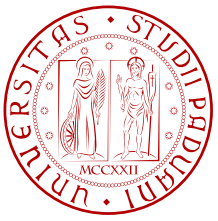


Università di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Elettronica

Implementazione di un encoder LDPC DVB-T2 su piattaforma FPGA

(Implementation of DVB-T2 LDPC encoder on FPGA)

Laureando:
Davide Biadene

Relatore:
Daniele Vogrig

Anno Accademico 2010/2011

Sommario

Questo documento descrive la realizzazione di un encoder LDPC secondo lo standard DVB-T2 su di una piattaforma FPGA della famiglia Xilinx Virtex 4.

Si è ripreso un lavoro svolto meno di anno fa da parte di ricercatori dell'Università di Padova nell'ambito dei test del nuovo standard per la televisione digitale terrestre effettuati su un dispositivo della Lyrtech.

La scelta del FPGA garantisce un forte vantaggio nella realizzazione del prototipo del modulatore per la sua facilità di riprogrammazione e per le potenzialità che ormai hanno raggiunto.

Indice

Capitolo 1

Introduzione

1.1 Scopo della tesi

Nell'ambito dello sviluppo dei nuovi standard di comunicazione, il DVB-T2, relativo al Digitale Terreste di seconda generazione, implementa un modulatore molto performante sotto il punto di vista della robustezza del segnale. Questo si ottiene utilizzando una combinazione di codici LDPC e BCH per la codifica, seguita da un interleaving esteso in più fasi della modulazione e il suo trasferimento in una costellazione ruotata per la modulazione OFDM.

Meno di un anno fa all'Università di Padova alcuni ricercatori avviarono uno studio del nuovo standard su di una piattaforma di sviluppo.

Il dispositivo utilizzato per i test di Figura 1.1 della Lyrtech (SFF SDR - *Small Form Factor Software-Defined Radio*) è composto essenzialmente da tre moduli: un ricetrasmettitore a banda larga dai 200MHz ai 3,8GHz coprendo sia gli spettri del segnale televisivo in UHF e VHF sia quelli del segnale Wi-Fi e Wi-MAX; un modulo di conversione dei segnali analogici in digitali, utilizzando convertitori ADC a 14 bit, DAC a 16 bit e un FPGA Virtex 4 il loro controllo; un modulo per l'elaborazione dei segnali contente memorie, dispositivi di controllo della potenza e soprattutto un altro FPGA Virtex 4 modello XC4VSX35 che sarà l'elemento nel quale andrà a inserirsi l'argomento di questa tesi.

L'oggetto che si vuole sviluppare è l'encoder LDPC DVB-T2 all'interno del FPGA della piattaforma di sviluppo. In realtà, il lavoro si svolgerà solo a livello di simulazioni e su un dispositivo minore della stessa famiglia e per questo l'obiettivo sarà di rendere il progetto il più portabile possibile su diverse piattaforme.

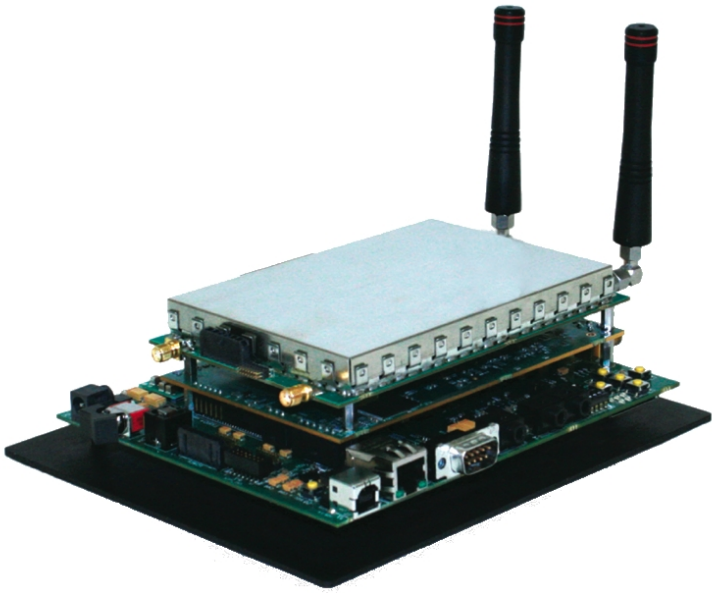


Figura 1.1: SFF SDR (*Small Form Factor Software-Defined Radio*) Development Platforms della Lyrtech.

1.2 Struttura della tesi

Nei primi due capitoli si forniscono le nozioni base per comprendere i contenuti di questa tesi. In particolare, nel primo si prendono in considerazione i dispositivi utilizzati, mentre nel secondo il concetto di codifica di canale e dei codici correttori lineari, tra i quali vi è anche quello LDPC.

Nel terzo capitolo si approfondisce lo standard DVB-T2 concentrandosi maggiormente su quanto riguarda il modulatore e la codifica FEC che viene effettuata sui vari stream audio e video precedentemente compressi, studiando anche la struttura che si vuole realizzare.

Nel quarto capitolo, poi, si analizza l'intera implementazione del dispositivo descritto in linguaggio VHDL, analizzando le varie entity che lo compongono e il loro funzionamento riportato alla struttura precedentemente citata.

Nel quinto capitolo seguono le simulazioni effettuate con iSim per la verifica del corretto funzionamento e l'utilizzo di programmi in Matlab per il controllo della codifica eseguita.

Nell'ultimo capitolo, infine, si citano i risultati raggiunti considerando anche le limitazioni della struttura progettata e fornendo eventuali soluzioni e spunti per sviluppi ulteriori.

1.3 FPGA (Field Programmable Gate Array)

Agli inizi degli anni '70 incominciarono a comparire i primi componenti logici programmabili, i SPLD (*Simple Programmable Logic Device*), che consentivano

di sviluppare funzioni solamente combinatorie e anche abbastanza semplici.

Esistono varie soluzioni di SPLD in base al grado di programmazione che si vuole avere sul dispositivo:

PROM (*Programmable ROM*), basato su di una ROM programmabile ed è composto essenzialmente da un piano AND predefinito, che svolge la funzione di decodificatore di riga, e da un piano OR programmabile. Il piano AND fornisce in uscita tutti i 2^N minterm degli N ingressi, mentre sul piano OR è possibile eseguire sulle M uscite la somma di un numero ristretto di minterm per creare le funzioni volute. Vista la struttura è possibile immaginarla come una ROM con 2^N indirizzi e larghezza di parola M, da cui il nome.

PLA(*Programmable Logic Array*), in cui sono presenti i due piani AND e OR, ma sono entrambi programmabili, quindi realizzano una maggior programmabilità del dispositivo permettendo di scrivere funzione ai minimi termini e anche di poter riutilizzarli in più funzioni.

PAL(*Programmable Array Logic*), in tal caso il piano AND è programmabile, mentre quello OR è già predefinito, quindi è possibile gestire solo i termini prodotto della funzione.

GAL(*Generic Array Logic*), consiste in un dispositivo PAL con l'aggiunta che il piano di OR comanda un altro gruppo di AND configurabili che possono gestire un buffer bidirezionale in uscita.

La programmazione avviene mediante tecnologia a fusibile o EPROM o E^2PROM. La prima tecnica è irreversibile perché elimina il collegamento fisico tra ingresso e array di porte, mentre le altre due sono programmabili più volte mediante l'utilizzo di MOS a gate flottante come collegamento. La differenza consiste che nella tecnologia EPROM la cancellazione avviene mediante irradiazione UV e quindi su tutto il dispositivo, mentre nella E^2PROM avviene elettricamente a discapito dell'aggiunta di un transistor di controllo.

Nel 1984 comparvero i primi CPLD che consistono in una matrice di SPLD fittamente interconnessi in modo da garantire la completa connettività tanto da renderli un fallimento, sia per il consumo di potenza, che per la bassa velocità dovuti entrambi all'enormità di linee presenti.

Nel medesimo anno per la prima volta la Xilinx propose un nuovo tipo di componente programmabile: l'FPGA (*Field Programmable Gate Array*). Sono realizzati con tecnologia CMOS con memoria di configurazione SRAM, cioè i collegamenti sono gestiti da transistor che sono abilitati o meno da una memoria SRAM di tipo volatile comportando quindi l'aggiunta di una non volatile per

salvare i bit di configurazione. In alcuni casi si trovano anche con programmazione ad antifusibile.

L'architettura è simile ai CPLD, non si mira alla completa connettività, ma piuttosto a un numero più elevato di blocchi programmabili più semplici, immersi in una griglia di interconnessioni con matrici di commutazione come in Figura 1.2.

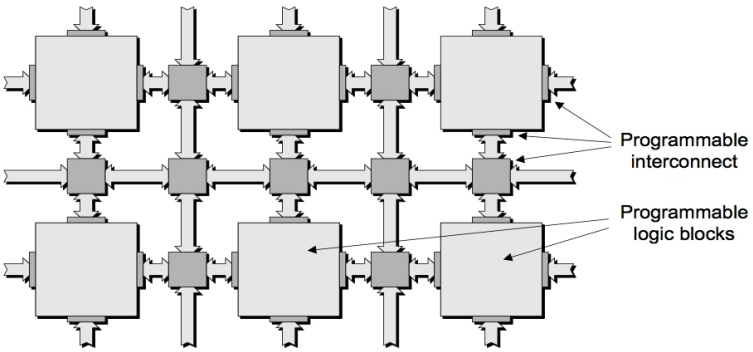


Figura 1.2: Schema della struttura interna di un FPGA (www.mentor.com).

I blocchi logici non contengono più solo porte logiche, ma includono anche multiplexer, registri e LUT (*Look-Up Table*) che possono realizzare qualsiasi funzione da 3 a 5 ingressi, mediante una memoria SRAM e un multiplexer. Il principale svantaggio è che, mentre nei CPLD i ritardi erano determinabili a priori dato che si conosceva la struttura dei blocchi logici e delle interconnessioni, negli FPGA non lo sono poichè dipendono dal piazzamento dei blocchi logici normalmente eseguito da un programma dedicato [11].

Ogni FPGA della Xilinx è composto da un numero variabile di CLB (*Configurable Logic Blocks*) che rappresentano i blocchi logici inseriti all'interno della griglia di interconnessioni. Ogni CLB è connessa sia a livello locale che globale con le altre presenti nel dispositivo. Il design delle connessione è costruito in modo tale da garantire linee veloci per le connessioni a breve raggio per poi diradarle man mano che ci si allontana dalla CLB in esame. Questo suggerisce che il circuito che si viene a implementare normalmente si compatti in blocchi adiacenti in modo da preferire i collegamenti veloci rispetto a quelle globali che son più lente.

Sono presenti altri blocchi essenziali quali moduli IOB che rappresentano l'interfaccia tra il dispositivo e il mondo esterno, buffer di clock globali e moduli DCM che gestiscono la propagazione dei segnali di sincronismo, e la possibilità di svariati componenti aggiuntivi, quali memorie, moltiplicatori, DSP.

Per questo motivo si passa ora alla trattazione dei soli FPGA della famiglia Virtex 4, in particolare del modello XC4VSX25 utilizzato in questo progetto.

1.3.1 Virtex 4

Ogni CLB è suddiviso ulteriormente in quattro *Slice*, due di tipo M e due di tipo S, come mostrato in Figura 1.3.

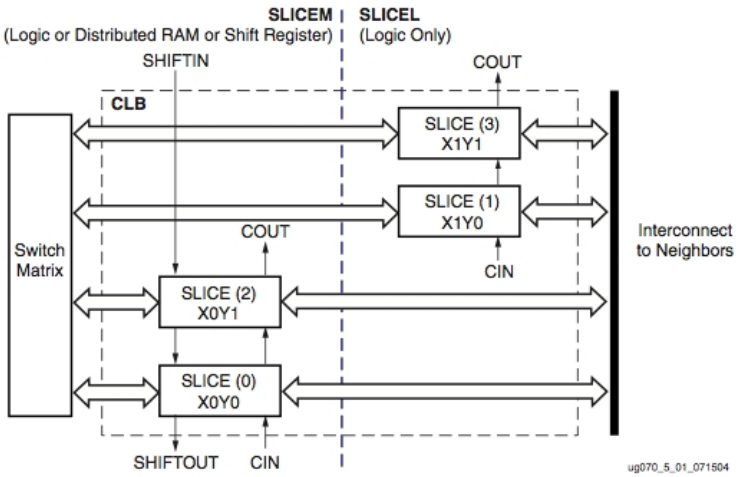


Figura 1.3: Schema di una CLB [13].

Ogni *Slice* contiene 2 LUT da 4 ingressi, 2 multiplexer, 2 Flip-Flop configurabili, 2 porte AND e 2 XOR, come mostrato nel dettaglio dalla Figura 1.4.

Dalla figura si posso osservare più multiplexer, ma in realtà non sono gestibili dall'utente perché non hanno nessun ingresso di selezione, infatti questi vengono gestiti dalla memoria di programmazione che li configura in base al circuito da realizzare. I multiplexer disponibili all'utente, in realtà, servono per incrementare il numero di ingressi di una funzione logica combinando sia le uscite delle LUT della stessa *Slice*, che quelle delle altre contenute nell'intera CLB realizzando così funzioni a 8 ingressi.

I due elementi di memoria sono completamente configurabili nel renderli Latch o Flip-Flop, nel fronte di lavoro, nell'inizializzazione ed anche nel tipo di segnali di controllo.

La presenza delle due porte è utile nel caso si dovessero implementare dei sommatori Carry Look Ahead data la predisposizione anche di linee adibite alla propagazione del riporto.

Le LUT non sono uguali nei due tipi di *Slice*: in quelle S posso essere utilizzate solo come funzione logica, mentre in quelle M posso essere adattate a memoria SRAM, oppure come un shift register.

La memoria realizzata tramite LUT prende il nome di Memoria Distribuita a differenza delle *Block RAM* che sono veri e propri banchi di memoria presenti nell'FPGA.

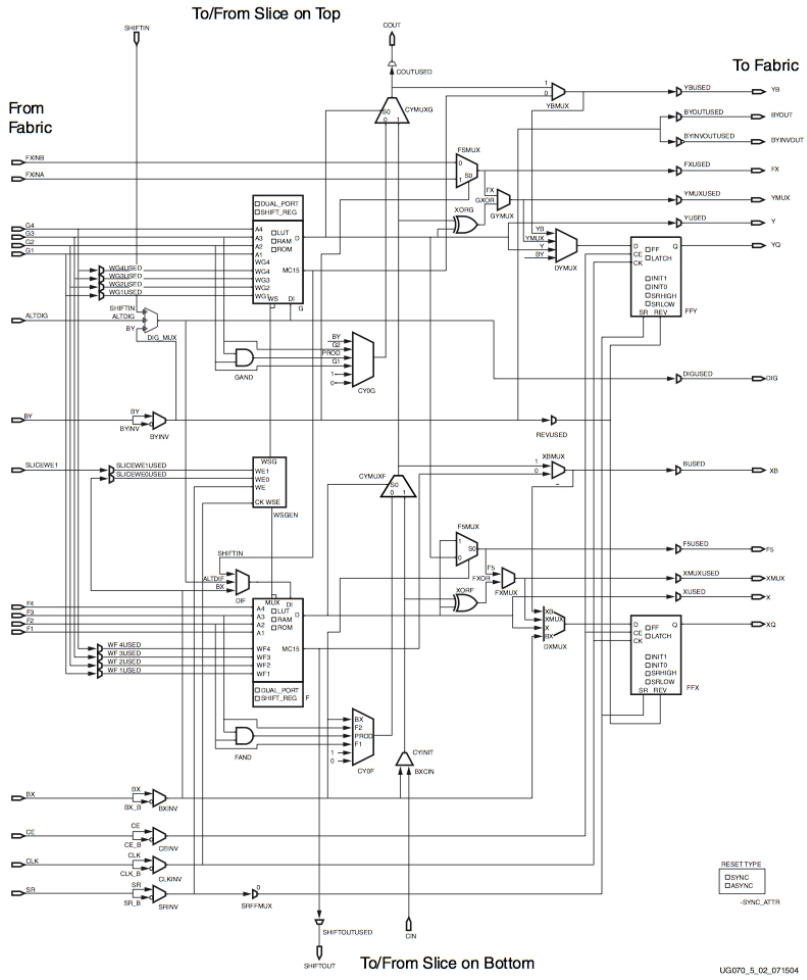


Figura 1.4: Schema di una SLICEM di una Virtex 4 [13].

