui lavorano. Questa sarebbe in effetti una buona modalità operativa, eccetto per quei casi in cui le latenze degli input non siano conosciute (es. quando deve essere recuperato da un memoria cache).

Il Vortex quindi può essere visto come una macchina che gestisce un flusso dati, con una finestra (di dimensione finita) di istruzioni che possono essere riordinate. Questo tipo di approccio elimina l'enorme costo della ricerca di operazioni abilitate, che affligge i design che gestiscono flussi dati (più propriamente: *pure dataflow design*).

Il register file del Vortex non ha un ruolo da protagonista come invece accade nel design RISC. Ciò è dovuto, come abbiamo già visto, al fatto che la completa connettività della crossbar permette di spostare dati dagli output di un'unità agli input di un'altra in modo diretto, senza passare per il register file (come fare un "register bypass"). Questo comportamento viene particolarmente utile nel trasposto di dati che coinvolge unità di memoria.

Per quanto riguarda le istruzioni vettorizzate, un programma può configurare una pipeline, ad esempio che parta e torni in memoria attraversando multiplier e adder. In questo modo, è possibile eseguire lunghe sequenze di istruzioni simili velocemente, come accade nei supercomputer vettoriali.

Una strategia comune nelle architetture RISC, volta a compensare le diverse latenze delle varie unità, è quella di controllare le unità che presentano latenze elevate con una istruzione "launch", che avvia la computazione, e con un'istruzione "land", che recupera il risultato. Tutte le istruzioni del Vortex si comportano di fatto in questo modo, poiché le istruzioni *move* che spostano i dati sono separate. Ciò permette di avviare in anticipo, rispetto al flusso del programma, le unità più lente (come quelle che effettuano divisioni o caricamenti in memoria), utilizzando i loro risultati molto tempo dopo. Un calcolo molto lento può quindi sovrapporsi a un certo numero di istruzioni più rapide, finché non viene terminato con un'istruzione *move* (che corrisponde al "land" delle architetture RISC).

Il Vortex può eseguire simultaneamente 3 istruzioni aritmetiche, 2 logiche, 3 che coinvolgono la memoria e una di branch. Ciò fornisce un picco nel throughput che equivale a 9 istruzioni RISC per ciclo. Inoltre, l'introduzione di unità funzionali aggiuntive dotate di funzioni più specializzate, diminuisce l'utilizzo medio di ciascuna unità, ma di fatto incrementa le prestazioni complessive del sistema.

10.5. Il compilatore

Scrivere un compilatore per l'*instruction set* del Vortex (VIS) non è cosa semplice. Ad esempio, basta dimenticarsi di accoppiare un'istruzione *move* con un'istruzione *function* in modo corretto per provocare il deadlock della CPU, risolvibile solo con un *hard* reset. La strategia seguita per affrontare il problema è stata quella di creare un linguaggio assembly "intermedio" (VASM) che nasconda questo tipo di problematiche e crei sempre un immagine binaria corretta, cioè, in questo caso, *deadlock-free*.

La sintassi del VASM deriva essenzialmente da una macchina RISC ideale, con un numero illimitato di registri identificabili. Ciascuna istruzione hardware del VIS è stata dotata di una controparte nel VASM, omettendo vari dettagli di basso livello come l'esplicito spostamento dei dati, la memorizzazione nei registri e la vettorizzazione.

```
// dot=v[0]*x+v[1]*y+v[2]*z+v[3]*w
load v[0] v0
load v[1] v1
load v[2] v2
load v[3] v3
mul v0 x t0
mul v1 y t1
mul v2 z t2
mul v3 w t3
add t0 t1 t4
add t2 t3 t5
add t4 t5 dot
```

Figura 10.7 – Prodotto scalare in 4 dimensioni (codice VASM).

In figura 10.7 presentiamo un semplice esempio di codice VASM, per un prodotto scalare in 4 dimensioni, dove un vettore v viene caricato dalla memoria e l'altro invece si trova nei registri:

```
move 0 max
memx v          // load variable address
3*memx lo       // load last address plus 4
4*move mdx muldivx
4*move 1 muldivy
muldiv mul      // multiply
2*muldiv mac    // multiply-accumulate
muldiv mac:l    // multiply-accumulate, output low
move mulz 2
```

Figura 10.8 – Prodotto scalare in 4 dimensioni (codice VIS).

In figura 10.8 invece è riportato il codice VIS relativo all'esempio. Gli N prefissi "*" rappresentano ripetizioni vettoriali e l'indirizzo base di v risiede nel registro 0. Le variabili x, y, z, w invece sono memorizzate nella coda del registro 1 e il risultato (dot[7]) viene inviato al registro 2.

Questo codice è sufficientemente piccolo per stare in una sola istruzione (su linea cache) e quindi può essere letto in un singolo ciclo. La memoria e il moltiplicatore richiedono 4 cicli sequenziali ciascuno e quindi il Vortex esegue l'equivalente di 12 istruzioni RISC in 4 cicli (per questo programma).

Per altri esempi, come nella moltiplicazione di matrici di grandi dimensioni, è possibile mantenere una media di 3 istruzioni RISC per ciclo.

Il compilatore VASM non deve preoccuparsi di gestire sintassi o strutture dati complicate, quanto piuttosto di creare grafici di flusso dati e assegnare operazioni e variabili alle risorse hardware. Realizzare ciò è stato abbastanza semplice; realizzarlo in modo efficiente invece è stato un compito molto difficile. Per supportare il C, il primo tentativo è stato scrivere un compilatore partendo da zero, ma questo ha portato i progettisti a comprendere che invece sarebbe stato più semplice modificare *Gcc*

[7] La ragione per cui il risultato del prodotto scalare si chiama *dot* deriva dal nome inglese di questa operazione: *dot product*.

perché supportasse un back-end VASM (e, così facendo, in automatico si è ottenuto il supporto per diversi front-end, come Java e C++). Quindi il codice C passa per 2 fasi di compilazione prima di diventare codice VASM e in questo senso il processo è esattamente identico a quello della compilazione del *bytecode* Java.

Al termine del progetto è stato possibile scrivere (e far girare) qualsiasi programma scritto in C, o direttamente in assembly VASM.

Utilizzando questi strumenti, è anche stato possibile simulare i test di Dhrystone 2.1 a $0.69DMIPS/MHz$. Questo valore è inferiore allo $0.91DMIPS/MHz$ raggiunto dal MiniMIPS [9] e allo $0.88DMIPS/MHz$ raggiunto dall'R3000 [10]; c'è però da notare che Dhrystone presenta alcune imperfezioni, ben note ai tester, e i suoi test presentano caratteristiche molto diverse dalle applicazioni tipiche per cui il Vortex è stato progettato. La principale causa delle non brillanti prestazioni raggiunte nei test Dhrystone sembra essere la mancanza di operazioni di tipo *byte memory*.

Implementando i test con codice scritto a mano direttamente in VIS, invece, è stato possibile ottenere risultati di gran lunga migliori, sostenendo medie di numerose istruzioni per ciclo. La discrepanza tra le due modalità sembra derivare principalmente dall'implementazione molto restrittiva del register file. Un register file più flessibile infatti avrebbe semplificato enormemente il lavoro del compilatore.

Ovviamente, se fosse stato speso più tempo (e fondi) nel progetto, anche i risultati sarebbero stati migliori.

10.6. Conclusioni

L'architettura Vortex è stata fabbricata nel 2001 con tecnologia TSMC a $0.15\mu m$ (processo G), come parte del "TestChip 2" (*Fulcrum Microsystems*). TestChip 2 ha dimensioni $10mm \times 10mm$ e la CPU Vortex occupa circa $50mm^2$ e $10.5M$ di transistor ($6.68M$ in logica, $3.34M$ per la SRAM 6T e $0.48M$ per la SRAM 10T). Queste dimensioni sono notevolmente maggiori rispetto a quelle di un semplice processore RISC di tipo *single-issue*, come il MiniMIPS ad esempio, anche se in effetti la differenza è dovuta in gran parte alla cache. Il design non è stato compresso al limite poiché il pezzo fabbricato era solo un prototipo. Sfortunatamente, un errore di progettazione nella SRAM 6T (la prima implementazione), ha causato il malfunzionamento di alcune linee di bit che venivano configurate sempre al contrario e alla fine il problema è stato risolto con delle modifiche nel compilatore e altri accorgimenti minori. L'implementazione su TSMC a $0.13\mu m$ produce prestazioni di circa $1GHz$ per ALU, crossbar e register file, alimentati con una tensione nominale di $1.5V$, a temperatura ambiente.

La mancanza di fondi non ha permesso né di poter creare altre implementazioni del processore (che correggessero i problemi sorti nella prima implementazione), né di eseguire test più approfonditi.

Per queste ragioni, nonostante fosse quasi perfettamente funzionante, il Vortex non è mai diventato un prodotto commerciale. Lo sforzo richiesto in ambito software (combinato all'inesistente domanda di mercato) per il supporto di un set di istruzioni completamente nuovo è una delle ragioni principali del mancato raggiungimento della fase di produzione.

I test eseguiti sul primo prototipo ne hanno comunque confermato la notevole qualità e infatti al giorno d'oggi molte sue parti come la crossbar, le memorie, l'aritmetica

e alcune librerie, vivono in altri prodotti commercializzati con successo dalla Fulcrum Microsystem.

11. Amulet3

In questo capitolo presentiamo Amulet3, un microprocessore ARM[(1)] ad alte prestazioni di tipo self-timed[(2)].

11.1. Introduzione

L'attuale design elettronico industriale è basato in modo predominante sull'uso di circuiti sincroni. Le tecniche di implementazione dei circuiti asincroni sono sempre rimaste in ombra, per via dell'apparente difficoltà di sviluppare progetti che presentino un'effettiva affidabilità. Negli ultimi tempi però, due fattori hanno contribuito a rinnovare l'interesse verso questo approccio:

- Con l'avanzare delle tecnologie di processo dei semiconduttori, il design sincrono sta diventando sempre più complesso, in particolar modo per quanto riguarda la gestione del *clock skew*, dei consumi e della compatibilità elettromagnetica (EMC).