Il modello utilizzato fa parte in particolare della famiglia Virtex-4 SX la quale mira particolarmente ad alte prestazioni per processi di segnali digitali che richiedono l'utilizzo di DSP di cui ne è la più fornita tra gli altri modelli. Ogni DSP contiene all'interno un moltiplicatore 18×18, un sommatore e un accumulatore.

In Figura 1.5 riportiamo un estratto della tabella di [12] dove vi sono riportate tutte le caratteristiche dei vari modelli. Notiamo che le differenze tra il modello utilizzato nel progetto e quello predisposto dalla Lyrtech.

Device	Configurable Logic Blocks (CLBs)(1)				XtremeDSP Slices(2)	Block RAM		DCMs	PMCDs	PowerPC Processor Blocks	Ethernet MACs	RocketIO Transceiver Blocks	Total I/O Banks	Max User I/O
	Array(3) Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)							
XC4VSX25	64 x 40	23,040	10,240	160	128	128	2,304	4	0	N/A	N/A	N/A	9	320
XC4VSX35	96 x 40	34,560	15,360	240	192	192	3,456	8	4	N/A	N/A	N/A	11	448
XC4VSX55	128 x 48	55,296	24,576	384	512	320	5,760	8	4	N/A	N/A	N/A	13	640

Figura 1.5: Risorse per i vari modelli di FPGA della famiglia Virtex 4 [12].

Capitolo 2

Codici correttori

Questo capitolo accenna ad elementi di telecomunicazione intenti a fornire una minima conoscenza sugli aspetti della codifica di canale e del loro scopo di esistere. Ci si focalizzerà sui codici di tipo lineare in modo da comprendere in che ambito si inseriscono i codici LDPC.

2.1 Introduzione

Nell'ambito delle telecomunicazioni, quando si parla di informazione il suo significato è leggermente diverso rispetto a quello che si è abituati a pensare.

L'informazione è una misura della quantità informativa di un evento che avviene con una certa probabilità. Si pensi che eventi con minor probabilità contengono maggior informazione rispetto a quelli che avvengono più frequentemente. Per questo motivo è possibile pensare ad un evento come una variabile aleatoria discreta x che assume dei valori all'interno di un alfabeto A con probabilità P[A].

Altra descrizione probabilistica, che si può effettuare, è indicare il valor medio dell'informazione contenuta dall'evento, cioè il valor atteso di x, che prende il nome di entropia.

Il rapporto tra l'entropia dell'evento in esame e quella di una variabile aleatoria uniforme discreta definita sul medesimo alfabeto si chiama efficienza. Il suo valore è unitario quando gli eventi di A sono equiprobabili, altrimenti è un numero non negativo inferiore all'unità. Si definisce anche come ridondanza la misura del complementare dell'efficienza rispetto all'unità.

In un sistema di trasmissione l'intento è quello di avere un'efficienza elevata e quindi una ridondanza molto bassa perché altrimenti si sta inserendo del canale dell'informazione superiore al necessario.

Per porre rimedio a questo problema esistono le **codifiche di sorgente** che consistono essenzialmente nel rielabolare il messaggio entrante in uno che abbia un'efficienza più elevata. Questo risulta molto importante soprattutto nelle

operazioni di memorizzazione, perché va detto che maggiore è l'informazione più grande è il numero di bit associato all'evento e quindi comporta un aumento della dimensioni in memoria. Da qui risulta che questo tipo di codifica venga chiamata anche compressione dati.

Una volta che si è diminuita la ridondanza della sorgente in modo da descrivere la medesima informazione, ma con un'occupazione molto minore a livello di bit, non è comunque possibile trasmetterla all'interno di un canale rumoroso se non prima di aver eseguito un altro tipo di codifica che lo renda adatto e robusto alla trasmissione: **la codifica di canale**.

2.2 Codifica di canale

La codifica di canale è una tecnica di trasformazione di messaggio in un altro che è molto più robusto agli errori introdotti dallo stesso canale. La robustezza di un codice è ottenuta a discapito dell'aggiunta di ridondanza mediante l'inserimento di opportuni simboli, detti di parità, che sono determinati solo dal messaggio informativo. Nel ricevitore si ha, quindi, una stima del messaggio in trasmissione ed è possibile riconoscere che non sia valido e in alcuni casi anche correggerlo.

Potrebbe risultare assurdo aver prima eliminato la ridondanza nella codifica di sorgente per poi reinserirla all'interno dello stesso codice, ma mentre prima lo scopo era quello di eliminarla dalla descrizione statistica degli eventi, ora si ha l'intento di aggiungerla per adattare il messaggio alle caratteristiche del canale e ad un maggior livello di robustezza di fronte ai possibili errori introdotti dal rumore.

Vi sono principalmente due modi in cui un ricevitore può lavorare in presenza di un messaggio non valido:

Automatic Retrasmission Query (ARQ), in tal caso il sistema richiede la ritrasmissione dell'intero messaggio o della parte nel quale è stato identificato l'errore.

Foward Error Correction (FEC), consiste nel sostituire il messaggio corrotto con uno che il trasmettitore può aver inviato e che si avvicina maggiormente a quello ricevuto. Questo può risolvere solo parzialmente gli errori introdotti dal canale.

La prima tecnica necessita di un canale di ritorno dal ricevitore al trasmettitore con la quale inviare la richiesta di ritrasmissione, il che limita molto il loro utilizzo, mentre il secondo necessità solo di un canale semplice unidirezionale. Tuttavia, i sistemi basati sul FEC sono meno affidabili dato che scelgono arbitrariamente la parola da sostituire e quindi potrebbero commettere anche più errori,

invece di ridurli a differenza di una ritrasmissione che comporterebbe comunque un maggior flusso di dati con relativa latenza.

Per questi motivi si utilizzano tecniche ARQ nelle trasmissioni dati nelle quali non ci sono limiti ristretti per il ritardo, mentre si utilizzano quelle FEC per applicazioni in tempo reale, quali il *Video Broadcasting* (DVB).

La ridondanza introdotta dalla codifica si esprime attraverso il *code rate* che è il rapporto tra il numero di simboli in ingresso rispetto al numero di simboli in uscita che sono trasmessi in un dato intervallo.

2.3 Codici a blocco

Si consideri di voler inviare un messaggio e di scomporlo in blocchi separati composti ciascuno da k simboli, formando così una parola d'informazione rappresentato da una sequenza binaria di k simboli definendo quindi un insieme A^k. Questa viene codificata in una parola di codice composta da n simboli tramite una legge μ_c che associa ad ogni delle 2^k parole in ingresso, una delle 2^n parole possibili in uscita. L'insieme di tutte le possibili parole che compongono il codominio della funzione di codifica $C = \mu_c(A^k)$ è chiamato (n, k) *block code* C definito sull'insieme A^n. Una volta trasmessa, la parola viene riconvertita nell'insieme A^k mediante la funzione inversa in codifica $\mu_d = \mu_c^{-1}$ dal decodificatore. Il rapporto tra k e n prende il nome di *code rate*.

Un codice C si dice sistematico se ogni parola ottenuta dalla codifica ha come prefisso la parola contenente l'informazione.

2.4 Codici lineari

Se l'insieme di parole composto da n simboli binari formano uno spazio lineare, allora si ha a che fare con codici lineari. Per realizzare uno spazio lineare basta trasformare l'alfabeto $A = \{0, 1\}$ in un campo, cioè introdurre le operazioni di somma e di prodotto interne, cioè che restituiscano sempre un valore all'interno del campo stesso. Per l'operazione di somma si dovrà utilizzare l'operatore di or esclusivo mentre per il prodotto l'operatore and, in tal modo si viene a creare un campo $(A, \cdot, +)$ che viene indicato con $\mathbb{F}$. Nel caso di un campo $(A^n, \cdot, +)$ verrà definito come $\mathbb{F}^n$.

Un (n, k) *block code* C si definisce lineare se è un sottospazio lineare di $\mathbb{F}^n$, cioè se

$$\forall \gamma_1, \gamma_2 \in C, \qquad \gamma_1 + \gamma_2 = \gamma_1 - \gamma_2 \in C$$

Questa definizione richiede che la funzione μ_c sia biunivoca, cioè associ un solo elemento di A^k ad un elemento di C e quindi quest'ultimo deve avere 2^k parole di codice.

Matrice generatice. Ogni matrice $\mathbf{G} \in \mathbb{F}^{n \times k}$ le cui colonne definiscono una base per il codice lineare C si definisce matrice generatrice di C. In tal caso la matrice deve avere grado k e quindi le colonne devono essere linearmente indipendenti. La matrice che si può generare non è unica, infatti eseguendo le semplici operazioni sulle colonne è possibile generare altre matrici che hanno lo stesso grado.

In un codice sistematico si ha che k parole di codifica dovranno avere come prefisso la parola di lunghezza k $\beta_1 = [100...0]$, $\beta_2 = [010...0]$, $\beta_k = [000...1]$ in modo da riportare all'inizio della parola codificata l'informazione, ed essendo anche linearmente indipendenti, si può pensare di costruire la matrice generatrice nel seguente modo:

$$\mathbf{G} = \begin{bmatrix} \mathbf{I}_k \\ \mathbf{A} \end{bmatrix}$$

dove $\mathbf{I}_k$ è la matrice d'identità $k \times k$ e $\mathbf{A}$ è una matrice arbitraria $(n-k) \times k$ definita in $\mathbb{F}$.

Essendo un codice lineare è possibile ottenere la codifica utilizzando semplicemente il calcolo matriciale, infatti definedendo $\mathbf{c}$ come la codifica della parola in ingresso $\mathbf{b}$, si ha che:

$$\mathbf{c} = \mu_c(\mathbf{b}) = \mathbf{G}\mathbf{b}$$

Data la forma della matrice $\mathbf{G}$, si può asserire che sarà la sottomatrice $\mathbf{A}$ a comportare le caratteristiche di codifica, per esempio la minima distanza tra le parole, cioè in numero di bit minimo per cui differiscono due parole.

Matrice di controllo di parità. La matrice di parità di un codice lineare C è una matrice $\mathbf{H}$ $(n-k) \times n$ il cui spazio nullo coincide con C, cioè:

$$\gamma \in C \Leftrightarrow \mathbf{H}\gamma = \mathbf{0}$$

Questo permette di definire se una parola sia corretta o meno solo moltiplicandola per $\mathbf{H}$ e controllando che il risultato sia un vettore di lunghezza $n-k$ nullo.

Altra principale caratteristica che deve soddisfare la matrice $\mathbf{H} \in \mathbb{F}^{(n-k) \times n}$ di un codice C definito con la matrice generatrice $\mathbf{G} \in \mathbb{F}^{n \times k}$ è che $\mathbf{HG} = \mathbf{0}$.

Da queste considerazioni si può ottenere che la forma della matrice $\mathbf{H}$ può essere definita similmente a quella data per $\mathbf{G}$:

$$\mathbf{H} = \begin{bmatrix} -\mathbf{A} & \mathbf{I}_{n-k} \end{bmatrix}$$

Questa trattazione risulta essere molto basilare dal punto di vista dei contenuti, ma fornisce quel minimo richiesto per comprendere la spiegazione dei codici lineari a blocco LDPC [9].

Capitolo 3

Codici LDPC nello standard DVB-T2

In questo capitolo si descrivere in modo essenziale lo standard DVB-T2 per fornire almeno una conoscenza basilare dell'intero sistema nel quale il dispositivo sviluppato in questa tesi viene inserito e si conclude con una focalizzazione sui codici LDPC utilizzati nello standard.

3.1 Lo standard DVB-T2

Dall'inizio degli studi sulla digitalizzazione dei segnali analogici, iniziata da C.E. Shannon tra gli anni '40 e '50, la trasmissione attraverso segnali digitali ha ormai preso il sopravvento su quella analogica.

I principali punti di forza sono dovuti essenzialmente alla robustezza al rumore e alle capacità rigenerative del segnale digitale, per non parlare di un migliore sfruttamento della banda dei canali trasmissivi. Per questi motivi il passaggio dalla televisione analogica a quella digitale (DTT, *Digital Terrestrial Television*) può essere considerata come la sua naturale evoluzione verso una sempre maggior quantità e qualità dei servizi disponibili.

Per garantire un'uniformità nelle trasmissioni all'interno della stessa area geografica sono sorti degli enti di standardizzazione, tra cui l'ETSI (*European Telecommunications Standards Institute*) in Europa, che hanno come intento l'unificazione e l'omologazione dei metodi di comunicazioni.

Nell'ambito delle trasmissioni video, nel 1993 nasce il DVB-Project (*Digital Video Broadcasting*) da una precedente alleanza di 250 compagnie europee formatasi due anni prima, che ha lo scopo di sviluppare un sistema per portare la televisione digitale nelle case, fino ad allora improponibile dato i costi di sviluppo.

Escono vari standard per i diversi mezzi trasmissivi, DVB-C per trasmissioni su cavi coassiali e DVB-S per quelle satellitari, ma solo nel 1997 venne pubblicato il DVB-T per la prima volta, cioè quello per le trasmissioni terrestri, e venne testato un anno dopo a Singapore [6].

Il DVB-T(*Digital Video Broadcasting Terrestrial*) è uno standard molto complesso perché mira ad essere robusto anche in situazioni in cui il rumore e la larghezza di banda posso risultare molto difficili. Nello standard si specifica la struttura dei frame, la codifica di canale, la modulazione e anche le linee guida per la sua implementazione. Nonostante questo, risulta molto flessibile e permette di configurare collegamenti per un gran numero di servizi, dalla HDTV(*High-Definition Television*) ai sistemi multicanale SDTV(*Standard Definition Television*) [3].

Il nuovo standard DVB-T2, viene approvato e pubblicato nel 2008 e offre maggiore efficienza, robustezza e flessibilità. Introduce tecniche di modulazione e di codifica che lo rendono molto efficiente per le trasmissioni terrestri di audio, video e servizi dati sui più svariati dispositivi, rendendolo il migliore tra gli altri sistemi DTT e anche il più adottato nel mondo come si vede in Figura 3.2.

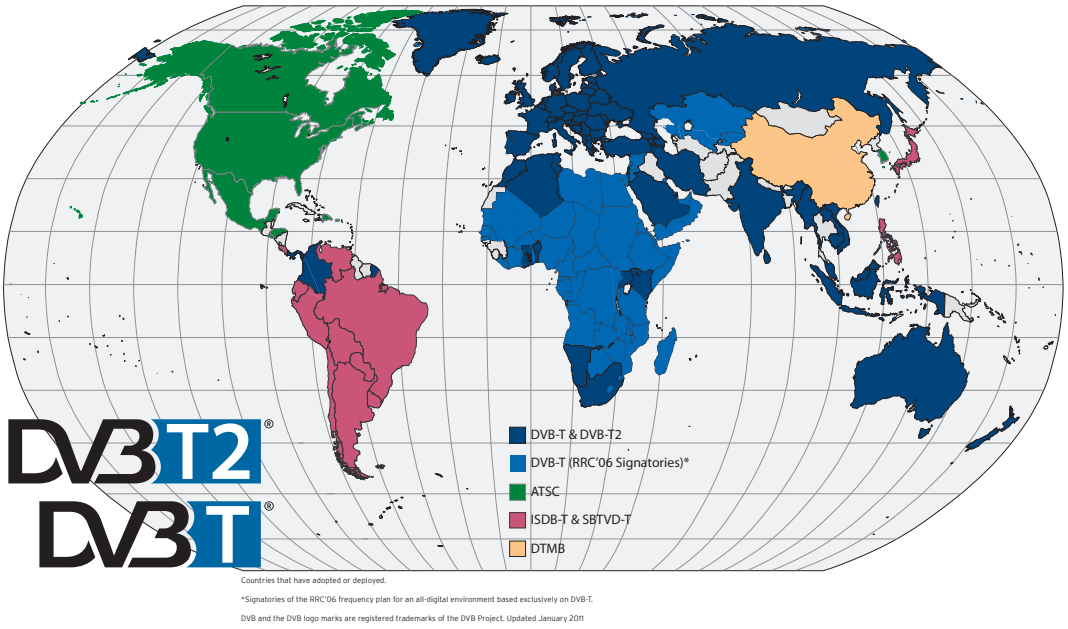


Figura 3.1: Mappa degli standard di trasmissione DTT [5].