- Le tecniche di design asincrono sono avanzate progressivamente in ambiente accademico e in alcuni laboratori di ricerca industriali e, nell'ultimo decennio, molte delle problematiche che affliggevano questo tipo di tecnologia hanno trovato soluzione mediante l'uso di nuovi stili e strumenti [110, 116].

Dato che il design asincrono è scevro di problematiche come il *clock skew* e in aggiunta presenta proprietà notevoli come basso consumo e una migliore EMC [86], c'è l'opportunità per questi vantaggi tecnici di essere tradotti in vantaggi commerciali e di conseguenza per il design asincrono di poter assumere il ruolo che gli spetta nell'orizzonte industriale. Un numero sempre crescente di gruppi a livello accademico e industriale sta producendo processori di tipo self-timed, specifici per svariate applicazioni. Alcuni esempi recenti includono: TITAC2 [8], il Caltech MIPS [10], il Philips 80C51 [42], ASPRO-216 [72], il Cogency DSP [86] e il TiniRISC LSI Logic/DTU [57].

La serie di microprocessori Amulet è stata sviluppata per dimostrare la fattibilità e la praticità dell'uso di tecniche asincrone in applicazioni embedded. Le pietre miliari includono:

- AMULET1 [92, 106]: sviluppato tra il 1991 e il 1993, ha mostrato che il design asincrono complesso è possibile.

---

(1) L'architettura ARM (precedentemente *Advanced RISC Machine*, prima ancora *Acorn RISC Machine*) indica una famiglia di microprocessori RISC a 32 bit sviluppata da ARM Holdings e utilizzata in una moltitudine di sistemi embedded. Grazie alle sue caratteristiche di basso consumo (rapportato alle prestazioni) l'architettura ARM domina il settore dei dispositivi mobili dove il risparmio energetico delle batterie è fondamentale. Attualmente la famiglia ARM copre il 75% del mercato mondiale dei processori a 32 bit per applicazioni embedded, ed è una delle più diffuse architetture a 32 bit del mondo. [http://it.wikipedia.org/wiki/Architettura_ARM]

(2) Si veda sezione 2.4.2 a pagina 25.

- AMULET2e [107]: (1994-1996) ha mostrato che i vantaggi del design asincrono possono essere effettivamente realizzati nella pratica.
- AMULET3: (1996-1998) il cui sviluppo è stato effettuato per stabilire il livello di vitalità commerciale del design asincrono.
- AMULET3i: (2000) è un'evoluzione più complessa della versione 3 e per questo motivo nel presente capitolo esporremo solo lo schema di base, rimandando alla bibliografia [53] il lettore che voglia approfondire.

Come i suoi predecessori, L'Amulet3 è un microprocessore ARM-compatibile che presenta tutte le funzionalità dell'ARM, compresi *interrupt* e *memory fault*. Le versioni 1 e 2 implementano l'architettura ARMv3 (terza versione dell'architettura ARM), mentre l'Amulet3 implementa la versione attuale, v4T, che include l'instruction set *Thumb* a 16 bit [104], una rappresentazione compressa del set di istruzioni ARM a 32 bit che migliora la densità del codice e l'efficienza nel consumo di potenza.

L'obiettivo dell'Amulet3 è quello di produrre un'implementazione asincrona dell'architettura ARM 4T che sia competitiva, in termini di consumi e prestazioni, rispetto al nucleo sincrono ARM9TDMI. Ciò implica avere come obiettivo prestazioni oltre i $100MIPS$ (misurazioni Dhrystone 2.1) su un processo a $0.35\mu m$, quindi un bel salto in avanti rispetto ai $40MIPS$ forniti dall'Amulet2e su un processo a $0.5\mu m$. L'incremento delle prestazioni di almeno un fattore 3, ha richiesto cambiamenti radicali nell'organizzazione del nucleo.

In questo capitolo vedremo il contesto in cui viene utilizzato l'Amulet3, il suo nucleo, i problemi sorti per garantire la compatibilità con il set di istruzioni ARM, e infine la struttura della *dual-ported memory*.

11.2. Il sottosistema Amulet3

L'Amulet3 è stato sviluppato nel contesto di una applicazione su microcontrollore che necessita di essere interfacciata a vari controller sincroni periferici. Per sfruttare appieno i vantaggi derivanti dall'uso della modalità operativa asincrona, il nucleo deve avere accesso a una memoria che opera anch'essa in modo asincrono. Dall'uso di una memoria asincrona *off-chip*, derivano inoltre enormi benefici in termini di EMC.

L'organizzazione del sottosistema asincrono è illustrata in figura 11.1. Il nucleo dell'Amulet3 è connesso direttamente con una RAM *dual-ported* (i cui dettagli vedremo più avanti nel capitolo) e con il bus MARBLE *on-chip* [119], il quale è concettualmente molto simile al bus AMBA dell'ARM [36], tranne per il fatto che il primo non utilizza un clock.

L'accesso alle rimanenti componenti del sistema avviene tramite il bus MARBLE; queste componenti includono una ROM on-chip, un controller DMA, un bridge verso il bus asincrono, dove risiedono controller per applicazioni specifiche, e un'interfaccia a una memoria esterna, che presenta un insieme di segnali convenzionali per un dispositivo off-chip. E' simile all'interfaccia dell'Amulet2e e, oltre a essere altamente configurabile, utilizza dei riferimenti temporali per gli accessi verso l'esterno [107].

Nel complesso, circa la metà dei dispositivi opera in modo asincrono e, dato che è composta da tutti i componenti ad alte prestazioni e dalla maggior parte dei driver off-chip, l'efficienza in termini di EMC e prestazioni non risulta troppo compromessa dalla presenza di dispositivi periferici sincroni.

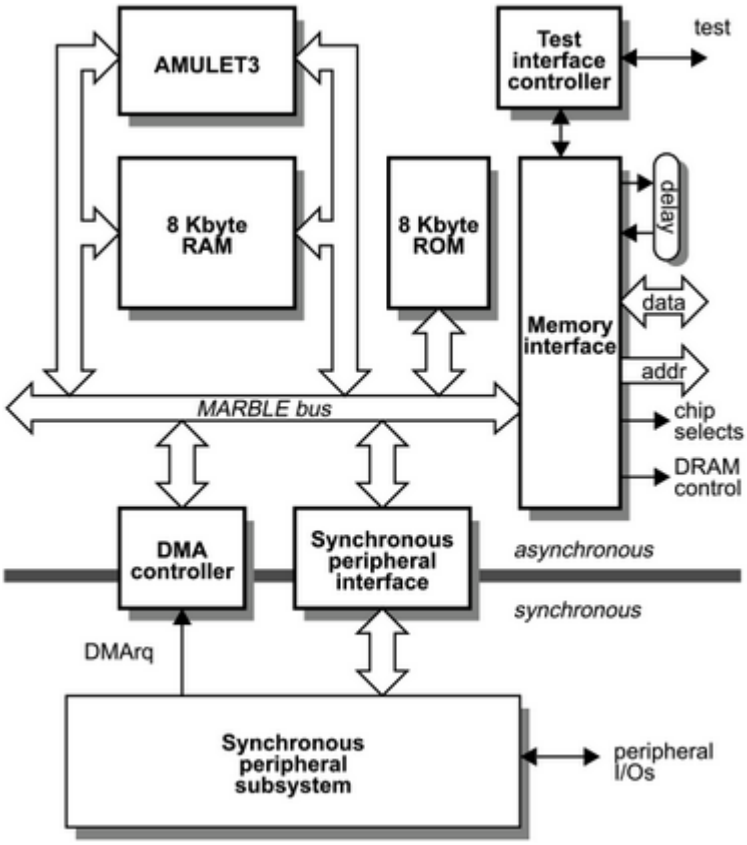


Figura 11.1 – Sottosistema asincrono Amulet3.

11.3. Organizzazione del nucleo

L'instruction set (IS) dell'ARM è di tipo *register-oriented* (orientato ai registri) e quindi l'organizzazione del register file è fondamentale per l'operatività del processore. Il register file dell'Amulet3 segue il paradigma dell'ARM9TDMI, in quanto presenta tre porte in lettura invece che solamente due, come invece è per le precedenti versioni (sia dell'Amulet che dell'ARM).

Un semplice esame dell'instruction set dell'ARM non offre una spiegazione convincente per questa scelta; la frequenza delle istruzioni che richiedono tre operandi sorgente è infatti troppo bassa per giustificare il costo per una terza porta in lettura. La motivazione è che, senza una terza porta, istruzioni che richiedono tre operandi sorgente devono essere decodificate e controllate separatamente così da poter accedere agli operandi in soli due cicli. Questo aggiunge notevole complessità alla logica di decodifica e inoltre la rallenta. Aggiungere una terza porta elimina la necessità di gestire queste istruzioni separatamente e ciò semplifica la logica di decodifica e di controllo in un modo che ne ripaga ampiamente il costo.

Il risultato è che quasi tutte le istruzioni dell'Amulet3 sono eseguite in un singolo ciclo, a eccezione della moltiplicazione a 64 bit che potrebbe richiedere quattro operandi e produrre il risultato in due segmenti da 32 bit ciascuno, la qual cosa richiede due cicli.

Vale la pena di menzionare che le operazioni *load* e *store* su registri multipli (LDM-

/STM) sono eseguite come istruzioni a singolo ciclo: mentre l'interfaccia dati e parte del decoder devono ciclare ripetutamente, l'ALU è utilizzata solo una volta per calcolare gli offset necessari ed è probabile che il pre-caricamento dell'istruzione stalli. Questa è semplicemente una conseguenza della natura asincrona del processore, che contribuisce al basso consumo delle operazioni.

11.3.1. Il reorder buffer

Le prestazioni di un processore che impiega una pipeline, dipendono fortemente dalla capacità di mantenere un flusso dati regolare all'interno della pipeline stessa. Questo flusso viene disturbato ogni volta che la pipeline stalli o venga svuotata (*flushed*).

Molti stalli nella pipeline sono dovuti alle dipendenze tra istruzioni consecutive, quando un'istruzione non può procedere finché non le vengano forniti gli operandi dalle istruzioni precedenti. L'Amulet1 impiega un meccanismo di bloccaggio sui registri [87] per gestire queste dipendenze; ciò assicura una corretta operatività, ma non è efficace nel minimizzare gli stalli. L'Amulet2 quindi ha introdotto percorsi di spedizione dati che rimuovono gli stalli nei casi più comuni, ma il sistema risultante è molto complesso da controllare. Nell'Amulet3 invece, viene impiegato un approccio completamente differente, basato sull'uso di un *reorder buffer* [40], come mostrato in figura 11.2.

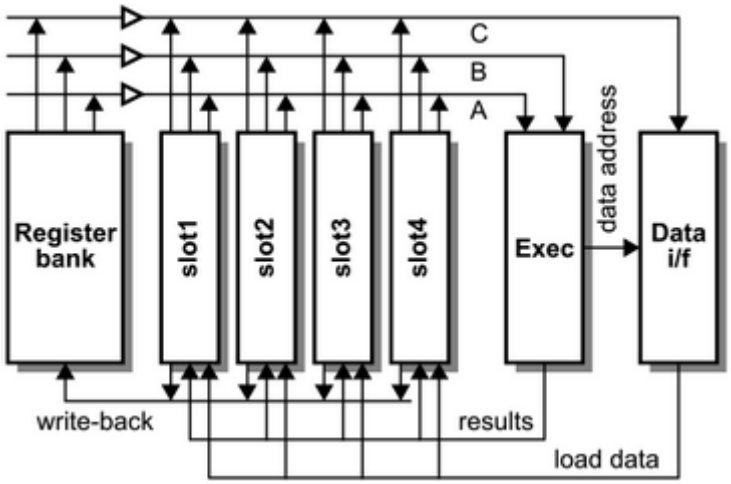


Figura 11.2 – Organizzazione con reorder buffer.

Il reorder buffer non è affatto un'idea nuova ed è stato usato anche in processori sincroni. Comunque, come spesso avviene per alcune funzionalità organizzative dei processori sincroni, trasferire l'idea in un modello asincrono richiede un'attenta riprogettazione del principio di operatività del componente poiché, venendo a mancare la sincronizzazione globale dovuta al clock, molte delle assunzioni temporali concernenti il funzionamento del dispositivo, perdono di validità.

Il funzionamento di base del reorder buffer dell'Amulet3 è molto semplice. I risultati arrivano da uno tra due (o tre) flussi: i risultati calcolati internamente vengono passati direttamente dall'unità di esecuzione mentre i dati caricati dalla memoria vengono passati dall'interfaccia dati. Dato che questi due flussi non sono sincroniz-

zati, generano risultati non ordinati. Ciascun risultato che arrivi può essere rispedito perché vada ad abilitare l'istruzione successiva.

I risultati vengono poi scritti nel register file in ordine e, durante questa operazione, ne viene saggiata la validità: un tentativo di caricamento dalla memoria può causare un *memory fault*, nel qual caso viene scatenata un'eccezione e i risultati associati a quell'operazione vengono scartati, mentre lo stato del processore rimane lo stesso, in modo che possa proseguire correttamente una volta che l'eccezione sia stata rimossa e i dati corretti.

Fin qui l'uso del reorder buffer è di tipo convenzionale. La difficoltà che va affrontata è l'esatta determinazione del momento in cui i dati nel reorder buffer possono essere spediti, visto che il processo di scrittura nel register file opera in modo completamente asincrono rispetto al meccanismo di spedizione. L'idea che risolve il problema consiste nell'osservare che la scrittura di un dato nel register file è un semplice processo di copia che non invalida il valore nel register file stesso. Il dato può quindi continuare a essere spedito finché la sua posizione non viene occupata da un altro risultato. Dato che l'allocazione viene eseguita nello stadio di decodifica, dove è anche controllato il meccanismo di spedizione, in realtà non c'è bisogno di alcun meccanismo di sincronizzazione. Il meccanismo di spedizione quindi funziona ignorando completamente se il valore che sta inviando sia già stato trasferito nel register file, anzi, le due operazioni possono persino avvenire in parallelo.

In quest'ultimo caso potrebbe verificarsi che il registro venga cambiato nello stesso istante in cui viene letto e ciò ovviamente produce un risultato indeterminato. Comunque, in situazioni come questa, la modalità operativa prevede che il valore del registro venga sempre rimpiazzato da quello del dato spedito. Per garantire la corretta operatività, è quindi necessario solamente fare in modo che i valori indeterminati (che potrebbero anche consistere di livelli logici non validi) non consumino troppa potenza.

Un'ulteriore complicazione nel reorder buffer, in relazione all'instruction set dell'ARM, è che ciascuna istruzione può essere eseguita in maniera condizionale e quindi possono essere lanciate istruzioni dalle quali ci si aspetta un risultato che magari successivamente non viene prodotto. Contrariamente al caso in cui si verificano dei *memory fault*, queste istruzioni devono convivere con operazioni valide e quindi marcano solamente il proprio risultato come non valido. Il meccanismo di spedizione quindi deve essere in grado di gestire situazioni in cui, ad esempio, numerose istruzioni ARM consecutive modificano un registro che rappresenta una sorgente per altre istruzioni. In questo caso, bisogna cercare nel reorder buffer (in ordine) per trovare il primo risultato valido associato a quel registro e procedere alla riassegnazione. La ricerca a volte è anche costretta ad attendere l'arrivo di un valore per poterne giudicare la validità. Il caso tipico è molto semplice: ci saranno uno o zero valori da esaminare e il caso peggiore consiste nel doverli esaminare tutti in sequenza. Comunque, le strutture di controllo asincrono tollerano facilmente la differenza nei tempi di ricerca e, ammesso che questa situazione si verifichi raramente, l'occasionale ricerca con il verificarsi del caso peggiore non inficia le prestazioni complessive del sistema.

11.3.2. Predizione dei salti

Gli svuotamenti nella pipeline (*flushes*) avvengono come conseguenza di istruzioni di salto (*branch*). L'Amulet3 minimizza l'impatto di questi eventi utilizzando un mec-

canismo di *branch prediction* che è un'evoluzione di quello impiegato dall'Amulet2e.

Anche se l'Amulet2e è in grado di prevedere le operazioni di salto, le istruzioni di salto vengono comunque lette in ordine per determinare il loro stato condizionale e per capire se salvano un risultato oppure no. Questo significa che un'istruzione a 32 bit viene letta per ottenere 5 bit di informazione e ciò è chiaramente inefficiente. Nell'Amulet3, quindi, questi bit vengono registrati nell'unità di *prefetch*, assieme all'obiettivo del salto. Quando un salto viene predetto, l'istruzione è già presente all'interno del processore e quindi il ciclo di memoria (necessario per la sua lettura) viene evitato. Siccome le istruzioni di salto sono circa il 10%-15% del totale [49] e la maggioranza di queste è in una cache [122], ciò rappresenta un considerevole risparmio di potenza. Oltretutto, il ciclo di prefetch è notevolmente più rapido di un accesso in memoria e ciò viene sfruttato automaticamente dalla pipeline asincrona.

11.3.3. Arresto e interruzione

L'esperienza con l'Amulet2e ha dimostrato i benefici di un meccanismo di arresto (*halt*) in un sistema asincrono, specialmente per via del risparmio di potenza [107]. Fermare la pipeline in qualunque punto causa l'arresto dell'intera pipeline in tempi brevissimi e il consumo di potenza precipita pressoché a zero (non essendoci un segnale di clock che causa ulteriori transizioni); la ripresa della piena operatività è altrettanto rapida. L'Amulet2e e l'Amulet3 decodificano un salto che punta a se stesso come un'istruzione di arresto e la utilizzano per fermare la pipeline, ma mentre l'Amulet2e stalla nello stadio di esecuzione, l'Amulet3 adotta un modello migliore, stallando nello stadio di prefetch. Ciò significa che il processore riparte con una pipeline vuota, fornendo la risposta più rapida possibile, e i dati da scartare sono eliminati all'ingresso del segnale di arresto.

La figura 11.3 mostra un'interessante conseguenza di questo fatto: i segnali di interruzione sono inseriti nell'unità di prefetch e non nell'instruction decoder. Questa insolita funzionalità architetturale fornisce sia un modello chiaro per la gestione degli interrupt, sia una loro bassa latenza. L'emissione di un segnale d'interruzione è identificata e gestita nel flusso di prefetch nello stesso modo in cui avviene per le operazioni di salto; poiché il codice dell'interruzione comincia in una posizione predefinita, l'unità di prefetch può modificare il flusso di lettura delle istruzioni, come se effettivamente fosse in grado di predire i salti. In seguito a un'interruzione viene inviato un marcatore per salvare l'indirizzo di ritorno (il quale può bypassare la parte di prefetch che coinvolge la memoria), e a questo segue immediatamente il codice di servizio dell'interruzione.

11.3.4. Salti incondizionati

I programmi per l'ARM spesso caricano il program counter (PC) direttamente dalla memoria come parte del ritorno da una procedura (e, meno frequentemente, accedendo a tabelle di salto). Quando il PC è caricato dallo stack durante il ritorno da una procedura, viene caricato dopo che ogni registro, facente anch'esso parte della stessa istruzione di caricamento, sia stato ripristinato. Quest'ordinamento, necessario per permettere al processore di riprendersi da un eventuale *memory fault* (che potrebbe verificarsi durante il caricamento), ritarda la ripresa della lettura delle istruzioni, degradando le prestazioni del sistema.