Come il suo predecessore usa una modulazione OFDM (*orthogonal frequency division multiplex*), ma in aggiunta utilizza la stessa codifica a correzione d'errore degli standard DVB-S2 e DVB-C2, cioè una codifica **LDPC** combinata con quella **BCH** generando un segnale molto robusto [4].

Vi sono ulteriori modifiche rispetto al DVB-T, ma riguardano aspetti che non interessano la trattazione di questo documento. Si ritiene comunque interessante accennare le principali caratteristiche dello standard che lo rendono il più performante al mondo.

Costellazione ruotata che fornisce una notevole aggiunta di robustezza al segnale.

Codifica di Alamouti viene utilizzata nelle trasmissioni wireless per implementare metodi *trasmitted diversity*, cioè dove vi siano più sorgenti indipendenti che emettono la medesima informazione.

Interleaving esteso in più fasi della modulazione.

Future Extension Frame che consentono allo standard di essere compatibile in futuro.

PLP multipli che permettono di ottenere l'adattamento migliore su ogni servizio in base al canale trasmissivo sul quale questo verrà trasmesso in modo da venir incontro alle condizioni del ricevitore.[1]

3.1.1 Architettura

L'intero sistema DVB-T2 può essere suddiviso in più sottosistemi che vengono composti in Figura 3.2 per realizzare l'intera catena di trasmissione.

Come è possibile osservare è composto da tre moduli in trasmissione e due in ricezione. Si riporta ora una breve descrizione dei moduli in modo da abbozzare quale sia la funzione di ogni singolo.

SS1: Codifica e multiplexing. Include la generazione dei MPEG-2 *Transport Stream*[2] (TS) e/o dei *Generic Stream* (GSE). Normalmente la codifica video, e possibilmente anche quella audio, sono gestite con bit rate variabile con un controllo comune che assicura un bit rate totale costante considerando tutti gli stream presi insieme.

SS2: Basic T2-Gateway. In uscita riporta una sequenza di pacchetti T2-MI i quali contengo ognuno un *Baseband* frame (BBFRAME), eventuali flussi ausiliari, o un *signalling message* di tipo L1 che contiene informazioni di codifica, modulazione, struttura riguardo il PLP utilizzato, o di tipo SFN che riguarda gli aspetti di trasmissione isofrequenziale adottata. Tali pacchetti

[1] Si parla PLP (*Phisical-Layer Pipe*) quando lo stream in ingresso viene processato in modo tale da creare una corrispondenza biunivoca con i canali dati nel modulatore.

[2] Flusso dati.

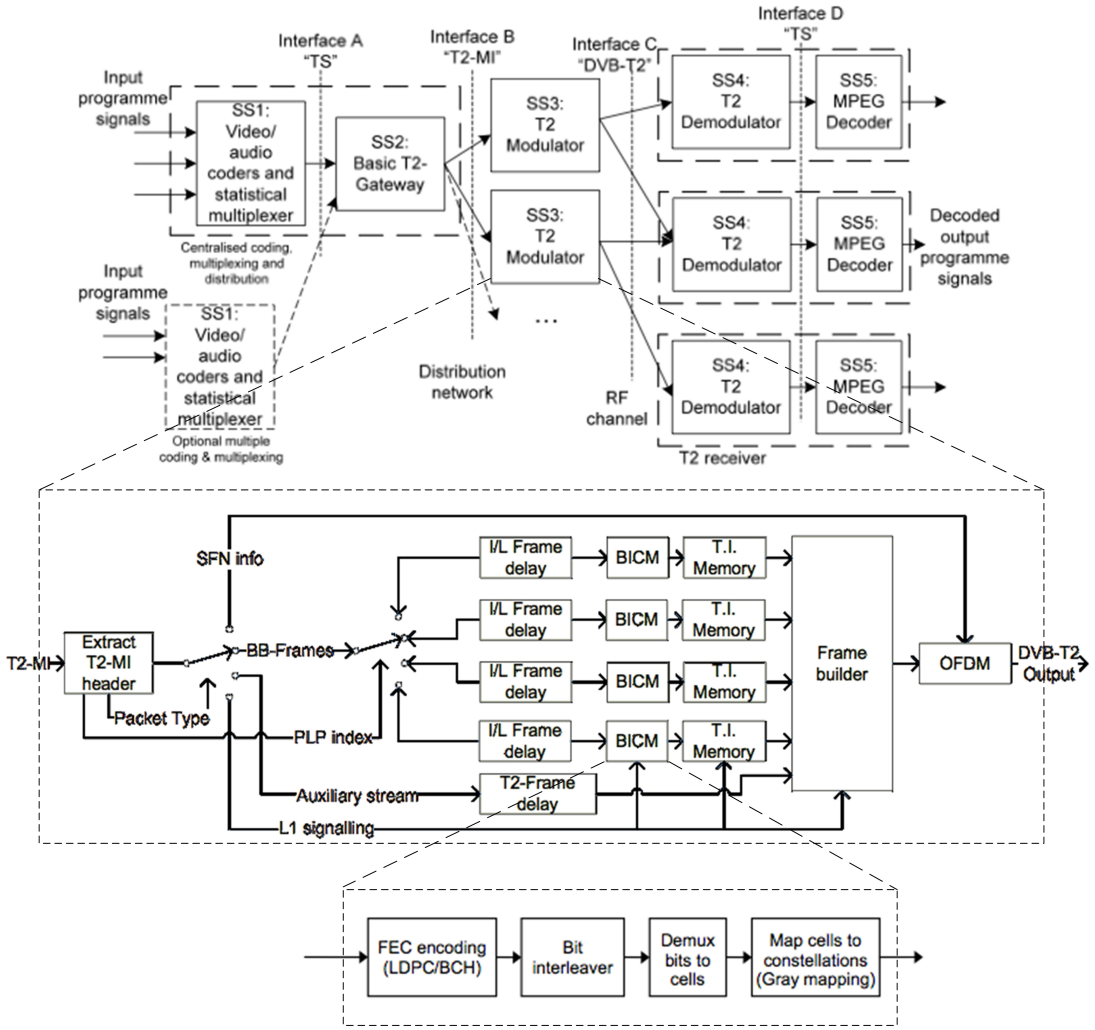


Figura 3.2: Architettura del DVB-T2 con dettaglio del modulatore e del BICM.

contengono tutte le informazioni richieste per descrivere sia il contenuto che il tempo di emissione del frame, e un singolo T2-MI stream è inviato a uno o più modulatori nella rete.

SS3: Modulatore DVB-T2. Utilizza il BBFRAME e le informazione di formazione del medesimo per creare i frame DVB-T2 ed emetterli con le relative tempistiche per avere una corretta sincronizzazione.

SS4: Demodulatore DVB-T2. Riceve un segnale RF da uno o più trasmettitori nella rete e riporta in uscita un solo flusso dati identico a quello uscente da SS1.

SS5: Stream decoder. Riceve il flusso dati e in uscita si ottiene la decodifica dei segnali audio e video.

Maggior attenzione va riportata al modulatore perché, come indicato nella Figura 3.2, esso contiene al suo interno la codifica FEC nel quale è inserito

l'argomento di questa tesi. Per questo motivo sembra doveroso svilupparne maggiormente la sua trattazione tralasciando gli aspetti che non riguardano questo documento.

Il modulatore DVB-T2. Il modulatore, come mostra il dettaglio di Figura 3.2, riceve lo stream proveniente dal modulo *T2-Gateway* ed una volta estratte le informazioni del tipo di pacchetto che sta per ricevere le indirizza verso un ulteriore modulo interno. Nella maggior parte dei casi sono BBFRAME e quindi viene inviato ad un buffer (*I/L Frame delay*) per attendere la ricezione del segnale *L1 signalling* che contiene i parametri per i successivi blocchi d'elaborazione.

Uno volta che questo viene ricevuto, il modulo BICM (*Bit-Interleaved Coding and Modulation)* provvede alla codifica del BBFRAME entrante secondo la struttura evidenziata nella figura.

Innanzitutto, si esegue la codifica FEC (*Foward Error-Correction*) basata sulla concatenazione di codici BCH e LDPC per poi applicare un *interleaving*[3] al FECFRAME appena generato. Questa ultima operazione è necessaria perché sia i codici LDPC nel DVB-T2, che le costellazioni multilivello non hanno protezione d'errore uniforme. Le prestazioni di un codice LDPC con una costellazione multi livello dipendono fortemente dalla corrispondenza tra bit della codifica e bit della costellazione rendendo la protezione ancora meno uniforme, ma facilmente risolvibile grazie l'utilizzo dell'*interleaving* e di un demux.

L'ultimo passaggio all'interno del modulo BICM è la mappatura sulla costellazione mediante codifica Gray, la cui caratteristica principale è quella di avere i punti a minore distanza che differiscono di un solo bit, per poi passare alla costruzione del frame e alla modulazione OFDM e di seguito essere inviati tramite canali RF[2].

Dopo questa semplice panoramica dello standard DVB-T2 si comprende come in realtà sia molto complesso nella sua struttura. Punto centrale della trattazione di questa tesi è lo sviluppo dell'encoder LDPC presente nel modulatore appena descritto su piattaforma FPGA, quindi si prosegue fornendo alcuni nozioni su questi codici.

3.1.2 Codici LDPC

Nel 1962, Robert G. Gallager scrive la tesi di dottorato al MIT su un nuovo tipo di codici a correzione d'errore dal titolo *"Low-Density Parity-Check Code"*.

[3]Tecnica di elaborazione di un segnale digitale con il quale dispongono i dati i maniera non contigua migliorando le prestazioni in termini di rilevazione e correzione d'errore nel caso di errori multipli consecutivi.

Questi codici sono sistematici e definiti da una matrice che contiene più '0' che '1', le cosiddette matrici sparse (da cui anche il nome *Low-Density*). In particolare, un *(n, j, k) low-density code* è un codice che ha lunghezza n e con una matrice dove ogni colonna contiene un numero fissato j di '1' e ogni riga ne contiene invece k. In tal caso il codice si dice regolare. In Figura 3.3 si riporta un esempio con $n = 20$, $j = 3$ e $k = 4$.

$$
\begin{array}{cccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Figura 3.3: Esempio di matrice di un codice LDPC (20,3,4) fatta da Gallager [7].

Un importante parametro da analizzare in un codice è la minima distanza[4] tra le diverse parole del codice e Gallager dimostra che per $j \geq 3$ essa cresce linearmente con la lunghezza del codice n per j e k costanti, mentre per $j = 3$ questa cresce logaritmicamente. Alla distanza è fortemente correlata la probabilità d'errore in ricezione perché trasmettendo su di un canale normalmente rumoroso è possibile la corruzione della parola.

In [7] vengono proposti due tipi di decodificatori: il primo effettua una decisione su ogni digit cambiando il suo valore nel caso che più di un numero prefissato di equazioni data dalla matrice di parità che lo contengono siano errate. Il processo viene iterato su ogni bit modificato finché la parola non viene decodificata. Il secondo, invece, si basa sul calcolo della probabilità condizionata che un determinato bit sia a '1' ed è condizionata dall'intero insieme di bit che lo contengono. Quest'ultimo richiede molta più elaborazione rispetto al primo, ma garantisce una bassa probabilità d'errore.

L'aspetto fondamentale di questi codici è che hanno prestazioni prossime alla capacità di canale definita dal teorema fondamentale della teoria dell'informazione introdotto da Claude Shannon nel 1948. Shannon definisce un limite massimo alla velocità con la quale si può inviare l'informazione all'interno di un canale rumoroso, ma dimostra che con un'opportuna codifica è possibile inviarla con una velocità inferiore scegliendo arbitrariamente un limite superiore alla probabilità d'errore. Nella sua dimostrazione non definisce un metodo per realizzare tale

[4]Con distanza si indica il numero di bit per le quali differiscono due parole.

codifica, ma solo che esiste e quindi questo limite è stato ritenuto per molto tempo irraggiungibile.

I codici LDPC non hanno avuto molto successo all'inizio perché la loro decodifica richiedeva unità d'elaborazione molto complesse e quindi vennero abbandonati per prediligere altri tipi di codice, quali i codici convoluzionali (P. Elias, 1955), i codici ciclici (E. Prange, 1957), i codici BCH (A. Hocquenghem, 1959, e R.C. Bose e D.K. Ray- Chaudhuri, 1960).

Nel 1993, C. Berrou, A. Glavieux, P. Thitimasjshimae sviluppano i Turbo codici che hanno prestazioni migliori dei precedenti e si avvicinano notevolmente al limite di Shannon avendo comunque una struttura molto semplice sia nel codificatore che nel decodificatore.

Intanto lo sviluppo tecnologico permette una potenza di calcolo sempre maggiore tanto che nel 1994 vengono riscoperti i codici LDPC, ma la loro comparsa avverrà molto più tardi con un incremento notevole del loro utilizzo segnato dalle seguenti tappe: DVB-S2 (2005), WiMAX (2005), WiFi (2007), DVB-T2 (2008) e DVB-C2(2010) [1].

Codice LDPC DVB-T2. I codici di questo standard differiscono in base alla lunghezza del frame, indicata con N_{ldpc} e in base alla lunghezza dell'informazione da codificare, K_{ldpc}. Va ricordato che prima del codificatore **LDPC** è presente un codificatore **BCH** che accoda al BBFRAME un segmento di codice BCHFEC di lunghezza $N_{bch} - K_{bch}$. In Figura 3.4 si riporta la composizione dei vari frame che generano il FECFRAME con l'aggiunta di quello inerente all'encoder LDPC.

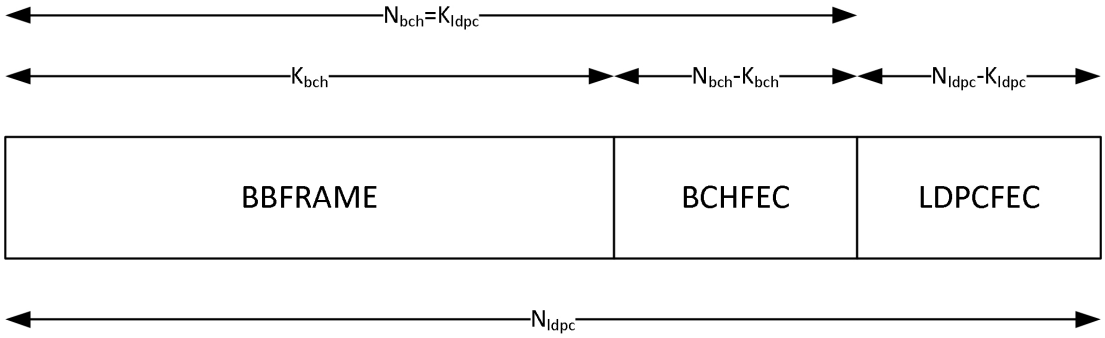


Figura 3.4: Struttura di un FECFRAME.

Esistono sole due dimensioni di FECFRAME, quella con $N_{ldpc} = 16200$ o con $N_{ldpc} = 64800$. Il primo permette un controllo più fine del bit-rate a discapito di una maggior computazione e performance peggiore rispetto al secondo. Ulteriore distinzione avviene secondo il *code rate*, cioè il rapporto fra la quantità di informazione e la lunghezza del codice inviato che corrisponde al rapporto K_{ldpc} su N_{ldpc}.

Di seguito si riportano le Tabelle 3.1 e 3.2 dove viene indicato come identificativo della codifica il relativo code rate. Nella seconda si indica anche quello reale perché non è equivalente a quello usato come identificativo e si può notare come questo abbia una granulosità di valori più fine.

Tabella 3.1: Parametri di codifica per *normal FECFRAME* $N_{ldpc} = 64800$.

LDPC Code	**LDPC Uncoded Block** K_{ldpc}	$N_{ldpc} - K_{ldpc}$
1/2	32400	32400
3/5	38880	25920
2/3	43200	21600
3/4	48600	16200
4/5	51840	12960
5/6	54000	10800

Tabella 3.2: Parametri di codifica per *short FECFRAME* $N_{ldpc} = 16200$.