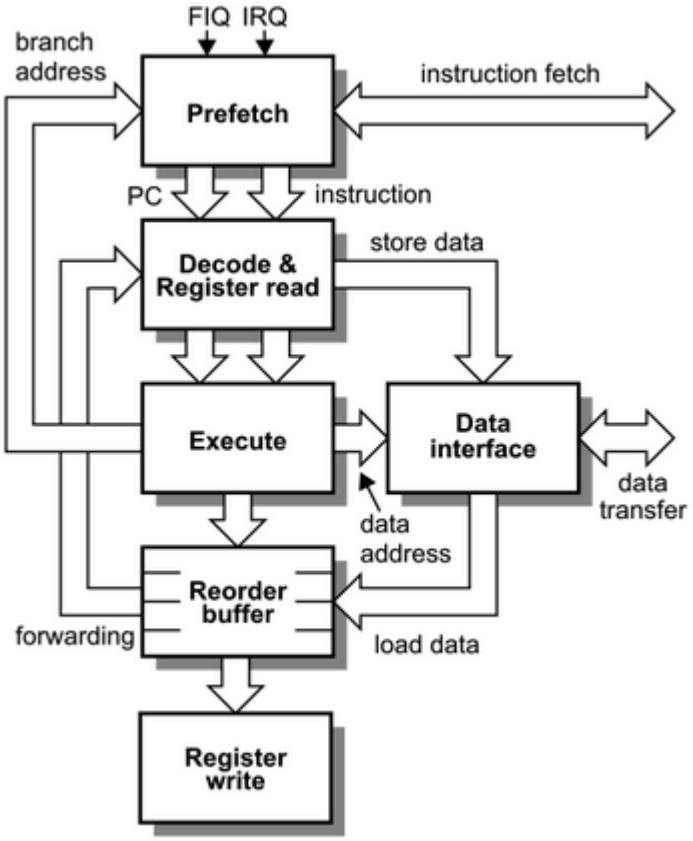


Figura 11.3 – Organizzazione del nucleo dell'Amulet3.

L'Amulet3 incorpora un'ottimizzazione che sfrutta la separazione tra le porte dell'instruction memory (figura 11.3). L'unità di esecuzione passa l'indirizzo del PC all'unità di prefetch nella prima fase del trasferimento, attraverso il percorso *branch*, permettendole di leggere in anticipo il valore del PC e quindi di ricominciare a leggere le istruzioni dall'indirizzo di ritorno (della procedura). In seguito, l'interfaccia dati deve comunque leggere il PC per verificare la presenza di un possibile *memory fault*.

Si noti che questa ottimizzazione richiede che le porte separate (istruzioni e dati) siano connesse in modo coerente alle memorie, dato che il PC viene salvato attraverso la porta dati ma viene caricato dalla porta istruzioni.

11.4. Compatibilità con l'ARM v4T

I precedenti processori Amulet hanno realizzato la piena compatibilità con il processore ARM6, ma questo insieme di istruzioni è stato prepotentemente soppiantato dalla nuova versione sviluppata per la famiglia ARM [50]. In particolare, nel mercato per i controller incorporati e portatili, l'instruction set *Thumb* a 16 bit [104] è stato accolto con grande entusiasmo, soprattutto grazie agli effetti benefici della densità del codice, da cui deriva l'efficiente gestione dei consumi. In sistemi con memorie esterne a 8 e 16 bit, il codice *Thumb* inoltre migliora anche le prestazioni.

11.4.1. Codice Thumb

L'instruction set *Thumb* non definisce un'architettura completa: i sistemi *Thumb* dipendono comunque dal fatto che il processore sottostante supporti il completo insieme di istruzioni ARM per operazioni saltuarie (come la gestione degli interrupt) e per procedure con tempi critici, in modo che possano essere eseguite come se fossero codice ARM standard. Il *Thumb* quindi può essere visto come una rappresentazione compressa di un sottoinsieme attentamente selezionato dell'insieme di istruzioni ARM, e perché un processore possa supportarlo può ad esempio necessitare di stadi di decompressione tra la lettura delle istruzioni e la loro decodifica.

L'ARM7TDMI supporta *Thumb* nello stesso modo. Utilizza una pipeline a 3 stadi che, nello stadio di decodifica, presenta un ritardo sufficiente perché le istruzioni possano essere decompresse senza che si creino ulteriori ritardi. Il più recente ARM9TDMI invece ha una più solida pipeline a 5 stadi dove la decompressione richiederebbe uno stadio addizionale, e così i progettisti hanno deciso di decodificare le istruzioni *Thumb* direttamente, piuttosto che decomprimerle in istruzioni ARM e in seguito usare la logica di decodifica standard.

La strategia dell'Amulet3 in questo frangente è una via di mezzo tra questi due estremi. Un vantaggio della natura "elastica" della pipeline asincrona è che fornisce alcune opzioni che non sono disponibili quando si ha a che fare con una struttura "rigida" come una pipeline sincrona.

La logica di decodifica è divisa in 3 aspetti:

1. I selettori degli operandi di registri *time-critical* sono decodificati direttamente.

2. La logica di estrazione dei valori immediati lavora direttamente su codice binario *Thumb*.

3. Il rimanente lavoro di decodifica viene fatto tramite decompressione e decodifica con logica ARM.

11.4.2. Tracciamento del program counter

L'architettura ARM definisce che il program counter (PC) sia memorizzato nel registro 15 (*r15*) del register file, dove può essere scritto e letto come qualsiasi altro registro (anche se in effetti l'architettura pone delle restrizioni al suo utilizzo). L'alta visibilità del PC rappresenta un grande vantaggio per la programmazione; è semplice generare codice indipendente dalla posizione e i valori letterali possono essere prelevati da riserve (*pools*) vicine al codice, utilizzando caricamenti con indirizzi relativi al PC (anche se l'uso di questa pratica è scoraggiato quando si usi un ARM con nucleo Harvard perché porta a un uso inefficiente delle cache). Un utilizzo particolarmente potente del PC viene illustrato dall'inizio e dalla fine delle procedure standard: all'ingresso, una singola istruzione *store* multipla accumula valori di registri di lavoro temporanei sullo stack, assieme all'indirizzo di ritorno, e all'uscita, una singola istruzione *load* multipla ripristina i registri di lavoro e provoca il ritorno della funzione leggendo il PC direttamente dallo stack.

Il valore che compare nel registro 15 non è, comunque, l'indirizzo dell'istruzione corrente. La pipeline ARM a 3 stadi infatti incrementa il PC due volte tra la fase di fetch di un'istruzione e la sua esecuzione, quindi *r15* ritorna il valore PC+8 (dove con

PC ora s'intende l'indirizzo dell'istruzione corrente). Il "+8" corrisponde a un doppio incremento da 4-byte (32 bit), cioè due volte la dimensione di un'istruzione ARM.

La pipeline dell'ARM7TDMI si comporta nello stesso identico modo quando esegue codice *Thumb*, ma poiché le istruzioni *Thumb* sono lunghe solo 2-byte, *r15* ritorna PC+4 (quindi PC + 2×2).

Nell'Amulet3, il prelievo (*fetch*) delle istruzioni procede in modo autonomo, indipendente dal processo di lettura del registro e quindi ciascuna istruzione è associata al proprio indirizzo quando viene inoltrata allo stadio di decodifica (figura 11.3). L'indirizzo può quindi essere modificato (aggiungendo 8 o 4, come abbiamo visto) per generare l'appropriato valore per *r15*. Ciò ci mostra come una conseguenza della precedente implementazione ARM sincrona imponga un costo notevole all'Amulet3.

Esistono diversi altri posti dove è necessario modificare il valore del PC per poter mantenere la compatibilità con l'architettura della pipeline ARM a 3 stadi. Alcuni esempi sono il calcolo dell'indirizzo di ritorno delle procedure (PC+4, o PC+2 per il codice *Thumb*) che nell'Amulet3 non viene calcolato dall'ALU, e l'indirizzo che è registrato quando si verifica un *memory abort* (PC+8, indipendente dalla modalità *Thumb*).

11.5. Organizzazione della memoria

Il nucleo del processore Amulet3 possiede bus separati (*address bus* e *data bus*) per accessi alla memoria istruzioni e a quella dati. Ciò, normalmente, richiede memorie istruzioni separate dalle memorie dati. I sistemi RISC utilizzano spesso un'architettura Harvard modificata dove ci sono cache dati e cache istruzioni separate, ma un'unica memoria principale.

Il controller dell'Amulet3 utilizza RAM a mappatura diretta e non memorie cache per via del costo e del comportamento più deterministico nelle applicazioni in tempo reale. Inoltre, evita memorie dati e memorie istruzioni separate (e le conseguenti difficoltà nel mantenerne la reciproca coerenza) attraverso l'uso di una struttura di memoria di tipo *dual-ported* (a doppia porta, figura 11.4).

Duplicare le porte a livello di bit sarebbe stato troppo oneroso e quindi la memoria è organizzata in otto blocchi da $1KB$, ciascuno dei quali possiede due porte che vengono arbitrate internamente. Quando si verificano accessi ai dati in parallelo ad accessi alle istruzioni, se coinvolgono blocchi diversi possono procedere senza impedimenti, altrimenti uno dei due (quello che non riceve la precedenza a seconda della decisione dell'arbitro) subirà un leggero ritardo mentre attende che l'altro arrivi a completamento.

I conflitti (e il tempo medio di accesso alla memoria) sono ridotti ulteriormente includendo istruzioni separate della dimensione di 4 parole ciascuna e buffer dati, su ciascun blocco RAM. In questo modo, ciascun accesso al blocco prima verifica se il dato cercato risiede nel buffer e, se non lo trova, solo allora interroga la RAM, rischiando un conflitto. Le simulazioni hanno mostrato come il 60% degli accessi per il prelievo delle istruzioni possa essere soddisfatto direttamente dal buffer e, in più, come molti cicli corti (e *time-critical*) vengano eseguiti direttamente da lì.