LDPC Code Identifier	**Effective LDPC Rate**	**LDPC Uncoded Block** K_{ldpc}	$N_{ldpc} - K_{ldpc}$
1/4	1/5	3240	12960
1/2	4/9	7200	9000
3/5	3/5	9720	6480
2/3	2/3	10800	5400
3/4	11/15	11880	4320
4/5	7/9	12600	3600
5/6	37/45	13320	2880

Gli elementi che caratterizzano i codici LDPC utilizzati in questo standard possono essere così elencati:

sistematici, la parola da codificare contenente l'informazione si trova come prefisso della parola codificata;

irregolari, il peso associato a ogni colonna della matrice di parità $\mathbf{H}$, j, non è costante;

struttura ciclica di H nella parte dell'informazione, la sottomatrice A_1 formata dalle prime K_{ldpc} colonne è sì fatta da soddisfare questa proprietà:

$$A_1[360u + j, i] = A_1[360u + j + 1, (i + Q_{ldpc}) \mathrm{mod} (N_{ldpc} - K_{ldpc})]^5$$

[5] L'operatore mod restituisce il resto ottenuto dalla divisione intera tra il primo e il secondo operando.

con $u = 0, 1, ..., \frac{K_{ldpc}}{360} - 1$, $j = 0, 1, ..., 359$ e $i = 0, 1, ..., N_{ldpc} - K_{ldpc} - 1$.

Considerando la matrice A_1 come affiancamento di matrici di 360 colonne, risulta che in ogni gruppo colonne adiacenti sono uguali a meno di una traslazione di Q_{ldpc} righe.

struttura a scala di H nella parte di parità, la sottomatrice A_2 formata dalle ultime $N_{ldpc} - K_{ldpc}$ colonne è costruita nel seguente modo:

$$A_2[0, 0] = 1$$

$$A_2[i, i] = A_2[i, i - 1] = 1 \text{ con } i = 1, 2, ..., N_{ldpc} - K_{ldpc} - 1$$

Date queste sue caratteristiche è stato possibile identificare un algoritmo che richiedesse un hardware molto più semplificato rispetto all'esecuzione completa del calcolo matrice, detto di Richardson - Urbanke, dai suoi ideatori. Nella sezione successiva viene riportato secondo lo schema descritto in [10].

3.2 Algoritmo implementato

Data la natura dei codici lineari, si potrebbe pensare che le operazioni da eseguire siano quelle relative al calcolo matriciale e quindi di dover implementare anche strutture di notevole complessità. Le matrici generatrici **G** dei codici LDPC sono delle matrici sparse, cioè con bassa densità di 1, e quindi normalmente vengono rappresentate mediante la scrittura delle posizioni in cui esse sono non nulle. Ulteriore proprietà è che tali matrici presentano determinate caratteristiche tra le colonne. Infatti, considerando gruppi di 360 colonne adiacenti esse sono differenti solo per una traslazione costante di un determinato numero di righe. Queste due importanti proprietà delle matrici generatrici rendono possibile implementare l'algoritmo in modo iterattivo, riducendo in tal modo le risorse utilizzate e soprattutto eliminando le operazioni il cui risultato sarebbe a priori nullo. Come precedentemente descritto, l'encoder LDPC dato in ingresso un blocco d'informazione di lunghezza K_{ldpc}, $I = (i_0, i_1, ..., i_{K_{ldpc}-1})$, lo codifica in una parola Λ di lunghezza N_{ldpc}, dove:

$$\Lambda = (\lambda_0, \lambda_1, ..., \lambda_{N_{ldpc}-1}) = (i_0, i_1, ..., i_{K_{ldpc}-1}, p_0, p_1, p_{N_{ldpc}-K_{ldpc}-1}) \tag{3.1}$$

Nel documento [10, par. 6.1.2] viene descritto l'algoritmo da eseguire per ottenere la codifica nel caso di *normal FECFRAME*, tuttavia estendibile anche nel caso di *short FECFRAME*. Si riporta a seguire una sintesi delle operazioni da effettuare.

1. Inizializzare tutti i bit di parità a zero:

$$p_0 = p_1 = p_2 = ... = p_{N_{ldpc}-K_{ldpc}-1} = 0$$

2. Sommare ai bit di parità indicati dalla prima riga della tabella della codifica considerata, il primo bit d'informazione i_0.

3. Per i successivi 359 bit, $i_m, m = 1, 2, ...359$, sommare ai bit di parità di indice

$$\{x + m \bmod 360 \cdot Q_{ldpc}\} \bmod (N_{ldpc} - K_{ldpc}) \tag{3.2}$$

il bit i_m, dove con x si intendono gli indirizzi già utilizzati per il primo bit i_0, mentre Q_{ldpc} è una costante che dipende dal code rate utilizzato, come mostra la Tabella 3.3.

4. Per il successivo bit d'informazione, i_{360}, si esegue la somma con i bit indicati nella seconda riga della medesima tabella. In egual modo per i seguenti 359 bit, $i_m, m = 361, 362, ..., 719$ gli indici si ottengono usando ancora 3.2, dove x corrisponde nuovamente ai valori utilizzati per il bit i_{360}.

5. Per ogni gruppo composto da 360 bit, una nuova riga della tabella è utilizzata per trovare gli indirizzi dei bit di parità da sommare.

6. Dopo che tutti i K_{ldpc} bit d'informazione sono processati, si deve eseguire sequenzialmente la seguente procedura partendo con $i = 1$

$$p_i = p_i \oplus p_{i-1}, i = 1, 2, ..., N_{ldpc} - K_{ldpc} - 1 \tag{3.3}$$

7. Il valore finale di $p_i, i = 0, 1, ..., N_{ldpc} - K_{ldpc} - 1$ è uguale al bit di parità p_i in 3.1.

3.3 Struttura implementata

Individuato l'algoritmo che si vuole implementare, è necessario scegliere una struttura di base che permetta di eseguirlo mantenendo un buon compromesso tra prestazioni e utilizzo delle risorse. Questo principio di progettazione è stato utilizzato per non essere limitati dalle caratteristiche del dispositivo FPGA sul quale si andrà a implementarlo, soprattutto nell'ottica che nel medesimo oggetto andranno inseriti altri componenti per costruire l'intero modulatore digitale descritto nel documento [10]. Normalmente questi dispositivi si trovano inseriti tra due strutture di tipo FIFO (*First Input First Output*) : entrambe svolgono una funzione di

Tabella 3.3: Valori di Q_{ldpc} per entrambi i tipi di frame.

Code Rate	Q_{ldpc} ($N_{ldpc} = 64800$)	Q_{ldpc} ($N_{ldpc} = 16200$)
1/4		36
1/2	90	25
3/5	72	18
2/3	60	15
3/4	45	12
4/5	36	10
5/6	30	8

interfaccia tra lo *bit stream* in ingresso o in uscita e, rispettivamente, la velocità effettiva di elaborazione dell'encoder o dei moduli a seguire. In questa trattazione non ci si occuperà del dimensionamento o dell'implementazione di tali moduli, ma verranno utilizzati solo nelle simulazioni per verificare il corretto funzionamento. In Figura 3.5 viene riportata l'esplosione del blocco *FEC encoding (LDPC/BCH)* di Figura 3.2 per poter comprendere la disposizione delle FIFO rispetto all'encoder nel progetto corrente.

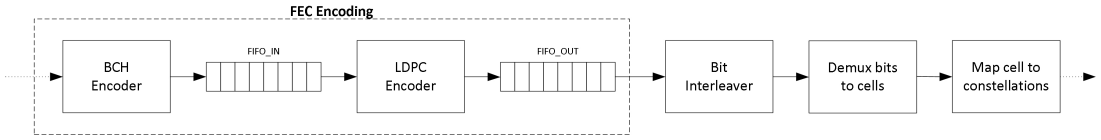


Figura 3.5: LDPC Encoder e FIFO all'interno dello schema di Figura 3.2.

Da una prima analisi del problema risulta sicuramente necessario aver a disposizione degli elementi di memoria sia per contenere le tabelle e i parametri relativi al tipo di code rate utilizzato, sia per elaborare il vettore dei bit di parità che viene modificato ad ogni iterazione dell'algoritmo. Per questi motivi, si può pensare di implementare per i primi delle memorie di tipo ROM nel quale inserire tali valori, mentre per il secondo una RAM nella quale l'indicizzazione dei bit è intrinseca nella struttura.

Sarà necessario anche predisporre un'unità aritmetica che vada a determinare il risultato della formula per il calcolo degli indirizzi. Ad un primo impatto l'espressione sembra che richieda l'utilizzo di moltiplicatori e divisori, ma è possibile ridurne la struttura a un contatore e dei sommatori integrati in una struttura pipeline. Per ulteriori approfondimenti si rimanda al capitolo successivo.

Dall'analisi dell'algoritmo si può notare che è possibile identificare due strutture separate nel quale l'unico elemento in comune è il vettore di bit di parità, cioè la RAM. Nella prima parte dell'elaborazione, l'accesso alla RAM è eseguito

secondo lo schema imposto dalle tabelle e dall'unità di calcolo e in uscita si hanno una versione ritardata dei bit in ingresso, mentre nella seconda si ha solo una procedura sequenziale interna al vettore che restituisce in uscita i bit di parità della parola codificata. È possibile, perciò, pensare di costruire un'unità di controllo mediante una FSM(*Finite State Machine*) che oltre a gestire i vari code rate, effettuare la gestione di situazioni critiche quali FIFO in ingresso vuota o in uscita piena, gestisca anche i collegamenti della RAM alle due strutture individuate. Da queste riflessioni è possibile ottenere uno schema di base che si riporta in Figura 3.6.

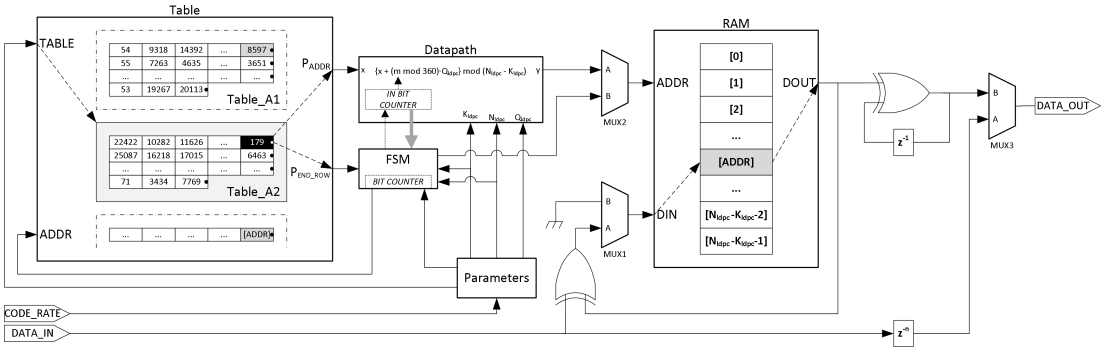


Figura 3.6: Schema riassuntivo della struttura implementata.

Algoritmo sviluppato mediante la struttura di Figura 3.6. Prima di proseguire con l'implementazione dei vari blocchi, si riscrive l'algoritmo 3.2 visto dalla struttura implementata in modo che il lettore possa comprendere il flusso di elaborazione dei dati che si intende realizzare successivamente.

Fase di inizializzazione L'utente fornisce all'encoder il *code rate* da utilizzare e il blocco *Parameters* fornisce agli altri elementi i parametri necessari alla codifica, sezione della tabella, dimensione della RAM, e la FSM invia loro i segnali di reset se necessario.

1. Si supponga che la RAM sia inizializzata nulla, cioè

$$RAM[ADDR] = 0, 0 \leq ADDR \leq N_{ldpc} - K_{ldpc} - 1$$

Fase di acquisizione del messaggio Si procede alla ricezione dei bit d'informazione in ingresso e alla loro elaborazione nella RAM. In questa fase tutti i multiplexer propagano i segnali indicati con A. In uscita si riporta il bit del messaggio letto, ritardato di n periodi in base alle esigenze strutturali del modulo.

2. All'acquisizione del primo bit, i_0, la FSM invia un segnale al *Datapath*, unità aritmetica del modulo, per incrementare il *in bit counter* che aggiorna il valore di m nella formula, in questo caso $m = 0$. Di seguito indirizza *Table*, il modulo che contiene tutte le tabelle, in modo da restituire in uscita la successione degli elementi della prima riga (essendo le tabelle memorizzate come vettori, vi sarà un flag che indicherà alla FSM che il valore corrente è l'ultimo della riga). Gli indirizzi calcolati vengono forniti alla RAM che legge il loro contenuto e li riscrive con l'operazione di xor tra il valore letto e i_0. In sintesi si esegue la seguente operazione:

$$RAM[Table[1,:]] = RAM[Table[1,:]] \oplus i_0$$

   Si indica con Table[n,:] il vettore di elementi della n-esima riga della tabella selezionata da *Parameters*.

3. Per i prossimi 359 bit in ingresso, $i_m, m = 1, 2, ..., 359$ la FSM richiama le medesime operazione svolte nel punto precedente, utilizzando sempre la prima riga. Il contenuto della RAM sarà aggiornato come segue:
   $ADDR_{(1,m)} = \{Table[1,:] + m \bmod 360 \cdot Q_{ldpc}\} \bmod (N_{ldpc} - K_{ldpc})$
   $RAM[ADDR_{(1,m)}] = RAM[ADDR_{(1,m)}] \oplus i_m, m = 1, 2, ..., 359$

4. Quando si arriva al 360° bit acquisito, i_{369}, il *in bit counter* asserisce un flag verso la FSM che indica che i prossimi bit in ingresso $i_m, m = 360, 361, ..., 719$ dovranno essere computati con la riga successiva.
   $ADDR_{(2,m)} = \{Table[2,:] + m \bmod 360 \cdot Q_{ldpc}\} \bmod (N_{ldpc} - K_{ldpc})$
   $RAM[ADDR_{(2,m)}] = RAM[ADDR_{(2,m)}] \oplus i_m, m = 360, 361, ..., 719$

5. Nello stesso modo, per ogni gruppo n-esimo di 360 bit consecutivi si andrà a considerare una nuova riga della tabella.
   $ADDR_{(n,m)} = \{Table[n,:] + m \bmod 360 \cdot Q_{ldpc}\} \bmod (N_{ldpc} - K_{ldpc})$
   $RAM[ADDR_{(n,m)}] = RAM[ADDR_{(n,m)}] \oplus i_m, m = 360(n-1), .., 360n - 1$
   La FSM interrompe l'acquisizione di nuovi bit quando e quindi la relativa procedura dopo l'elaborazione del bit K_{ldpc}-esimo bit, $i_{K_{ldpc}-1}$.

Fase di output dei bit di parità In questa fase si inviano in uscita del vettore dei bit di parità elaborato ed inizializzando la RAM per una nuova acquisizione. I multiplexer selezionano gli ingressi B.

6. La FSM procede allo scorrimento sequenziale della RAM: in uscita della memoria si riporta quindi la successione dei valori letti e intan-

to la si inizializza cancellandola. In uscita dell'encoder si trasmette l'operazione di xor tra il bit letto e quello precedente.

$DATA\_OUT = RAM[i] \oplus RAM[i-1], i = 1, ..., N_{ldpc} - K_{ldpc} - 1$

L'unico inconveniente si ha per la prima locazione che deve essere semplicemente letta. Se si presuppone che l'elemento di memoria sia inizializzato nullo all'avvio della procedura, il processo viene eseguito correttamente.

7. Il contenuto della RAM è completamente nullo dato che si son già restituiti tutti i bit di parità ed è già inizializza come supposto al punto 1.

Questa rielaborazione dell'algoritmo deve essere considerata dal lettore come una linea guida del processo che verrà implementato in seguito a livello concettuale.

Capitolo 4

Implementazione VHDL

Partendo dallo schema di base ottenuto dall'analisi dell'algoritmo che rappresenta per lo più l'idea generale di come realizzarlo, è necessario passare ad una descrizione e suddivisione dettagliata dei moduli che andranno a formare l'encoder.

La soluzione che si è deciso di realizzare è stata valutata in modo tale che ogni modulo descrivesse una funzione o un oggetto fondamentale e quindi seppur potendo realizzare meno entity, si è andati verso una comprensione maggiore del codice. Di seguito riportiamo quindi tutte le entity realizzate:

parameters Dato in ingresso il code rate, restituisce in uscita tutti i parametri inerenti alla codifica scelta;

parameters_rom Memoria ROM che contiene i parametri essenziali dai quali poi è possibile ricavarsi tutti gli altri;

table Rappresenta la tabella in uso dall'encoder per la codifica selezionata;

table_A1, ..., table_B1, ... Rappresentano le tabelle presenti nel documento [10][Allegato A] in un formato tale da occupare il minor spazio possibile;

datapath Elemento aritmetico che realizza la funzione 3.2;

ram Memoria RAM che rappresenta il vettore dei bit di parità da realizzarsi;

ldpc_encoder Implementa l'encoder LDPC vero e proprio utilizzando tutte le entity finora descritte;

ldpc_encoder_module Espansione dell'encoder alle periferiche adiacenti, cioè alle FIFO di ingresso e uscita, e viene utilizzato solo per vedere il corretto funzionamento anche con esse.

Codifica dei *code rate*. Essendo questo dispositivo configurabile per 13 *code rate* differenti e anche con variazioni della lunghezza della parola codifica, impone di scegliere innanzitutto una codifica utile che raccolga questo tipo di informazione senza dover poi aggiungere altri elementi di memoria per la configurazione. Avendo perciò 13 possibilità, il segnale di selezione che verrà indicato con `CODE_RATE` dovrà essere di 4 bit, e notando che sono 7 le codifiche con $N_{ldpc} = 16200$ e 6 con $N_{ldpc} = 64800$, è possibile pensare che il bit discriminante sia il MSB. Per tale motivo si sono scelte le corrispondenze riportate nella Tabella 4.1. Come si può notare i valori 0000, 1000 e 1111 non sono utilizzati e perciò in queste configurazioni il dispositivo si metterà in stato di attesa di un valore corretto.

Tabella 4.1: Codifica utilizzata per i code rate dell'encoder LDPC.

$N_{ldpc} = 16200$		$N_{ldpc} = 64800$	
Code Rate	`CODE_RATE`	**Code Rate**	`CODE_RATE`
—	0000	—	1000
1/4	0001	1/2	1001
1/2	0010	3/5	1010
3/5	0011	2/3	1011
2/3	0100	3/4	1100
3/4	0101	4/5	1101
4/5	0110	5/6	1110
5/6	0111	—	1111

4.1 Parameters

Questa entity nasce dalla necessità di avere a disposizione tutti i parametri per una determinata codifica salvandone, però, il meno possibile in modo da minimizzare le risorse utilizzate. Per far ciò si devono determinare i valori che concorreranno nelle operazioni di controllo e confronto della FSM e anche nell'unità aritmetica:

N_{ldpc} Lunghezza dell'intero FECFRAME

K_{ldpc} Rappresenta la dimensione del blocco d'informazione da acquisire per generare la codifica.

$N_{ldpc} - K_{ldpc}$ Rappresenta la dimensione del vettore dei bit di parità;

Q_{ldpc} Costante necessaria per il calcolo degli indici, si veda formula 3.2;

$359 \cdot Q_{ldpc}$ Consiste nel valore massimo assunto da (m mod $360 \cdot Q_{ldpc}$) nell'unità aritmetica;

Il valore N_{ldpc} può assumere solo due valori, quindi come detto nella sezione precedente, tale informazione è ricavabile dal MSB della codifica utilizzata per cui non serve implementare nessun elemento di memoria per questo valore, se non un multiplexer che selezioni quale valore considerare. Prima di procedere con gli altri parametri elencati, bisogna osservare se vi sia una qualche relazione tra di essi, altrimenti si sarebbe costretti a salvarne 3 per codifica, poichè K_{ldpc} e $N_{ldpc} - K_{ldpc}$ sono ricavabili uno dall'altro. Andando a determinare analiticamente tali valori, si può osservare un'importante relazione:

$$359 \cdot Q_{ldpc} = N_{ldpc} - K_{ldpc} - Q_{ldpc}$$

che quindi permette di decidere quali siano i valori da salvare per poi determinare gli altri. Nella tabella 4.2 riportiamo ora il contenuto delle due ROM che si vogliono utilizzare.

Tabella 4.2: Visualizzazione del contenuto delle ROM nella entity *parameters*.

ADDRESS	ROM_NK	ROM_Q
0000	0	0
0001	12960	36
0010	9000	25
0011	6480	18
0100	5400	15
0101	4320	12
0110	3600	10
0111	2880	8
1000	0	0
1001	32400	90
1010	25920	72
1011	21600	60
1100	16200	45
1101	12960	36
1110	10800	30
1111	0	0

Implementazione di una ROM. Vi sono due possibilità per implementare una ROM: o utilizzando i moduli IP messi a disposizione dall'ambiente integrato, oppure scrivendo un modulo VHDL che quando venga sintetizzato sia riconosciuto

come tale. I moduli IP sono comodi da utilizzare perché permettono di scegliere varie strategie di realizzazione, però hanno lo svantaggio di non essere portabili, generano, cioè, un codice associato al dispositivo scelto e quindi se si volesse cambiarlo, bisognerebbe implementare nuovamente tutti i moduli realizzati. Per questo motivo si è scelto di andare a sviluppare mediante codice VHDL che risulta comunque relativamente semplice da utilizzare.

Nell'ambiente integrato ISE, si trova una sezione chiamata *Language Templates* nel quale sono raccolti vari esempi e costrutti delle funzioni, strutture e dichiarazione maggiormente utilizzate. Infatti se si cerca il codice per realizzare una ROM, lo si può trovare in *Synthesis Constructs -> Coding Examples -> ROM*. Nel frammento di codice 4.1 si riporta il template utilizzato nel quale con a[i], i = 0...15, si indica il valore associato all'indirizzo i-esimo della ROM e con rom_width si indica il numero di bit necessari per descrivere il massimo valore assunto da a[i]. Per osservare il relativo codice si consulti il file *parameters_rom.vhd*.

Listing 4.1: Template VHDL delle ROM utilizzate.

```
1 -- ROM Type Declaration
2 type <rom_type> is array (0 to (2**<ram_addr_bits>-1)) of integer
      ;
3 constant <ROM_NAME> : <rom_type>:=(a[0], a[1], a[2], ..., a[(2**<
      ram_addr_bits>-1)]);
4
5 signal <rom_data> : std_logic_vector ((<rom_width>-1) downto 0);
6
7 -- Process Declaration
8 <rom_data> <= conv_std_logic_vector(<ROM_NAME>(conv_integer(ADDR)
      ), <rom_width>);
9
10 process (<clock>)
11 begin
12   if (<clock>'event and <clock> = '1') then
13     DATA <= <rom_data>;
14   end if;
15 end process;
```

La dimensione della ROM dipende dalla lunghezza dell'indirizzo e non dal numero di elementi contenuti in essa. Seppur nulla vieti di impostare un array di dimensione diversa da 2^n, in fase di prova il simulatore non riesce a compilare il circuito ed eseguirlo correttamente perché non è possibile sintetizzare una memoria nella quale alcune locazioni non siano indicizzate perché l'uscita sarebbe indeterminata. Per questo motivo tutte le locazioni non utilizzate verranno impostate nulle.

Avendo a disposizione i parametri essenziali, è possibile ora andare a delineare l'intero modulo *parameters*. Oltre che ai 4 parametri sopra citati a cui saranno

associate delle porte, avrà ulteriori ingressi e uscite qui elencate:

CODE_RATE Ingresso di selezione della codifica secondo la Tabella 4.1.

ENABLE Ingresso di abilitazione del modulo: nel caso sia disasserito i parametri in uscita non vengono cambiati. Tale ingresso deve rimanere asserito finché il segnale `VALID_DATA` non risulta alto altrimenti i parametri coerenti alla codifica selezionati non si propagheranno fino all'uscita.

VALID_DATA Flag di segnalazione che i valori in uscita sono corretti.

CODE_TABLE Versione campionata di `CODE_RATE` da utilizzarsi poi nella selezione della tabella associata. Lo si utilizza per evitare che fluttuazioni impreviste del segnale si selezione vadano a corrompere tutta la codifica.

Senza troppo dilungarsi nella spiegazione del codice riportiamo la Figura 4.1 che descrive la sintesi associata al file *parameters.vhd* in modo da rendere immediato il funzionamento.

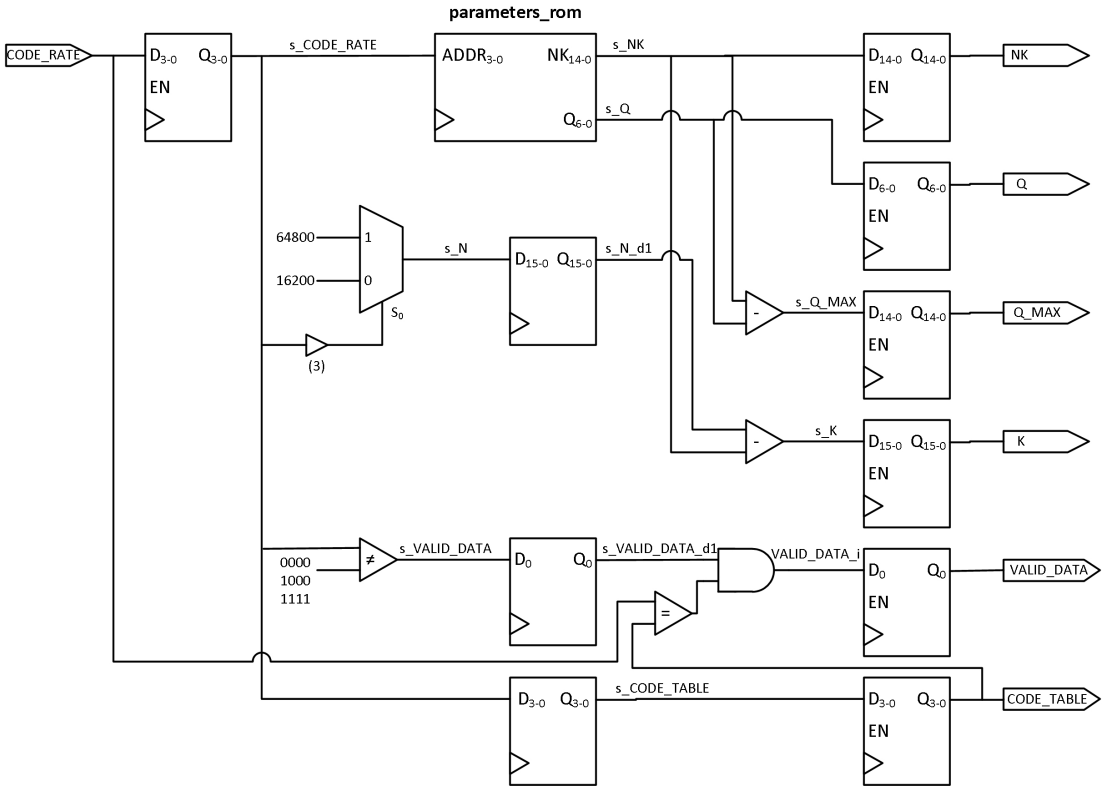


Figura 4.1: Schema descrittivo del modulo *parameters*. Il segnale di clock e di enable non sono stati collegati per una maggior comprensione.

I registri di ingresso e di uscita sono abilitati dal segnale ENABLE, in tal modo eventuali variazioni indesiderate del code rate non si possono propagare nel caso che questo sia disasserito. Questa implementazione evita di gestire casi di errore per un cambio di parametri durante l'esecuzione.

Il segnale VALID_DATA effettua essenzialmente due controlli: il primo è che la codifica in ingresso sia valida, cioè che corrisponda a un code rate esistente, mentre il secondo controlla che il segnale CODE_RATE sia uguale al segnale CODE_TABLE. Dato che quest'ultimo ha una latenza uguale a tutti gli altri parametri in uscita, questo garantisce che se vi è coerenza tra i due, allora in uscita si son propagati i parametri corretti a quella codifica. In realtà questo sistema non è a prova d'errore perché la variazione del code rate per un periodo di clock dà in uscita dei parametri errati per il modulo validi. Ciò è dovuto al registro d'uscita di VALID_DATA che ritarda il risultato del secondo controllo sopra citato. Per ovviare a questo problema è necessario campionare i parametri appena questo risulti alto, operazione che verrà svolta dalla FSM, però è lasciato al buon senso dell'utilizzatore finale mantenere stabili i segnali di selezione.

4.2 Table

Si pone ora il problema di tradurre in formato digitale le tabelle necessarie per effettuare la codifica, dato che contengono gli indici dei bit di parità da processare. Per far ciò sarà necessario sviluppare una o più memorie che contengano le diverse tabelle e di individuare anche un possibile protocollo di gestione dato che una tabella è un formato bidimensionale di dati e non unidimensionale, come per esempio un vettore.

L'unita di controllo, come visto in 3.2, processa tutti gli elementi di una riga per gruppi di 360 bit consecutivi per poi passare a quella successiva. È possibile, perciò, pensare che nell'ottica di implementare un iteratore su un vettore, struttura facilmente realizzabile mediante una ROM, l'unica informazione da aggiungere è un flag che indichi quando si è arrivati all'ultima locazione della riga corrente. In tal modo se si vuole passare alla riga sottostante basta passare all'elemento successivo, altrimenti si ritorna a puntare al primo elemento della riga precedentemente salvato. Si sono vagliate varie soluzioni tra cui quella di memorizzare il numero di elementi per riga, ma questo avrebbe comportato una maggior complessità del controllo e l'onere di salvare anche tali valori, oppure di inserire tra due righe adiacenti un valore mai assunto, per esempio 111...1, però il controllo sarebbe avvenuto successivamente all'ultimo valore.

Quella implementata, invece, è quella di utilizzare come flag il MSB della codifica in binario dei valori. Questa scelta potrebbe comportare un aumento delle risorse utilizzate perché nelle altre soluzioni i bit aggiunti erano sicuramente minori, però si ha una notevole semplificazione a livello di gestione e soprattutto di immediatezza.

Si pone ora il problema di determinare quale sia la lunghezza in bit dei valori delle tabelle. Per questo motivo si riporta nella Tabella 4.3, una sintesi del valore massimo e del numero di elementi trovati per ciascuna tabella, compresi anche il numero di bit minimi necessari per la loro rappresentazione. Infatti $\lceil \log_2 x_{MAX} \rceil$ andrà a determinare la larghezza della ROM mentre $\lceil \log_2 N_x \rceil$ la sua profondità.

Tabella 4.3: Statistiche delle tabelle implementate nel modulo *Table*.

Tabella	x_{MAX}	$\lceil \log_2 x_{MAX} \rceil$	N_x	$\lceil \log_2 N_x \rceil$
A1	32370	15	450	9
A2	25904	15	648	10
A3	21563	15	480	9
A4	16145	14	540	10
A5	12925	14	576	10
A6	10752	14	600	10
B1	12726	14	63	6
B2	8935	14	85	7
B3	6340	13	126	7
B4	5287	13	120	7
B5	4207	13	108	7
B6	3545	12	105	7
B7	2831	12	121	7

Si può procedere in due modi differenti: o implementare una ROM per ogni singola tabella oppure compattarle tutte in un'unica memoria. La prima comporta il vantaggio che per cambiare una singola tabella basta cambiare semplicemente il contenuto della ROM associata, mentre nel secondo bisognerebbe modificare l'intera memoria aggiornando anche eventualmente gli indici che mi definiscono gli estremi di ogni singola tabella. Per ragioni di semplicità e anche di maggior facilità di utilizzo, si procede implementando la prima soluzione, dimensionando le memorie con la larghezza $\lceil \log_2 x_{MAX} \rceil + 1$ e profondità $2^{\lceil \log_2 N_x \rceil}$. Il contenuto della ROM corrisponderà al vettore ottenuto dalla concatenazione delle righe con la modifica che l'ultimo elemento di ogni riga sia incrementato di $2^{\lceil \log_2 x_{MAX} \rceil}$.

Di seguito si mostra un estratto della Tabella A1, nel documento [10][Allegato A], e il contenuto della ROM associata. In entrambi si evidenzia l'ultimo elemento della prima riga.

$$\begin{bmatrix} 54 & 9318 & 14392 & 27561 & 26909 & 10219 & 2534 & \mathbf{8597} \\ 55 & 7263 & 4635 & 2530 & 28130 & 3033 & 23830 & 3651 \\ \vdots & & & & & & & \vdots \end{bmatrix}$$

$$\begin{bmatrix} 54 & 9318 & 14392 & 27561 & 26909 & 10219 & 2534 & \mathbf{41365} & 55 & \cdots \end{bmatrix}$$

In questo modo vengono implementate tutte le ROM contenute nei file indicati con *table_xx.vhd* per esser coerenti con la denominazione associata nel documento [10]. Per realizzare il contenuto della memoria, in maniera rapida, si utilizza il file Matlab *CreateHFile.m* che verrà successivamente descritto nella sezione 5.1.