Questi buffer, in effetti, formano cache di primo livello da 128 bit di fronte ai blocchi RAM. Questa è un'analogia particolarmente adatta quando si osservi che evitare la

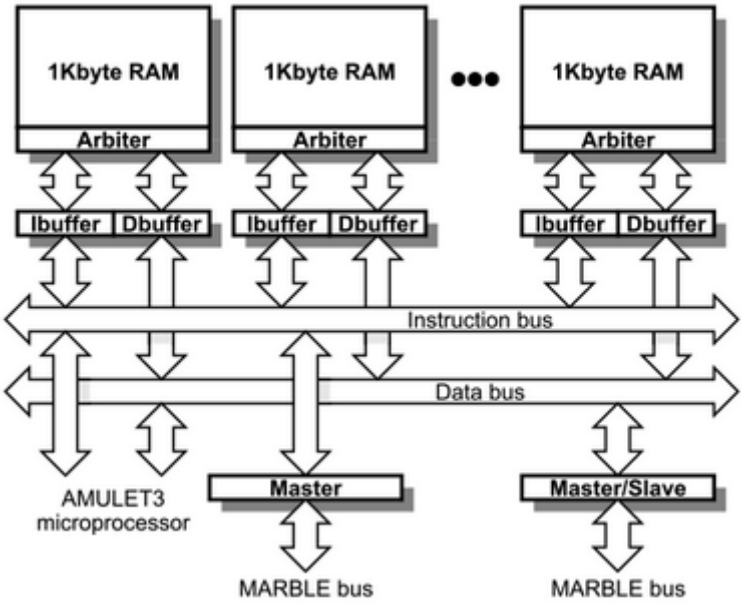


Figura 11.4 – Organizzazione della memoria.

lettura dell'array di RAM produce un ciclo di lettura più veloce. Inoltre, ciò viene automaticamente accettato dalla pipeline asincrona.

11.6. Conclusioni

L'Amulet3 è pensato per essere utilizzato come macro-cella in un nucleo per dimostrare la vitalità commerciale del design asincrono. I particolari benefici che porta sono evidenti quando questioni come il consumo o la compatibilità elettromagnetica (EMC) siano importanti (ad esempio, nelle telecomunicazioni, computer portatili, contactless smart card, etc.).

Per ottenere le prestazioni cercate, l'organizzazione interna dell'Amulet3 è stata rivista in modo significativo rispetto alle precedenti versioni. Gli stalli dovuti alle dipendenze tra dati vengono minimizzati con l'uso di un reorder buffer che, pur non essendo in sé nulla di nuovo, ha richiesto comunque lavoro e idee per poter essere adattato a un ambiente di lavoro asincrono. La simulazione del circuito per quelle parti che sono arrivate a completamento (o quasi) suggerisce che le prestazioni dovrebbero almeno eguagliare quelle dell'ARM9TDMI e il progetto preliminare indica un quantitativo di area necessario per il nucleo che si aggira attorno ai $4mm^2$ (l'ARM9TDMI è di $4.8mm^2$).

La compatibilità all'indietro con i precedenti set di istruzioni è una grossa problematica per ogni nuovo processore. Il set di istruzioni ARM si è molto evoluto negli ultimi 15 anni e ha raggiunto un notevole livello di densità che, se da un lato rappresenta un vantaggio per via della gestione efficiente dei consumi che garantisce, dall'altra rende una nuova implementazione un compito davvero difficile. L'esposizione della pipeline originale a 3 stadi nella definizione dell'instruction set è anch'essa causa di difficoltà per ogni implementazione che utilizzi una pipeline strutturata in

modo diverso. L'Amulet3 mostra come il design asincrono non impedisca che queste difficoltà possano essere superate.

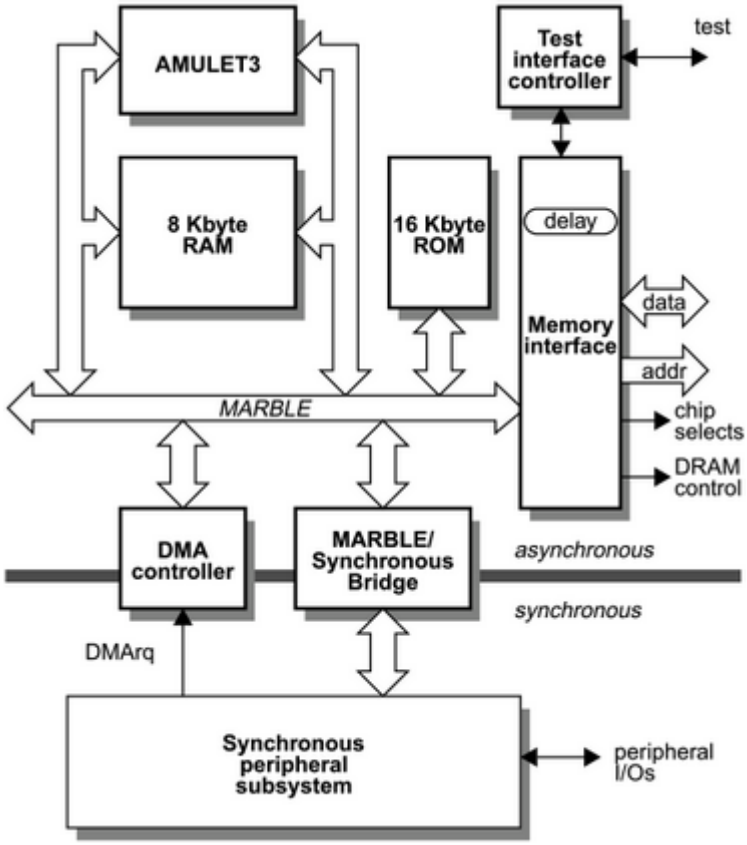


Figura 11.5 – Sottosistema asincrono Amulet3i.

In figura 11.5 possiamo vedere il sottosistema Amulet3i, il primo processore ARM asincrono che sia giunto in produzione, frutto del lavoro congiunto dei progetti OMI-DE2 e OMI-ATOM[3] [53]. L'Amulet3i raggiunge circa i $100MIPS$, cioè poco meno di quanto previsto in fase di simulazione, mentre per i consumi si attesta sui $620MIPS/W$, quindi in linea con l'ARM9.

Possiamo notare come l'organizzazione del sottosistema sia simile a quella dell'Amulet3, ma il livello di complessità raggiunto da questa implementazione è di gran lunga più alto rispetto a quella del suo predecessore e in questa sede abbiamo ritenuto opportuno presentare un modello che fosse un po' più semplice e in qualche modo maggiormente fruibile, provvedendo a fornire le informazioni necessarie per chi sia interessato ad approfondire l'argomento.

[3] OMI/ATOM-ARM: Open Microprocessor systems Initiative - Applications in Telecommunications for OMI Macrocells (ATOM) project. Partecipano a questo progetto la ARM Ltd., la Hagenuk, l'università di Manchester (gruppo APT), la OptionExist Ltd. e la Virata Ltd. Per ulteriori informazioni si veda "http://intranet.cs.man.ac.uk/apt/projects/processors/amulet/ATOM.php".

Conclusioni

Durante la stesura di questo lavoro, abbiamo cercato di raggiungere un compromesso tra estensione e profondità. Ci è parso che organizzare la tesi in due parti, una introduttiva e una più specifica, fosse un buon modo per poter offrire al lettore un quadro generale, arricchito da qualche concetto un po' più approfondito.

Il lavoro di selezione dei molti documenti che abbiamo letto per decidere come orientarci è stato molto interessante. In primis perché interessante è questa tecnologia, che ci piace definire "sfortunata", e in secondo luogo perché abbiamo avuto l'occasione di arricchire la nostra conoscenza con tecniche con le quali è raro poter venire in contatto.

Abbiamo visto che esistono già diversi chip asincroni, alcuni creati *ex novo*, e altri che invece sono il risultato della desincronizzazione di architetture sincrone già esistenti e collaudate. In quasi tutti i casi il modello asincrono presenta vantaggi in molteplici settori, come le prestazioni, l'area occupata dall'implementazione in silicio, il consumo di potenza. Spesso inoltre, il chip asincrono necessita anche di una minore tensione di alimentazione. Ciò si traduce in una minore radiazione elettromagnetica, minori interferenze, minore riscaldamento del circuito, solo per citarne alcuni.

A fronte di questi dati, viene spontaneo domandarsi quali siano i motivi per cui questa tecnologia non riesca a "sbocciare", a trovare il posto che le spetterebbe nella produzione di massa. Purtroppo, mancano ancora strumenti dedicati per il design e di conseguenza ai tecnici manca l'esperienza. I progettisti sono costretti a riparare con i più svariati espedienti, come la modifica dei framework dedicati allo sviluppo di circuiti sincroni, la scrittura di librerie aggiuntive per linguaggi di programmazione estensibili, l'ampliamento delle grammatiche dei compilatori e altro ancora.

Quindi, poiché la progettazione di chip asincroni *puri* è un compito estremamente arduo, al momento l'unico approccio ad avere delle concrete possibilità di sviluppo e diffusione è il Globally Asynchronous, Locally Synchronous: Isole sincrone, ciascuna dotata di un proprio clock, che comunicano tra loro in modo asincrono, mediante handshake. Questa soluzione ibrida risolve il problema del clock-skew perché, suddividendo il circuito in molteplici domini di dimensioni più piccole, elimina di fatto la necessità di propagare un segnale di clock che presenti una differenza di fase trascurabile lungo tutta l'estensione del chip. Inoltre, la progettazione del circuito è agevolata dal fatto che per il design dei domini è sufficiente utilizzare gli usuali strumenti per la progettazione sincrona, che sono diffusi e ben conosciuti.

Resta comunque il problema che, da una parte, molti progettisti non intendono passare al design asincrono finché non ci saranno strumenti adatti e, dall'altra, le case produttrici sono restie a investire tempo e denaro per crearli, data la scarsa entità della domanda. Finché una delle due parti non deciderà di compiere il primo passo, è davvero difficile che la situazione evolva.

Come possiamo trovare allora una motivazione convincente per promuovere lo sviluppo di questo settore? Vediamo un semplice esempio.