Il modulo *table* è l'unione di tutte le tabelle realizzate mediante un multiplexer, la cui selezione avviene mediante il code rate selezionato, quindi presenta le seguenti porte:

CODE_TABLE Ingresso di selezione della tabella: corrisponde alla medesima codifica del code rate.

ADDR Indirizzo della memoria a cui accedere.

P_ADDR Indice del vettore dei bit di parità, cioè il contenuto della tabella puntato da `ADDR`.

P_END_ROW Flag che indica se il valore corrente corrisponde all'ultimo della riga.

Le varie tabelle per come sono state implementate hanno ingressi e uscite di dimensioni differenti. Nell'unirle si deve aver l'accortezza di omologarle tutte con la medesima dimensione, cioè quella massima. Nella Figura 4.2 si riassume la struttura implementata nel file *table.vhd*.

4.3 Datapath

In questa sezione si va a sviluppare l'unita aritmetica dell'encoder LDPC, cioè quel circuito che permette di realizzare nel miglior modo possibile la funzione 3.2.

Come detto precedentemente, l'operazione di modulo e di moltiplicazione farebbe pensare di dover implementare moltiplicatori e divisori a livello hardware. Ciò comporterebbe o a tempi di propagazione elevati, per esempio dovendo realizzare un moltiplicatore a matrice, oppure a tempi di latenza notevoli immaginando che si debba eseguire una moltiplicazione e due divisioni. Per tali motivi si è analizzato quali sono effettivamente le dinamiche di questa funzione in relazione alle tabelle date per osservare se fossero necessari inserire questi due elementi.

Per riuscire ad estrarre il circuito finale si deve scomporre la funzione nelle varie operazioni elementari che la compongono ed analizzarle singolarmente:

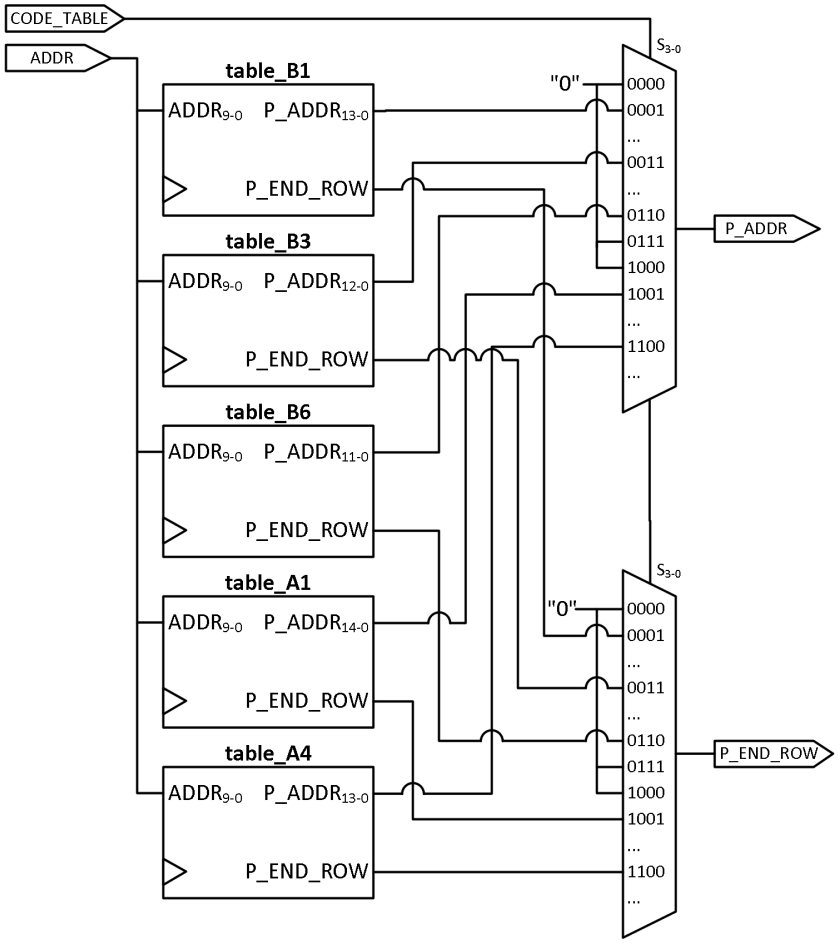


Figura 4.2: Schema descrittivo del modulo *table*.

m **mod** 360 Si ricordi che m è l'indice del bit m-esimo che viene elaborato, quindi può assumere valori compresi tra 0 e $K_{ldpc} - 1$. Essendo che i bit in ingresso arrivano in successione, si ha che $m(nT+1) = m(nT)+1$, quindi facilmente realizzabile con un contatore che ad ogni nuovo bit incrementa il suo valore. L'operazione di modulo non fa altro che limitare il conteggio al valore 359. È sufficiente implementare un contatore in cui lo stato successivo ad $m = 359$ sia $m = 0$.

m **mod** $360 \cdot Q_{ldpc}$ Per rendere immediata la risoluzione è sufficiente riscrivere l'operazione nel seguente modo:

$$(m \bmod 360) \cdot Q_{ldpc} = (m \cdot Q_{ldpc}) \bmod (360 \cdot Q_{ldpc})$$

Risulta, quindi, possibile descrivere questa funzione come una successione, tale che $c(nT+1) = c(nT) + Q_{ldpc}$, realizzabile mediante un contatore il cui incremento non è più unitario, ma di Q_{ldpc}. Per il medesimo ragionamento di prima, il contatore dovrà azzerarsi per lo stato successivo a $c = 359Q_{ldpc}$. Trova, quindi, spiegazione l'aver salvato tale valore nel modulo *parameters*.

$x + m$ **mod** $360 \cdot Q_{ldpc}$ In questo caso è immediato l'utilizzo di un sommatore e l'unico parametro da determinare è a quanti bit questo dovrà essere. Nella Tabella 4.4 si va a calcolare il massimo valore assunto dalla funzione, cioè $x_{MAX} + 359Q_{ldpc}$ per ogni codifica. Dovendo scegliere il valore massimo, è necessario implementare un sommatore a 16 bit.

$\{x + m$ **mod** $360 \cdot Q_{ldpc}\}$ **mod** $(N_{ldpc} - K_{ldpc})$ Per procedere alla semplificazione di questo passaggio è utile notare il rapporto contenuto nell'ultima colonna della Tabella 4.4. Come è possibile osservare, il valore è sempre minore di 2, quindi il risultato o è uguale al valore in uscita dal sommatore oppure alla differenza tra questo e $N_{ldpc} - K_{ldpc}$, realizzabile mediante un altro sommatore. Definiamo quindi la funzione $t_2(x)$, che rappresenterà un segnale nel modulo *datapath*:

$t_2(x) = x + m \bmod 360 \cdot Q_{ldpc} - (N_{ldpc} - K_{ldpc})$

e dalle considerazioni fatte è perciò possibile definire la 3.2 come:

$$\begin{cases} x + m \bmod 360 \cdot Q_{ldpc}, & \text{se } t_2(x) < 0 \\ t_2(x), & \text{se } t_2(x) \geq 0 \end{cases}$$

Tabella 4.4: Determinazione dei massimi valori assunti nella formula 3.2.

Tabella	Q_{ldpc}	x_{MAX}	$x_{MAX} + 359Q_{ldpc}$	$\frac{x_{MAX}+359Q_{ldpc}}{N_{ldpc}-K_{ldpc}}$
A1	90	32370	64680	1,996
A2	72	25904	51752	1,997
A3	60	21563	43103	1,996
A4	45	16145	32300	1,994
A5	36	12925	25849	1,995
A6	30	10752	21522	1,993
B1	36	12726	25650	1,979
B2	25	8935	17910	1,990
B3	18	6340	12802	1,976
B4	15	5287	10672	1,976
B5	12	4207	8515	1,971
B6	10	3545	7135	1,982
B7	8	2831	5703	1,980

Da questo risultato si ha che per implementare il modulo sono necessari tre sommatori, uno per il contatore di incremento Q_{ldpc}, uno per eseguire la somma con il segnale x e uno con per il calcolo di $t_2(x)$. Non sono da sottovalutare i comparatori che vengono implementati anch'essi mediante sommatori.

La topologia del circuito potrebbe risultare complessa e lenta dato che normalmente nella sintesi si implementano sommatori ripple-carry all'interno degli FPGA e avendone più in cascata il tempo di propagazione del riporto potrebbe far ridurre le prestazioni dell'intero sistema se si considera il periodo di clock.

Per ovviare a questo problema si può pensare di inserire i vari elementi in una struttura pipeline, perciò il periodo di clock sarà condizionato solamente dal tempo di propagazione di un solo sommatore o comparatore invece che dall'intera catena. Nella figura 4.3 viene illustrato il modello descritto nel file *datapath.vhd*, di cui a seguito vengono indicate le varie porte e segnali che lo interessano.

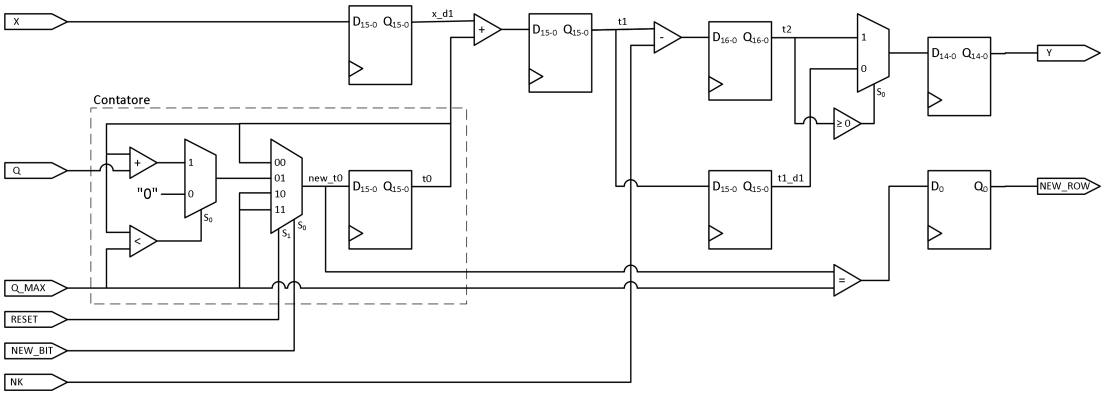


Figura 4.3: Schema descrittivo del modulo *datapath*.

RESET Se asserito, imposta il contatore t_0 a $359Q_{ldpc}$ in modo tale che all'arrivo del primo bit il contatore si azzeri;

X Operando che consiste negli indirizzi provenienti dal modulo *table*;

NEW_BIT Se assetito, incrementa il contatore di Q_{ldpc} aggiornandolo;

Q Costante di incremento del contatore (Q_{ldpc});

Q_MAX Costante che viene utilizzata come soglia per il contatore ($359Q_{ldpc}$);

NK Costante utilizzata per il calcolo della funzione $t_2(x)$ ($N_{ldpc} - K_{ldpc}$);

NEW_ROW Porta in uscita che viene asserita quando si sta processando l'ultimo bit della riga corrente, in tal modo la FSM passa alla riga successiva di *table*;

Y Risultato che consiste negli indici dei bit di parità da eleborare;

t0 Segnale interno: $t_0 = m \bmod 360 \cdot Q_{ldpc}$

t1 Segnale interno: $t_1 = x + m \bmod 360 \cdot Q_{ldpc}$

t2 Segnale interno: $t_2(x) = x + m \bmod 360 \cdot Q_{ldpc} - (N_{ldpc} - K_{ldpc})$. Il segnale `t2` è di tipo *signed* dato che può assumere sia valori positivi che negativi, perciò dovrà esser rappresentato in complemento a due. Per questo motivo è importante ricordare che la dimensione di tale segnale dovrà essere maggiorata di un bit rispetto a quella utilizzata per i segnali `t1` e `NK`.

Nel capitolo 5.1 è possibile osservare il suo comportamento in una simulazione di tipo behavioral per comprenderne maggiormente il funzionamento.

4.4 RAM

Dall'algoritmo visto in 3.2 si comprende che è indispensabile una memoria per contenere i risultati parziali ottenuti dall'elaborazione. Infatti se si esplicitano le operazioni eseguite su ogni singolo bit di parità si ottiene che in realtà corrispondono ad una funzione di più valori in ingresso. Per esempio, si riporta un estratto di tali espressioni nel caso di una codifica con code rate $1/2$ di un *normal FECFRAME* per i primi tre valori.

$p_0 = i_{9420} \oplus i_{9821} \oplus i_{11093} \oplus i_{12960} \oplus i_{26819}$

$p_1 = i_{3899} \oplus i_{3910} \oplus i_{7274} \oplus i_{13320} \oplus i_{14966} \oplus p_0$

$p_2 = i_{1769} \oplus i_{6106} \oplus i_{9762} \oplus i_{13680} \oplus i_{18582} \oplus p_1$

Dato che il blocco d'informazione viene fornito tramite una FIFO, quindi un ingresso di tipo seriale, implica che ogni bit di parità, p_i, verrà modificato se il bit i_m in ingresso è presente nella sua espressione e questo avverrà in istanti temporali diversi. Si dovrà allora memorizzare il valore ottenuto da ogni precedente operazione e aggiornare il vettore con i nuovi valori ottenuti.

Potrebbe esser lecito pensare che esplicitando tutte funzioni associate per ogni bit di parità sia possibile realizzare l'intero encoder mediante una rete combinatoria semplicemente convertendo l'ingresso da seriale a parallelo. Anche se risulta la scelta con prestazioni in assoluto migliori, in realtà si deve far i conti con l'utilizzo di risorse pressoché spropositato.

Si deve, quindi, implementare una RAM che abbia lunghezza di parola unitaria, cioè di un singolo bit, e che sia profonda tanto quanto il vettore di bit di parità da generare. Essendo che l'encoder è progettato per funzionare in più modalità, si dovrà dimensionare la memoria in modo tale che questa riesca a contenere in ogni caso i bit di parità. Consultando, perciò, le tabelle 3.1 e 3.2 si ha che il valor massimo è di 32400 bit corrispondente ad un'indirizzamento minimo della memoria a 15 bit.

Definiti, quindi, larghezza e profondità si deve ora scegliere il tipo di RAM che si vuole implementare. La procedura più problematica dell'algoritmo utilizzato è

che una locazione di memoria deve essere contemporaneamente letta che scritta. Difatti prima se ne deve leggere il contenuto, elaborarlo con il bit in ingresso e poi scrivendolo di nuovo nella medesima locazione. Per questo motivo si è scelto di utilizzare una DUAL PORT RAM, in tal modo le due porte possono essere specializzate per le due operazioni in modo da lavorare indipendentemente una dall'altra. Unica importante attenzione che si deve fare è che non son consentite operazioni di lettura e scrittura nella medesima cella contemporaneamente. Per ovviare a questo problema si rimanda alla struttura di controllo dato che sarà lei a gestire gli accessi.

Implementazione di una RAM. Come fatto per le ROM, è possibile trovare i template consultando la sezione *Synthesis Constructs -> Coding Examples -> RAM* all'interno dell'ambiente integrato. Andando a modificare secondo esigenza la struttura è possibile ricavare la RAM desiderata di cui si riporta nel listato 4.2 solamente il processo legato ad una porta dato che per l'altra è simmetrico. Come convenzione dei nomi si è utilizzata quella riportata nei datasheet della *Virtex4* nella sezione *BLOCK RAM*.