All'inizio del 2009 il numero di telefoni cellulari utilizzati nel mondo si attestava sulle 4.6×10^9 unità. Con una popolazione mondiale intorno ai 6.6×10^9 individui, questo implica circa 0.7 telefoni cellulari a testa. Diciamo, senza cercare la precisione assoluta, che il peso medio della batteria di un telefono cellulare sia intorno ai 45g e supponiamo che ogni telefono si trasformi in un dispositivo asincrono, risparmiando l'1% sul consumo di potenza. Ne deriva la seguente equazione:

$$(4.6 \times 10^9) \times \frac{1}{100} \times 45g = 2.07 \times 10^9 g = 2070t$$

Un risparmio, in peso, di 2070 tonnellate di batterie in meno da smaltire ogni x anni (dove x rappresenta la durata media di vita di un telefono cellulare). Considerando quanti sono gli strumenti elettronici che ci circondano, possiamo farci un'idea di quale sarebbe il risultato dell'equazione se includessimo nel calcolo le batterie di orologi, sveglie, palmari, strumentazione biomedica, computer portatili, ebook reader, sensori, smartphone, e così via. Un risparmio forse piccolo per il singolo individuo, che però diventa davvero considerevole se lo proiettiamo sui quasi 7 miliardi di persone che abitano il pianeta.

Un minore consumo energetico si traduce in un minore sfruttamento delle risorse che impieghiamo per alimentare o ricaricare i dispositivi, un minore inquinamento, un impatto ambientale un po' più sostenibile.

Questa lista di benefici, sebbene incompleta, ci sembra essere già di per sé una motivazione più che convincente. In un mondo come il nostro, dominato da un'economia di mercato che segue la legge della domanda e dell'offerta, che impone tempi di produzione strettissimi e che trae motivazione dal guadagno, troviamo auspicabile che si cominci a considerare il *guadagno* non solo come la quantità di denaro (o di potere) ricavabile da un'operazione commerciale, ma anche – e soprattutto – come qualcosa di strettamente correlato al miglioramento della qualità della vita.

Riteniamo che porre l'attenzione sul rapporto tra progresso e inquinamento sia molto importante e, per sottolineare come questa tematica non sia affatto nuova e venga spesso trascurata, lasciamo che a concludere il capitolo siano le parole di Toro Seduto (1831-1890), guerriero e capo indiano della tribù dei Sioux Hunkpapa:

> *«Quando avranno inquinato l'ultimo fiume, abbattuto l'ultimo albero, preso l'ultimo bisonte, pescato l'ultimo pesce, solo allora si accorgeranno di non poter mangiare il denaro accumulato nelle loro banche.»*

Note

Il presente lavoro è stato scritto utilizzando il software open source LyX 1.6.6.1. La versione di LaTeX è quella inclusa in MikTeX 2.8. JabRef 2.6 per la bibliografia BibTeX. Faststone 4.2 per le immagini raster e Google Docs unito a InkScape 0.47 per le immagini vettoriali SVG.

Tutto il materiale tecnico riportato in questa tesi, comprese le immagini, le tabelle, i grafici, etc., proviene dai lavori citati in bibliografia. L'autore non vanta diritti di alcuna natura su quanto riportato. In particolare, i riferimenti per ciascun capitolo sono riportati di seguito.

Capitolo 2: [109]

Capitolo 3: [69, 90]

Capitolo 4: [109, 61]

Capitolo 5: [18]

Capitolo 6: [44]

Capitolo 7: [83]

Capitolo 8: [125]

Capitolo 9: [117]

Capitolo 10: [66]

Capitolo 11: [105, 53]

Ringraziamenti

Innanzitutto voglio ringraziare il mio relatore, il Prof. Carlo Fantozzi, per la gentilezza e la disponibilità che ha dimostrato nei miei confronti, per la guida e i preziosi consigli con cui mi ha aiutato a migliorare la qualità di questo lavoro.

Ringrazio anche i Prof. Geppino Pucci e Carlo Minnaja, per la bellezza delle loro lezioni, per la chiarezza delle loro spiegazioni, per come espongono concetti complessi con una semplicità disarmante, che lascia affascinati.

Ogni storia ha un "cattivo", e nella mia questo ruolo va al Prof. Noè Trevisan. Ciò che non mi uccide mi fortifica, quindi, per essere stato l'ostacolo più duro senza motivo, grazie anche a lui.

Ringrazio tutti i miei parenti, dalla Francia alla Russia, passando per l'Italia.

Ringrazio tutti i miei amici, in particolare: gli arroganti Luca Rossi, Luca Rabboni, Th e Marco. Elena, Nicole, Carletto, il Giok, Simone, SanzMax, TalaMax, il buon Clod, Matteo e Monica, il Maurino, Cristo, il Bucc, Alberto e Silvia, Marica, gli amici del Centro, Danilo l'imperatore di UT, Patrizia e Alessia, Fredrik Bergroth, Vicky e Dan Cowens, Alvise e Chiara, Fausto, Pippo, "la donna della mia vita" Simonetta.

Per gli auguri "magici" e per tanto altro: Marco "Taifu" Beri.

Per i colori: "l'altra donna della mia vita" Lalale Palloncino.

A Lama Luigi e a Lama Giang Chub: བླ་ཁམས་རྡེ་མོ།

Un grazie speciale va a coloro che negli ultimi anni mi sono stati accanto e mi hanno sostenuto nei momenti davvero difficili, quando ne ho avuto più bisogno: Margherita, Graziano, Alessandra e il mio "migliore buon conoscente" Stefano.

Infine, last but not least, ringrazio coloro che mi hanno dato questa vita:

Mio padre, per avermi sempre sostenuto, per l'immensa pazienza, per non essersi mai lasciato abbattere.

Mia madre, per avermi fatto giocare da bambino, per la lezione di coraggio che mi ha dato.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] A. Gara R. Haring G. Kopcsay R. Lembach J. Marcella M. Ohmacht V. Salapura A. Bright, M. Ellavsky. Creating the bluegene/l supercomputer from low-power soc aiscs. *IEEE ISSCC Dig. Tech. Papers, pp. 188–189*, 2005.

[2] M.R. Greenstreet A. Chakraborty. Efficient self-timed interfaces for crossing clock domains. *Proc. 9th IEEE Int'l Symp. Asynchronous Circuits and Systems (ASYNC 03), IEEE CS Press, pp. 78-88*, 2003.

[3] Z. Zilic A. Chattopadhyay. A globally asynchronous locally dynamic system for asics and socs. *Copyright 2003 ACM 1-58113-677-03/03/0004*, 2004.

[4] I. Papaefstathiou A. Efthymiou, J. D. Garside. A low-power processor architecture optimized for wireless devices. *Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)*, 2005.

[5] J. D. Garside A. Efthymiou. Adaptive pipeline structures for speculation control. *in Proc. ASYNC, pp. 46–55, Feb.*, 2003.

[6] G. Russel A. Maamar. Checkbit prediction using dong's code for arithmetic functions. *Proceedings 3rd IEEE On-Line Testing Workshop, Crete, pp 254 – 258,July*, 1997.

[7] H. Wada H. Miyake Y. Nakamura Y. Takebe K. Azegami Y. Himura H. Okano T. Shiota M. Saito S. Wakayama T. Ozawa T. Satoh A. Sakutai T. Katayama K. Abe K. Kuwano A. Suga, T. Sukemura. A 4-way vliw embedded multimedia processor. *IEEE ISSCC Dig. Tech. Papers, pp. 240–241*, 2000.

[8] M. Imai T. Fujii M. Ozaw I. Fukasaku Y. Ueno T. Nanya. A. Takamura, M. Kuwako. Titac2: An asynchronous 32-bit microprocessor based on scalable-delay insensitive model. *Proc. ICCD'97, 288-294, October*, 1997.

[9] C. Wong A.J. Martin, M. Nyström. Three generations of asynchronous microprocessors. *IEEE Design Test of Computers, special issue on Clockless VLSI Design, Nov./Dec.*, 2003.

[10] R. Manohar M. Nystroem P. Penzes R. Southworth U. Cummings A.J. Martin, A. Lines. The design of an asynchronous mips r3000 microprocessor. *Advanced Research in VLSI, pages 164–181*, 1997.

[11] T. Lee D. Borkovic P.J. Hazewindus A.J. Martin, S.M. Burns. The design of an asynchronous microprocessor. *Charles L. Seitz, editor, Proc. International Conference on Advanced Research in VLSI, pages 351–373*, 1991.

[12] U. Michigan ARM Limited & CCCP Research Group. Optimode: Programmable accelerator engines through retargetable customization. *HotChips 16 Conference, IEEE Computer Society Press*, 2004.

[13] A. Steininger B. Rahbaran. Is asynchronous logic more robust than synchronous logic? *IEEE Transactions On Dependable And Secure Computing, Vol. 6, No. 4, Oct-Dec*, 2009.

[14] B.M. Baas. A parallel programmable energy-efficient architecture for computationally-intensive dsp systems. *37th Asilomar Conf. Sig-nals, Systems and Computers, Nov.*, 2003.

[15] R. Manohar C. Kelly IV, V. Ekanayake. Snap: A sensor-network asynchronous processor. *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC'03)*, 2003.

[16] D. Patterson C. Kozyrakis. Vector versus superscalar and vliw architectures for embedded multimedia benchmarks. *Proc. IEEE/ACM MICRO, pp. 283–289, Nov.*, 2002.

[17] R. Manohar C. LaFrieda. Reducing power consumption with relaxed quasi delay insensitive circuits. *15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009.

[18] H. Min C. Chan K. Pun C. Yu, C. Choy. A low power asynchronous java processor for contactless smart card. *IEEE*, 2004.

[19] C.J. Bleakley C.H. Fernández, R.K. Raval. Gals soc interconnect bus for wireless sensor network processor platforms. *GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy. Copyright 2007 ACM 978-1-59593-605-9/07/0003*, 2007.

[20] D.M. Chapiro. Globally-asynchronous locally-synchronous systems. *Doctoral Dissertation, Dept. of Computer Science, Stanford Univ.*, 1984.

[21] L. Lavagno C.P. Sotiriou. De-synchronization: Asynchronous circuits from synchronous specifications. *0-7803-81 82-3/03 IEEE*, 2003.

[22] U. Cummings. Pivotpoint: Clockless crossbar switch for high-performance embedded systems. *IEEE Micro, vol. 24, no. 2, pp. 48-59, Mar. Apr.*, 2004.

[23] M. Martonosi D. Brooks. Dynamically exploiting narrow width operands to improve processor power and performance. *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[24] R. Manohar D. Fang. Non-uniform access asynchronous register files. *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems, April*, 2004.

[25] M. Bolliger M.N. Day H.P. Hofstee C. Johns J. Kahle A. Kameyama J. Keaty Y. Masubuchi M. Riley D. Shippy D. Stasiak M. Suzuoki M. Wang J. Warnock S. Weitzel D. Wendel T. Yamazaki K. Yazawa D. Pham, S. Asano. The design and implementation of a first-generation cell processor. *IEEE ISSCC Dig. Tech. Papers, pp. 184–185*, 2005.

[26] S. Mupparaju D. Wobschall. Low-power wireless sensor with snap and ieee 1451 protocol. *SAS 2008 – IEEE Sensors Applications Symposium*, 2008.

[27] O. Patashnik D.E. Knuth, R.L. Graham. Concrete mathematics. *Addison-Wesley, ISBN 0-201-55802-5*, 2006.

[28] M.N. Sweeting D.J. Barnhart, T. Vladimirova. System-on-a-chip design of self-powered wireless sensor nodes for hostile environments. *IEEEAC paper 1362, Final version, Last updated 12/08/06*, 2006.

[29] H. Dong. Modified berger codes for the detection of unidirectional errors. *IEEE Transactions on Computers, Volume C-33, Number 6, pp 575 – 575, June*, 1984.

[30] P.Y.K. Cheung D.S. Bormann. Asynchronous wrapper for heterogeneous systems. *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors (ICCD 97), IEEE CS Press, 1997, pp. 307-314*, 1997.

[31] P. Endecott. Scalp: A superscalar asynchronous low-power processor. *PhD Thesis, U. Manchester*, 1996.

[32] F.K. Gurkaynak et al. Gals at eth zurich: Success or failure? *Proc. 12th IEEE Int'l Symp. Asynchronous Circuits And Systems (async 06), IEEE Cs Press, 2006, Pp. 150-159*, 2006.

[33] J. Hill et al. System architecture directions for network sensors. *In Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[34] N.J. Boden et al. Myrinet: A gigabit-per-second local area network. *IEEE Micro, vol. 15, no. 1, Jan.-Feb. pp. 29-36*, 1995.

[35] P. Levis et al. The emergence of networking abstractions and techniques in tinyos. *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[36] S.B. Furber. Arm system architecture. *Addison Wesley Longman. ISBN 0-201-40352-8*, 1996.

[37] S.B. Furber. Breaking step: The return of asynchronous logic. *IEE*, 1996.

[38] I.L. Sayers G. Russell. Advanced simulation and test methodologies for vlsi design. *Van Nostrand Reinhold (International), pp251 - 252*, 1989.

[39] D. Geer. Is it time for clockless chips. *Published by the IEEE Computer Society*, 2005.

[40] D.A. Gilbert. Dependency and exception handling in an asynchronous microprocessor. *PhD Thesis, Manchester Univ.*, 1997.

[41] M.R. Greenstreet. Implementing a stari chip. *IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors (ICCD 95), IEEE CS Press, pp. 38-43*, 1995.

[42] K. van Berkel D. Gloor A. Peeters G. Stegmann H. van Gageldonk, D. Baumann. An asynchronous low-power 80c51 microcontroller. *Proc. Async'98, April*, 1998.

[43] L. Cheng C. Chou A. Dixit K. Ho J. Hsu K. Lee J. Wu J. Hart, S. Choe. Implementation of a 4th-generation 1.8 ghz dual-core sparc v9 microprocessor. *IEEE ISSCC Dig. Tech. Papers, pp. 186–187.*, 2005.

[44] K. Cho J. Lee, Y. Hwan Kim. A low power implementation of asynchronous 8051 employing adaptive pipeline structure. *IEEE Transactions On Circuits And Systems-II: Express Briefs, Vol. 55, No. 7, July*, 2008.

[45] W. Fichtner J. Muttersbach, T. Villiger. Practical design of globally-asynchronous locally-synchronous systems. *Proc. 6th Int'l Symp. Advanced Research in Asynchronous Circuits and Systems (ASYNC 00), IEEE CS Press, 2000, pp. 52-59*, 2000.

[46] S.S. Sapatnekar J. Singh. A fast algorithm for power grid design. *ACM 1-59593-021-3/05/0004*, 2005.

[47] S.S. Sapatnekar J. Singh. Partition-based algorithm for power grid design using locality. *IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems, Vol. 25, No. 4, April*, 2006.

[48] C. Morganti M. Dreesen J. Wuu, D. Weiss. The asynchronous 24mb on-chip level-3 cache for a dual-core itanium®-family processor. *IEEE International Solid-State Circuits Conference*, 2005.

[49] D. Jaggar. A performance study of the acorn risc machine. *M.Sc. Thesis, Univ. of Canterbury*, 1990.

[50] D. Jaggar. Advanced risc machines architecture reference manual. *Prentice Hall. ISBN 0-13-736299-4*, 1996.

[51] T.R.N. Rao J.C. Lo, S. Thanawastien. Concurrent error detection in arithmetic and logical operations using berger codes. *Proceedings of 9th Symposium on Computer Arithmetic, pp 233 – 240, Sept.*, 1989.

[52] T.R.N. Rao J.C. Lo, S. Thanawastien. Berger check prediction for array multipliers and array dividers. *IEEE Transactions on Computers, Volume 42, Number 7, pp 892 – 896, July*, 1993.

[53] A. Bardsley D.M. Clark D.A. Edwards S.B. Furber J. Liu1 D.W. Lloyd S. Mohammadi J.S. Pepper O. Petlin S. Temple J.V. Woods J.D. Garside, W.J. Bainbridge. Amulet3i: An asynchronous system-on-chip. *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, 2000.

[54] S.H. Chung J.D. Garside, S.B. Furber. Amulet3 revealed. *Advanced Research in Asynchronous Circuits and Systems, 1999. Proceedings., Fifth International Symposium on*, 1999.

[55] K.R. Cho J.H. Lee, Y.H. Kim. Design of a fast asynchronous embedded cisc microprocessor: A8051. *IEICE Trans. Electron., vol. E87-C, no. 4, pp. 527–534, Apr.*, 2004.

[56] O. Garnica J. Lanchares J.I. Hidalgo G. Miñana S. Lopez J.M. Colmenar, C.E.S. Felipe II. Comparing the performance of a 64-bit fully-asynchronous superscalar processor versus its synchronous counterpart. *IEEE Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06)*, 2006.

[57] P. Korger J. Sparsoe. K.T. Christensen, P. Jensen. The design of an asynchronous tinyrisc tr401 microprocessor core. *Proc. Async'98, April*, 1998.

[58] A.E. Dooply K.Y. Yun. Pausible clocking-based heterogeneous systems. *IEEE Trans. Very Large Scale Integration (VSLI) Systems, vol. 7, no. 4, Dec. 1999, pp. 482-488*, 1999.

[59] R.P. Donohue K.Y. Yun. Pausible clocking: A first step toward heterogeneous systems. *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors (ICCD 96), IEEE CS Press, 1996, pp. 118-123*, 1996.

[60] M. Renaudin L. Fesquet. A programmable logic architecture for prototyping clockless circuits. *0-7803-9362-2005 IEEE*, 2005.

[61] S. Taylor L. Tarazona A. Bardsley L.A. Plana, D. Edwards. Performance driven syntax directed synthesis of asynchronous processors. *ACM 978-1-59593-826-8/07/0009*, 2007.

[62] D.M.G.H. Levy. A wireless, low power, asynchronous, multi-sensor, temperature network. *Massachusetts Institute of Technology*, 2004.

[63] A. Lines. Pipelined asynchronous circuits. *M.S. Thesis, Caltech CS-TR-95-21*, 1995.

[64] A. Lines. Nexus: An asynchronous crossbar interconnect for synchronous system-on-chip designs. *HotInterconnects 11 Conference, IEEE Computer Society Press*, 2003.

[65] A. Lines. Asynchronous interconnect for synchronous soc design. *IEEE Micro, vol. 24, no. 1, Jan.-Feb. 2004, pp. 32-41*, 2004.

[66] A. Lines. The vortex: A superscalar asynchronous processor. *13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'07)*, 2007.

[67] S.A. Jawed M. Gottardi, N. Massari. A 100 microwatt 128x64 pixels contrast based asynchronous binary vision sensor for sensors networks applications. *0018-9200 IEEE*, 2009.

[68] W. Dally M. Horowitz. How scaling will change processor architecture. *IEEE ISSCC Dig. Tech. Papers, pp. 132–133.*, 2004.

[69] S.K. Shukla M. Kishinevsky, K.S. Stevens. Guest editors' introduction gals design and validation. *Copublished by the IEEE CS and the IEEE CASS*, 2007.

[70] B. Baas M. Meeuwsen, O. Sattari. A full-rate software implementation of an ieee 802.11a compliant digital baseband transmitter. *Proc. IEEE Workshop on Signal Processing Systems, pp. 297–301., Oct.*, 2004.

[71] O. Kebichi M. Nicolaidis, V. Castro Alves. Trade-offs in scan path and bist implementations for rams. *0-8186-3360-3/93 IEEE*, 1993.

[72] F. Robin M. Renaudin, P. Vivet. Aspro-216: A standard-cell qdi 16-bit risc asynchronous microprocessor. *Proc. Async'98, April*, 1998.

[73] R. Manohar. Width-adaptive data word architectures. *Proc. International Conference on Advanced Research in VLSI, March*, 2001.