Listing 4.2: Descrizione VHDL della DUAL PORT RAM simmetrica a un clock utilizzata nel progetto.

```
1  architecture Behavioral of ram is
2
3      type ram_type is array (0 to 32399) of std_logic;
4      shared variable RAM : ram_type := (others => '0');
5      signal s_ADDRA, s_ADDRB : integer range 0 to 32399;
6
7  begin
8    -- Limitazione dell'indirizzamento della memoria
9    s_ADDRA <= conv_integer(A);
10   s_ADDRB <= conv_integer(B);
11
12 process (CLK)
13   begin
14   if (CLK'event and CLK = '1') then
15       if (ENA = '1') then
16          if (WEA = '1') then
17             RAM(s_ADDRA) := DIA;
18          end if;
19          DOA <= RAM(s_ADDRA);
20       end if;
21     end if;
22     end process;
23     ...
```

Dal codice si può osservare che la RAM viene inizializzata a zero alla sua implementazione. Facendo così è possibile evitare di dover cancellare la RAM nel primo passaggio dell'algoritmo semplificandone quindi il relativo codice.

4.5 LDPC Encoder

In questo modulo si vanno ad organizzare i componenti finora descritti in modo da realizzare l'encoder LDPC secondo lo schema di Figura 3.6.

Prima di passare all'implementazione del modulo è necessario confrontarsi con i segnali che lo caratterizzeranno e quindi valutare anche gli elementi con cui andrà a interagire. Come si può evincere dalla figura sopra citata, i frame di ingresso e di uscita vengono scambiati con delle memorie FIFO e quindi bisognerà comprendere anche i segnali per il loro controllo. Si dovrà anche creare un'interfaccia da garantire un minimo di controllo da parte dell'utilizzatore. Da queste considerazioni si ricavano l'insieme di porte della entity di seguito elencate:

`CLK` Segnale di sincronismo dell'encoder: viene utilizzato come segnale di clock per tutti i componenti del modulo;

`CODE_RATE` Segnale di selezione della *code rate* secondo le specifiche di Tabella 4.1;

`ENABLE` Segnale di abilitazione: se l'encoder è inizializzato, asserendo tale ingresso viene avviata la codifica di un nuovo frame in ingresso;

`DATA_IN` Ingresso seriale che viene collegato alla FIFO contenente il frame da codificare;

`EMPTY_FIFO` Segnale di FIFO vuota in ingresso;

`FULL_FIFO` Segnale di FIFO piena in uscita;

`READY` Flag di encoder inizializzato: nel caso sia asserito indica che l'encoder è pronto a ricevere il blocco d'informazione;

`DATA_OUT` Uscita seriale che invia la parola codificata alla FIFO;

`RD_EN` Segnale di lettura della FIFO in ingresso: se asserito, con latenza di un ciclo di clock, si presenta in uscita della FIFO un nuovo bit;

`WR_EN` Segnale di scrittura della FIFO in uscita: se asserito, la FIFO campiona il bit al suo ingresso e lo salva in memoria.

Passando alla sua implementazione, come anticipato nella sezione 3.3 è possibile suddividere l'algoritmo in due iterazioni separate: nella prima il vettore di parità viene elaborato dall'informazione entrante nel modulo, mentre nel secondo è associata solo al vettore stesso. Vista l'opportunità di specializzare anche la parte hardware, si è ritenuto opportuno creare una FSM principale, che a seconda

dello stato corrente gestisca i collegamenti tra le varie entity interne modificando, quindi, i vari percorsi dei segnali e potendo anche riutilizzare strutture già istanziate. Per la gestione dei segnali di controllo di tali strutture si è valutata interessante l'idea di farli controllare da una macchina a stati locale, cioè definita all'interno degli stati di quella principale, in modo che la struttura su cui si elaborano i dati sia la medesima e vi sia una FSM che la gestisse autonomamente.

Si parte dal presupposto che i segnali di controllo siano sincroni, onde evitare che la logica combinatoria legata alla loro generazione possa restituirli con ritardi diversi. Tutto ciò a discapito di un incremento della latenza del circuito che sarà da tenere conto nella realizzazione delle strutture e dei controlli.

Visto che in seguito verranno citati i segnali d'uscita della FSM è opportuno citare la convenzione che si è utilizzata:

[new_*] Segnale in ingresso ad un registro che verrà riportato sul segnale `s_*` nel periodo successivo

[s_*] Segnale di tipo generico

[en_*] Segnale di abilitazione

[*_XX] Indicano che tali segnali sono collegati a porte di ingresso e o uscita di moduli

[*_i] Rappresentazione interna al modulo di una porta

[delay_reg_*] Shift register

Altra scelta progettuale è stata quella di far sì che le uscite della FSM che rappresentano segnali di controllo della RAM e dell'unità aritmetica siano normalmente disasseriti, perciò è possibile riuscire a inviare impulsi di durata arbitraria in base alla permanenza in uno stato che li asserisce. Questo è essenziale dato che le principali operazioni che dovranno svolgere saranno di campionamento o di incremento di contatori e, quindi, si deve esser certi della durata dei segnali di abilitazione, soprattutto nel caso di *clock enable* nei contatori.

Da queste premesse, si è arrivati a definire quattro stati per la FSM principale e i relativi ambiti di funzionamento:

idle_0 Abilitazione dell'encoder per la modifica del code rate precedentemente impostato e inizializzazione di eventuali registri;

idle_1 Stato di attesa di convalida dei parametri utilizzati e dell'abilitazione da parte dell'utente dell'esecuzione della codifica;

acquireBit Ricezione del blocco d'informazione da codificare e elaborazione del vettore di parità con i bit entranti. In questo momento l'encoder risulta trasparente, cioè i bit in uscita sono una versione ritardata di quelli in ingresso;

bufferedOutBit Computazione del vettore di parità e invio in uscita del suo contenuto.

Gli stati sul quale è necessario soffermarsi sono sicuramente `acquireBit` e `bufferedOutBit` perché sono quelli dove vengono implementate le due strutture computazionali di Figura 4.4 e anche si sono andate a definire le FSM locali di controllo delle medesime. La differenza sostanziale tra i due stati consiste nel come essi collegano la RAM, rappresentante il vettore di parità, con i rimanenti componenti istanziati. Si procede ora con una descrizione dettagliata di ciascuno schema in modo che il lettore possa comprenderne il funzionamento.

Descrizione della struttura implementata in `bufferedOutBit`. In questo stato si eseguono le operazione descritte nell'algoritmo di sezione 3.2 nei punti 6 e 7, quindi si svolge l'ultima elaborazione dei dati contenuti nella RAM e il successivo invio alla FIFO.

Quando si raggiunge questo stato, nella memoria vi sono contenuti i bit precedentemente calcolati con l'informazione da codificare. L'ultima operazione da eseguire è quella definita dalla formula 3.3 che suggerisce di dover salvare di nuovo nella memoria il risultato appena ottenuto perché deve essere utilizzato per calcolare il bit successivo. Questo risulta molto sconveniente perché una volta calcolati i bit e quindi possibilmente anche inviati in uscita, la memoria non risulta vuota e, quindi, è necessario cancellarla prima di acquisire un nuovo frame in ingresso.

Per ovviare a entrambi i problemi si è arrivati a sintetizzare lo schema di figura che permette, se comandato opportunamente, di aver la memoria già inizializzata dopo l'invio dell'ultimo bit di parità.

Per comprenderne il funzionamento si pensi di inviare in successione gli indirizzi da 0 a $N_{ldpc} - K_{ldpc}$ su `new_A` e contemporaneamente un impulso su `new_ENA` e `new_wr_en`. Dopo due cicli di clock all'uscita della PORTA della RAM, `DOA`, si presenta il contenuto della cella selezionata e il segnale `s_DOA_d1` assume il valore del bit di parità precedentemente elaborato dato che nel periodo precedente questo è stato abilitato. Nel ciclo di clock successivo si ha che la cella di memoria precedentemente puntata viene cancellata mediante la PORTB che ora è abilitata, sulla porta `DATA_OUT` viene riportata il valore campionato dalla porta xor e `WR_EN` viene asserito.

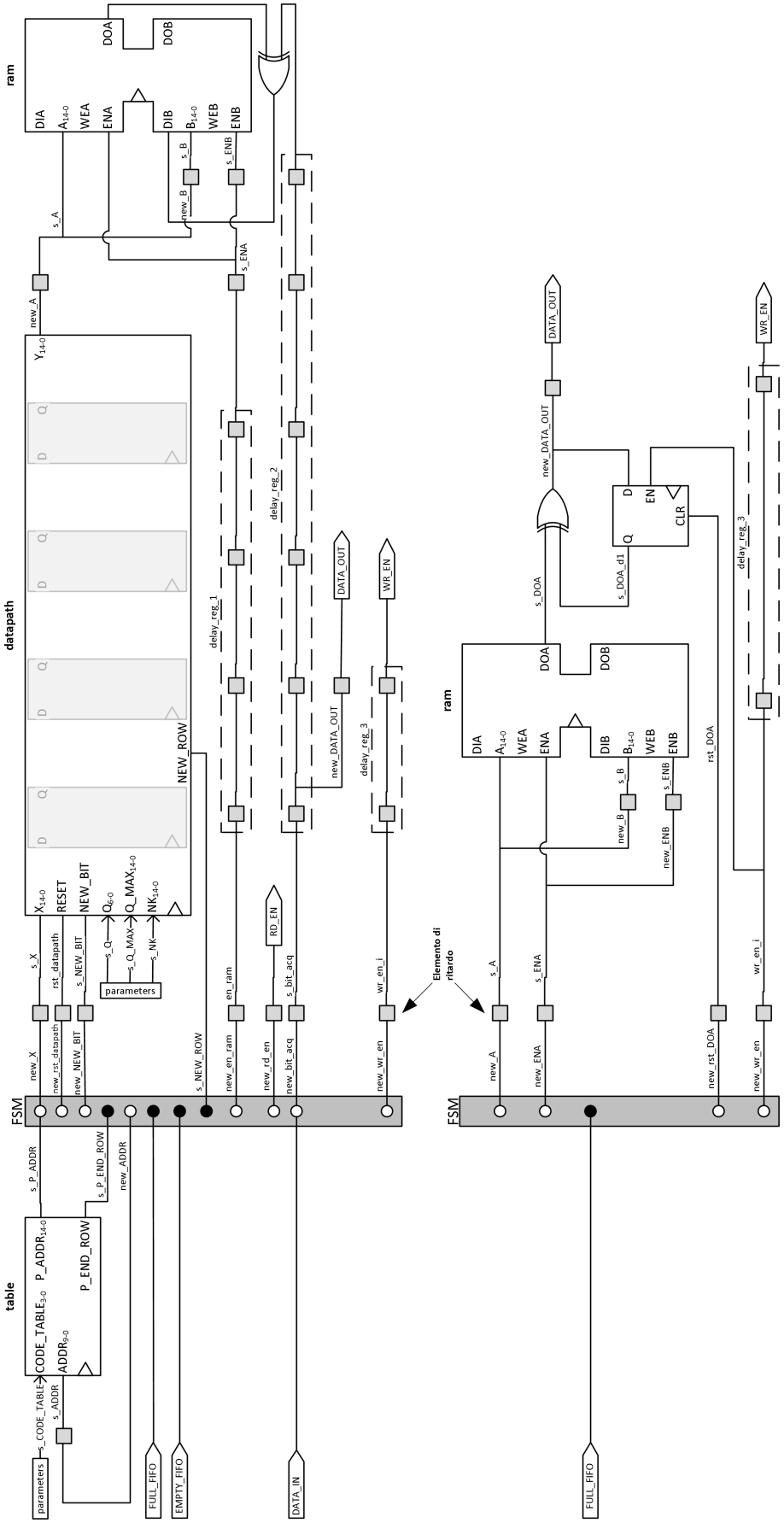


Figura 4.4: Strutture computazionali realizzate dagli stati `acquireBit` e `bufferedOutBit`.

L'unica eccezione a questa procedura deve esser fatta per il primo bit, infatti p_0 deve essere riportato in uscita senza esser elaborato. Per non sconvolgere ulteriormente la struttura si è andati ad inserire un reset sincrono nel registro e la catena di ritardo legata al segnale `new_rst_DOA`. In tal modo inviando assieme agli altri segnali un impulso anche su quest'ultimo, dopo due periodi, quando in uscita della RAM si presenta p_0, il registro viene resettato e, quindi, in uscita si avrà $p_0 \oplus 0 = p_0 \cdot \overline{0} + \overline{p_0} \cdot 0 = p_0$ riportandolo in uscita nel ciclo di clock successivo.

Le operazioni così fatte permettono di inizializzare la RAM automaticamente una volta inviati i dati, ma rimane il problema della prima inizializzazione, cioè l'accensione. In realtà è facilmente risolvibile facendo sì che all'istanziazione della RAM questa sia posta inizialmente nulla come indicato nel Listato 4.2 alla riga 4.

Descrizione della struttura implementata in `acquireBit`. Di maggior complessità è, invece, la struttura realizzata per questo stato, cui concerne l'esecuzione del primo blocco dell'algoritmo nei punti dal 2 al 5. La necessità di consultare le tabelle e il calcolo degli indirizzi implica la necessità di utilizzare sia il modulo *table* che *datapath* nella gestione della RAM.

La FSM comanda il modulo *table* mediante il segnale `new_ADDR` e imponendogli l'attraversamento desiderato secondo il valore dei segnali `s_P_END_ROW` e `s_NEW_ROW`. Se solo il primo è asserito indica che si è al termine della riga corrente, ma bisogna ancora elaborare dei bit con quella riga e quindi di riportare il puntatore al primo elemento della riga, mentre se entrambi sono asseriti, indica che si è terminata la computazione di quella riga e il puntatore dovrà essere impostato sul primo elemento della riga successiva.

Prima di passare alla trattazione delle gestione della RAM da parte dell'unità aritmetica è opportuno fare alcune osservazione sull'acquisizione dei bit. Per acquisire un bit dalla FIFO in ingresso si dovrà inviare un impulso mediante `new_rd_en`, in tal modo la memoria dopo due periodi di clock, porrà in uscita il bit da elaborare che è prelevabile dal segnale `DATA_IN`.

In questo stato l'encoder risulta trasparente, cioè riporta in uscita una versione ritardata dei bit in ingresso, quindi collegando il segnale `DATA_IN` a `new_bit_acq` e inviando un impulso su `new_wr_en`, dopo tre cicli di clock in uscita dall'encoder si presentano i segnali per caricare sulla FIFO in uscita il valore corrente. Il tempo di latenza è stato deciso per mantenere inalterata la struttura relativa al segnale `new_wr_en` già utilizzata in quella precedente.

Analizzando l'operazione dell'algoritmo da eseguire, si nota che si deve leggere e scrivere sulla stessa cella di memoria. Si ricorda che in una memoria DUAL

PORT RAM non è consentita tale operazione anche se effettuata utilizzando porte diverse se avviene contemporaneamente. Normalmente si ritiene sconsigliato anche puntare alla medesima locazione con entrambe le porte se abilitate. Per risolvere questo problema si utilizza la stessa configurazione implementata in `bufferedOutBit` anche se con scopi diversi. Una volta che in `new_A` si è caricato il valore dell'indirizzo da elaborare e inviato l'impulso di abilitazione su `new_ENA`, dopo un ciclo di clock i segnali raggiungono PORTA; nel periodo successivo si legge il valore della cella riportandolo in uscita e si propagano i segnali su PORTB, compreso il valor aggiornato di p_i, disabilitando quindi PORTA. In quello dopo la medesima cella viene scritta e la RAM viene disabilitata completamente. In questo modo le porte non sono mai abilitate contemporaneamente e questo evita le possibili collisioni.

Per controllare la RAM si devono utilizzare gli indirizzi provenienti da *table* ed elaborati da *datapath*. Per la gestione del secondo modulo, quindi, sono necessari i segnali `new_rst_datapath`, che permettano di resettarlo all'inizio dell'acquisizione di un nuovo frame, `new_NEW_BIT`, che è un clock enable del contatore interno relativo al segnale t_0 (si veda 4.3) che va assetito quando si riceve un nuovo bit, e `new_X` che consiste nel valore in ingresso da calcolare.

Per ogni indirizzo relativo ad un singolo bit, perciò, lo si deve inviare tramite `new_X` e asserire per un periodo di clock *new_en_ram*. La sincronizzazione del segnale `new_bit_acq` con questi segnali non è richiesta, dato che esso si mantiene costante per tutta la serie di indirizzi grazie al registro `s_bit_acq` che lo campiona.

I blocchi indicati con `delay_reg_1`, `delay_reg_2` e `delay_reg_3` rappresentano dei registri a scorrimento che servono per realizzare la struttura pipeline. Così facendo si evita di gestire tutte le varie tempistiche di arrivo dei segnali alla RAM dovute alle latenze dei vari componenti, dato che inviandoli assieme essi si propagano lungo la struttura e arrivano simultaneamente alla RAM correttamente.