[74] A. Martin. The limitations to delay-insensitivity in asynchronous circuits. *Sixth MIT Conference on Advanced Research in VLSI, W.J. Dally, Ed. MIT Press*, 1990.

[75] A.J. Martin. Synthesis of asynchronous vlsi circuits. *J. Straunstrup, editor, Formal Methods for VLSI Design, pages 237–283. North-Holland*, 1990.

[76] A.J. Martin. The lutonium: Sub-nanojoule asynchronous 8051 microcontroller. *Proc. ASYNC, pp. 14–23*, 2003.

[77] A.J. Martin. Asynchronous techniques for system on chip design. *IEEE 94, No. 6, June 2006 | Proceedings of the IEEE*, 2006.

[78] J. Miller D. Wentzlaff F. Ghodrat B. Greenwald H. Hoffman P. Johnson W. Lee A. Saraf N. Shnidman V. Stumpen S. Amarasinghe A. Agarwal M.B. Taylor, J. Kim. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. *IEEE ISSCC Dig. Tech. Papers, pp. 170–171.*, 2003.

[79] D.G. Messerschmitt. Synchronization in digital system design. *IEEE J. Selected Areas in Communications, vol. 8, no. 8, Oct. 1990, pp. 1404-1419*, 1990.

[80] T. Horseman M. Butler A. Nix M.F. Tariq, Y. Baltaci. Development of an ofdm based high speed wireless lan platform using the ti c6x dsp. *IEEE Int. Conf. Communications, pp. 522–526, Apr.*, 2002.

[81] Sun Microsystems. Java processors - the coming of age. *http://industry.java.sun.com/javanews/stories/story2/0,1072,36281,00.html*, 2001.

[82] Sun Microsystems. Java card 2.2 virtual machine specification. *http://java.sun.com/products/javacard*, 2002.

[83] G.Russell M.Marshall. A low power information redundant concurrent error detecting asynchronous processor. *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, 2007.

[84] S. Jawed N. Massari, M. Gottardi. A 100 microwatt 64x128 pixel contrast-based asynchronous binary vision sensor for wireless sensor networks. *IEEE International Solid-State Circuits Conference*, 2008.

[85] G. Russell A. Yakovlev N. Minas, M. Marshall. Fpga implementation of an asynchronous processor with both online and offline testing capabilities. *14th IEEE International Symposium on Asynchronous Circuits and Systems*, 2008.

[86] C. Farnsworth D.L. Jackson W.A. Lien J. Liu N.C. Paver, P. Day. A low-power, low-noise configurable selftimed dsp. *Proc. Async'98, April*, 1998.

[87] S.B. Furber J.D. Garside J.V. Woods N.C. Paver, P. Day. Register locking in an asynchronous microprocessor. *Proc. ICCD'92, 351-355, Oct.*, 1992.

[88] J. Nurmi. Network-on-chip: A new paradigm for system-on-chip design. *IEEE*, 2005.

[89] David Renshaw P. B. Denyer. Vlsi signal processing; a bit-serial approach. *Addison-Wesley Longman Publishing Co., Inc.*, 1985.

[90] G. Lemieux P. Teehan, M. Greenstreet. A survey and taxonomy of gals design styles. *IEEE*, 2007.

[91] B. Parhami. Approach to the design of parity-checked arithmetic circuits. *Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems and Computers, Volume 2, pp 1084 – 1088, Nov.*, 2002.

[92] N.C. Paver. The design and implementation of an asynchronous microprocessor. *PhD Thesis, Manchester Univ.*, 1994.

[93] G. Russell P.D. Hyde. Assec: An asynchronous self cheching risc based processor. *IEEE Proceedings of the Euromicro Systems on Digital System Design (DSD'04)*, 2004.

[94] W.J. Price. A benchmark tutorial. *IEEE Micro, vol. 9, pp. 28–43, Oct.*, 1989.

[95] G. Theodoropoulos Q. Zhang. Modelling samips: A synthesisable asynchronous mips processor. *IEEE Proceedings of the 37th Annual Simulation Symposium (ANSS'04)*, 2004.

[96] J.E. Smith. R. Canal, A. Gonzàlez. Very low power pipelines using significance compression. *Proc. International Symposium on Microarchitecture, Dec.*, 2000.

[97] C. Kelly IV R. Manohar. Network on a chip: Modeling wireless networks with asynchronous vlsi. *IEEE Communications Magazine 09/01*, 2001.

[98] J. Montanaro R. Witek. Strongarm: A high-performance arm processor. *Proc. IEEE Computer Society Int. Conf.: Technologies for the Information Superhighway (COMPCON), pp. 188–191, Feb.*, 1996.

[99] R. Kivelevich-Carmi R. Zacher. The design and implementation of an asynchronous risc microprocessor. *VLSI Laboratory. Department of Electrical Engineering TECHNION - IIT*, 2000.

[100] P.F. Corbett R.I. Hartley. A digit-serial silicon compiler. *Proceedings of the 25th ACM/IEEE conference on Design automation, pages 646–649. IEEE Computer Society Press*, 1988.

[101] M.J. Meeuwsen T. Mohsenin B.M. Baas R.W. Apperson, Z. Yu. A scalable dual-clock fifo for data transfers between arbitrary and haltable clock domains. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 15, no. 10, pp. 1125–1134, Oct.*, 2007.

[102] R. Damodaran P. Wiley S. Mullinnix J. Leach A. Lell M. Gill A. Rajagopal A. Chachad M. Agarwala J. Apostol M. Krishnan D. Bui Q. An N.S. Nagaraj T. Wolf T.T. Elappuparackal S. Agarwala, M. D. Ales. A 600-mhz vliw dsp. *IEEE J. Solid-State Circuits, vol. 37, no. 11, pp. 1532–1544, Nov.*, 2002.

[103] B. Stackhouse S. Naffziger, T. Grutkowski. The implementation of a 2-core multi-threaded itanium family processor. *IEEE ISSCC Dig. Tech. Papers, pp. 182–183, 592*, 2005.

[104] L. Goudge S. Segars, K. Clarke. Embedded control problems, thumb, and the arm7tdmi. *IEEE Micro, 15(5):22-30, Oct.*, 1995.

[105] D.A. Gilbert S.B. Furber, J.D. Garside. Amulet3 a high performance self timed arm microprocessor. *Proc. International Conf. Computer Design (ICCD), Oct. 98*, 1998.

[106] J.D. Garside N.C. Paver J.V. Woods. S.B. Furber, P. Day. The design and evaluation of an asynchronous microprocessor. *Proc. ICCD'94, 217-220, Oct.*, 1994.

[107] S. Temple J. Liu P. Day N.C. Paver S.B. Furber, J.D. Garside. Amulet2e: An asynchronous embedded controller. *Proc. Async'97, 290-299, April*, 1997.

[108] J.N. Seizovic. Pipeline synchronization. *Proc. Int'l Symp. Advanced Research in Asynchronous Circuits and Systems (ASYNC 94), IEEE CS Press, pp. 87-96*, 1994.

[109] J. Sparsø. Principles of asynchronous circuit design: A system perspective. *Kluwer Academic Publishers (Boston/Dordrecht/London)*, 2001.

[110] I.E. Sutherland. Micropipelines. *Communications of the ACM, 32 (6): 720-738, June*, 1989.

[111] S.M. Nowick T. Chelcea. Robust interfaces for mixed-timing systems. *IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol. 12, no. 8, pp. 857-873 Aug.*, 2004.

[112] C. Jen T. Lin. Cascade—configurable and scalable dsp environment. *Proc. IEEE ISCAS, pp. 26–29*, 2002.

[113] P. Nilsson T. Olsson. A digitally controlled pll for soc applications. *IEEE J. Solid-State Circuits, vol. 39, no. 5, pp. 751–760, May*, 2004.

[114] V. Akella T. Werner. Asynchronous processor survey. *IEEE*, 1997.

[115] R. Manohar V. Ekanayake, C. Kelly IV. An ultra low-power processor for sensor networks. *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct.*, 2004.

[116] K. van Berkel. Handshake circuits: An asynchronous architecture for vlsi programming. *vol. 5, Intl. Series on Parallel Computation, Cambridge University Press*, 1993.

[117] R. Manohar V.N. Ekanayake, C. Kelly IV. Bitsnap: Dynamic significance compression for a low-energy sensor network asynchronous processor. *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)*, 2005.

[118] B. Towles W. Dally. Route packets, not wires: On-chip interconnection networks. *IEEE Int. Conf. Design Automation, pp. 684–689, Jun.*, 2001.

[119] S.B. Furber W.J. Bainbridge. Asynchronous macrocell interconnect using marble. *Proc. Async'98, April*, 1998.

[120] I.W. Jones S.M. Fairbanks I.E. Sutherland W.S. Coates, J.K. Lexau. Fleetzero: An asynchronous switching experiment. *Seventh International Symposium on Asynchronous Circuits and Systems*, 2001.

[121] P. Chen J. Chen Z. Li Y. Liu, G. Xie. Designing an asynchronous fpga processor for low-power sensor networks. *IEEE*, 2009.

[122] R. York. Branch prediction strategies for low power microprocessor design. *MSc Thesis, Manchester Univ.*, 1994.

[123] B.M. Baas Z. Yu. High performance, energy efficiency, and scalability with gals chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 17, No. 1, 01/09*, 2009.

[124] R. Apperson O. Sattari M. Lai J. Webb E. Work T. Mohsenin M. Singh B. Baas Z. Yu, M. Meeuwsen. An asynchronous array of simple processors for dsp applications. *in IEEE ISSCC Dig. Tech. Papers, pp. 428–429.*, 2006.

[125] R.W. Apperson O. Sattari M. Lai J.W. Webb E.W. Work D. Truong T. Mohsenin B.M. Baas Z. Yu, M.J. Meeuwsen. Asap: An asynchronous array of simple processors. *Ieee Journal Of Solid-state Circuits, Vol. 43, No. 3, 03/08*, 2008.