Per una maggior comprensione delle tempistiche con le quali avvengono questi scambi si rimanda alla sezione 5.1 dove son raffigurate le simulazioni di tutto il procedimento.

Conoscendo, quindi, le strutture che si andranno ad utilizzare è possibile ora andare a descrivere tutti gli stati, sia della FSM principale che di quella locale, mediante i segnali che andranno ad asserire e a gestire per eseguire le operazioni.

`idle_0` Abilitazione del modulo *parameters* in modo che l'utente possa cambiare la codifica scelta per il precedente frame (new_en_param = 1) e inizializ-

zazione del puntatore alla tabella indirizzi *table*, alla prima locazione di memoria (new_ADDR_reg = 0);

idle_1 Attesa che il modulo *parameters* restituisca dei dati validi (s_VALID_DATA = 1) e che l'utente avvii la codifica del nuovo frame (ENABLE = 1). Nel caso che avvenga si ha l'invio al modulo *datapath* del segnale di reset (new_rst_datapath = 1) e la disabilitazione del modulo *parameters* (en_param = 0);

acquireBit Implementazione della prima struttura: collegamento dell'uscita del *datapath* all'ingresso della PORTA della RAM (new_A = s_Y) e relativa versione ritardata su PORTB (new_B = s_A), analogamente con il segnale di abilitazione della RAM allo shift register (new_A = delay_reg_1(4), new_B = s_A), collegamento della porta xor alla RAM (s_DIB = s_DOA xor delay_reg_2(5)) e della versione ritardata del bit d'ingresso all'uscita (new_DATA_OUT = delay_reg_2(0));

acquireBit_0 Attesa che la FIFO in ingresso non sia vuota e quella in uscita non sia piena (`EMPTY_FIFO` = `FULL_FIFO` = 0). Una volta verifica tale condizione, se il numero di bit letti è minore della lunghezza del frame da leggere (`bit_cnt` < `s_K`) si procede alla lettura di un nuovo bit (`new_rd_en` = 1), altrimenti si azzera il contatore dei bit e si procede alla struttura successiva;

acquireBit_1 Invio al modulo *table* del valore del puntatore al primo valore della riga precedentemente salvato (`s_ADDR` = `s_ADDR_reg`) e segnalazione al modulo *datapath* dell'arrivo di un nuovo bit (`new_NEW_BIT` = 1);

acquireBit_2 Campionamento del bit in ingresso (`new_bit_acq` = `DATA_IN`) e invio del segnale per la scrittura in uscita del medesimo (new_wr_en = 1);

acquireBit_3 Invio al *datapath* del valore proveniente dalla tabella (`new_X` = `s_P_ADDR`) e del relativo segnale per l'abilitazione della RAM (`new_en_ram` = 1). Nel caso l'elemento sia l'ultimo della sua riga (`s_P_END_ROW` = 1) allora si ritorna ad acquisire un nuovo bit. Se il bit successivo è il primo di un nuovo gruppo di 360 bit(`s_NEW_ROW` = 1) allora si salva nel registro puntatore l'indirizzo della riga successiva (`new_ADDR_reg` = `s_ADDR`+1) altrimenti rimane immutato. Nel caso, invece, questo non sia l'ultimo elemento della riga, si incrementa l'indirizzo passando al valore successivo e si passa ad uno stato di attesa;

acquireBit_4 Stato di attesa dovuto al tempo di latenza del modulo *table*;

acquireBit_5 Stato di attesa perché venga terminata l'elaborazione dei dati che si stanno propagando verso la RAM altrimenti cambiando la struttura del circuito i dati verrebbero persi. Per verificare tale condizione basta che non vi sia nessun impulso di abilitazione che stia andando alla RAM (`en_ram` = `delay_reg_1` = `s_ENA` = `s_ENB` = 0);

bufferedOutBit Implementazione della seconda struttura: collegato della PORTB come versione ritardata della PORTA di un periodo di clock (`new_B` = `s_A`, `new_ENB` = `s_ENA`) e collegamento della porta xor tra l'uscita della RAM e il registro contenente il bit di parità precedentemente elaborato (`new_DATA_OUT` = `s_DOA` xor `s_DOA_d1`);

bufferedOutBit_0 Inizializziazione del puntatore alla RAM alla prima locazione (`new_A_reg` = 0);

bufferedOutBit_1 Attesa che la FIFO in uscita non sia piena. Se si verifica e la memoria non sia stata ancora attraversata completamente (`A_reg` < `s_NK`), invio alla RAM dell'indirizzo corrente (`new_A` = `A_reg`), incremento del puntatore (`new_A_reg` = `A_reg`+1) e invio delle abilitazioni del moduli (`new_ENA` = `new_wr_en` = 1). Nel caso che sia il primo bit (`A_reg` = 0) allora, come detto precedentemente, si invia il segnale di reset al registro in uscita (`new_rst_DOA` = 1);

bufferedOutBit_2, bufferedOutBit_3, bufferedOutBit_4 Stati di attesa che i segnali inviati nel precedente stato si propaghino fino all'uscita e alla FIFO. Infatti, prima di inviare i segnali si deve esser certi che la FIFO possa riceverli e, quindi, si deve attendere che questi si propaghino in modo tale da poter controllare se questa diventi piena.

Si riporta in Figura 4.5 il diagramma di stato, così da poter vedere concretamente l'evoluzione degli stati. Per convenzione nelle transizioni si è scritto nella parte soprastante la freccia la condizione, se esiste, e invece al di sotto eventuali uscite dipendenti da una transizione.

Nel file *ldpc_ encoder.vhd* è implementato quanto detto finora in questa sezione e grazie alla divisione in vari moduli del processo finale è stato possibile descrivere abbastanza facilmente la sua struttura in codice VHDL.

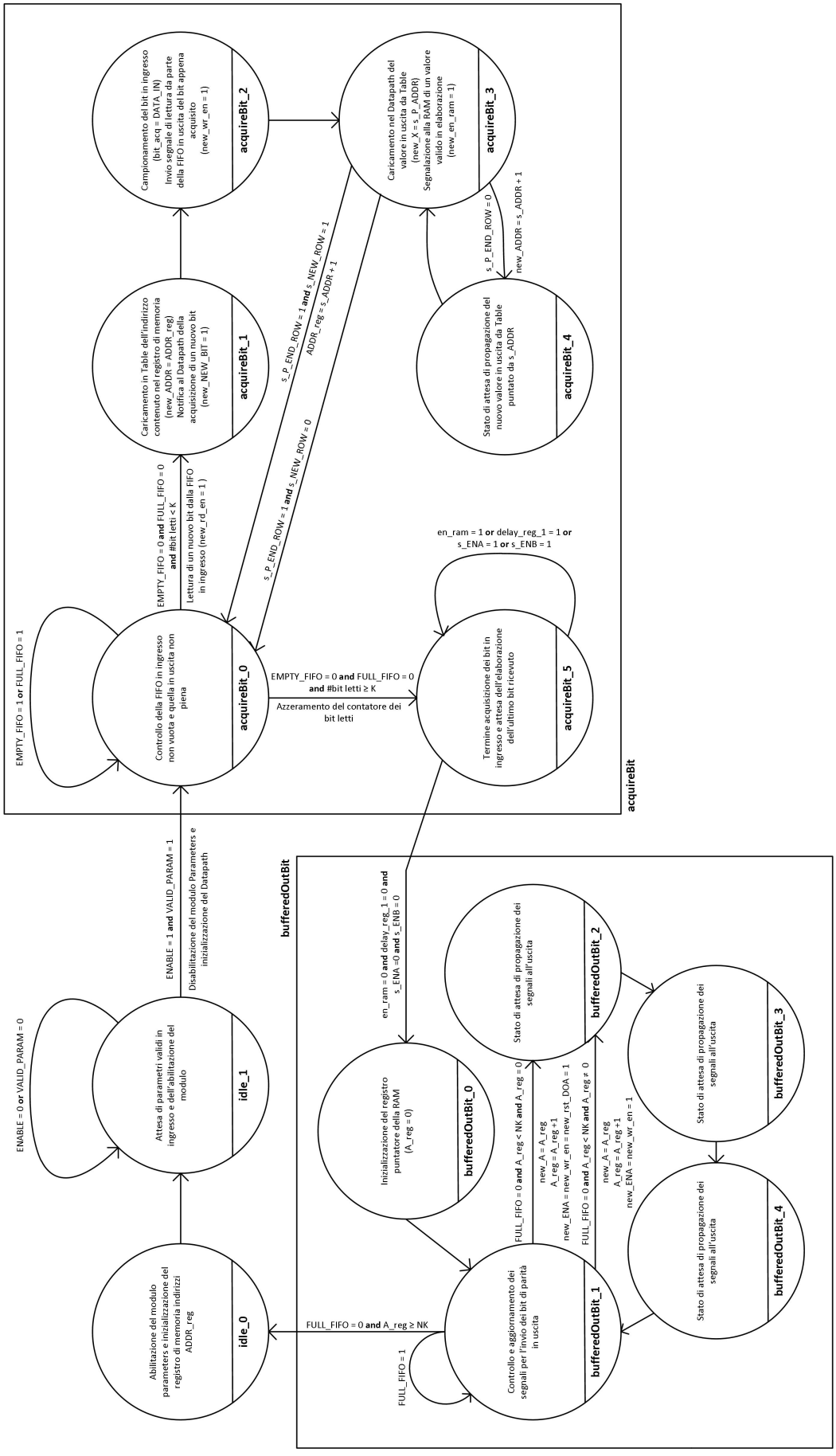


Figura 4.5: Diagramma di stato della FSM del modulo *ldpc_ encoder*.

4.6 Estensione del modello alle periferiche adiacenti

Questa entity come accennato all'inizio serve solo nelle simulazioni e per incorniciare l'encoder LDPC nel contesto nel quale lo si è pensato.

Per la realizzazione delle FIFO si utilizzano i moduli IP [1] facilmente implementabili grazie all'interfaccia *LogiCORE IP FIFO Generator v6.2* che si può trovare nell'ambiente integrato ISE andando su *Project -> IP(CORE Generator & Architecture Wizard) -> Memories & Storage Elements -> FIFOs*.

Viene scelto di realizzarle con memoria di tipo Block RAM e con clock in comune per entrambe le porte in scrittura e in lettura dato che risulta una delle forme più compatte nella loro realizzazione. Non dovendo interessarci del loro dimensionamento perché non si conosce l'entità dei flussi di dati in ingresso e in uscita, si imposta come larghezza in scrittura unitaria e come profondità pari a 65536 in modo che riesca a contenere per lo meno due frame in ingresso e uno in uscita.

In questo modo si realizzano le due entity *fifo_in* e *fifo_out* che andranno collegate all'encoder precedentemente implementato. Ogni memoria FIFO presenta i seguenti ingressi e uscite ricavabili dal template di istanziazione fornito dall'IDE:

din Vettore di bit in ingresso della memoria: nel nostro caso ha larghezza unitaria;

wr_en Abilitazione alla scrittura: se asserito, al successivo ciclo di clock avviene il campionamento e memorizzazione del valore in `din`;

rd_en Abilitazione alla lettura: se asserito, al successivo periodo di clock viene riportato in uscita `dout` il prossimo valore;

dout Vettore di bit in uscita della memoria: come l'ingresso, anch'esso ha larghezza unitaria;

empty Flag di segnalazione di memoria vuota;

full Frag di segnalazione di memoria piena.

Seguendo lo schema iniziale di Figura 3.6 è possibile ora realizzare nel file *ldpc_encoder_module* i collegamenti necessari, riassunti nella Figura 4.6.

[1] Modulo IP (Intellectual Property): componenti presviluppati che posso essere integrati all'interno del progetto e di cui non se ne conosce la realizzazione software e hardware.

La realizzazione di questo modulo permette di eseguire una simulazione più veritiera rispetto a simulare anche il comportamento delle memorie FIFO, soprattutto nella gestione dei ritardi e delle situazioni critiche. Questo modello non verrà, però, utilizzato nel calcolo delle prestazioni nel quale si andrà a considerare il caso migliore, il caso in cui nessuna delle due FIFO risulti piena o vuota.

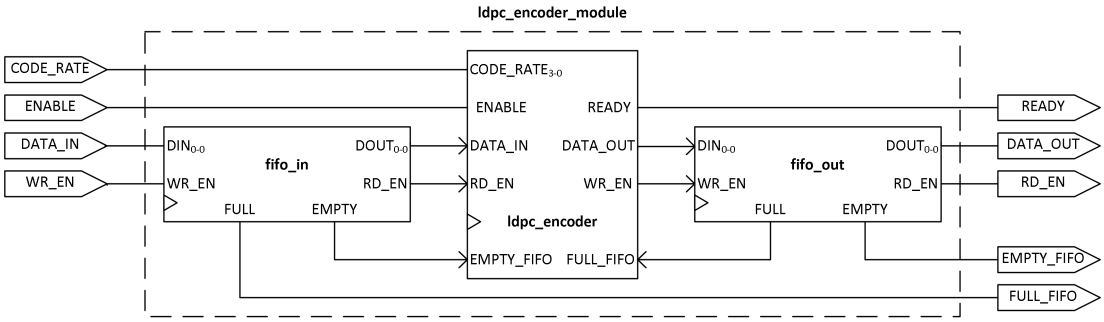


Figura 4.6: Esplosione del modulo *ldpc_ encoder_ module* nei suoi componenti.

Capitolo 5

Simulazioni e prestazioni

Una volta descritto l'intero modulo mediante linguaggio VHDL, è possibile realizzare prima la sintesi e poi anche la sua implementazione nel modello di FPGA scelto, cioè il modello XC4VSX25 nel package FF688.

Per verificare il corretto funzionamento dei vari componenti è possibile utilizzare il programma *iSim* che simula il loro comportamento grazie ai modelli generati appositamente dall'ambiente integrato ISE. Esistono vari tipi di simulazione in base al livello di sintesi o di implementazioni a cui si è arrivati:

Behavioral, detta anche comportamentale, in questo caso viene simulato solamente il codice e non vi è nessuna soluzione circuitale al modello;

Post-Translate, il modello viene elaborato miscelando tutti gli ingressi e le uscite, quindi passando per un processo di prima ottimizzazione, e viene compilato il codice in un formato di primitive definite dalla Xilinx;

Post-Map, dalle primitive sopra citate si passa ad assegnarli le risorse effettive presenti nel modello di FPGA scelto. In questo caso dalla simulazione risultano anche visibili i tempi di propagazione dei vari componenti;

Post-Place&Route, in tal caso si ha una completa mappatura del circuito all'interno dell'integrato. La sua simulazione tiene conto anche delle linee di propagazione utilizzate per collegare i vari dispositivi e cercando di rispettare le Timing Constraints [1].

Nella trattazione a seguire si tiene conto che tutte simulazioni presentate sono del tipo Behavioral, dato che il tempo per una simulazione Post-Place&Route è notevole. Se ne esegue solo una per controllare il corretto funzionamento alla frequenza di clock impostata per poi procedere con quello di tipo comportamentale.

[1]Timing Constraints: sono delle specifiche definite dall'utente nell'ambiente ISE legate al tempo di propagazione all'interno del FPGA.

Con il modello utilizzato si è scelta una frequenza pari a 160Mhz, pari a un periodo di clock di 6.25ns. In realtà non è la massima frequenza trovata perché è possibile salire anche leggermente al di sopra dei 170Mhz, ma per mantenere un certo margine di sicurezza si ritiene più che sufficiente eseguire le simulazioni ad una frequenza inferiore.

5.1 Simulazione del funzionamento mediante iSim e Matlab

Come anticipato nella sezione 4.6, nelle simulazioni verrà utilizzata la entity contenente solo l'encoder, *ldpc_ encoder*, e non il modello contenente anche le FIFO poichè si vogliono determinare le prestazioni nel caso migliore e controllare, innanzitutto, il funzionamento del modulo stesso.

Si premette che per verificare la correttezza della codifica si utilizzerà un programma scritto in Matlab e quindi è necessario salvare il bit stream in ingresso e in uscita dall'encoder quando questo verrà simulato. Per tali ragioni il Testbench che verrà utilizzato, di cui un frammento è nel Listato 5.1, creerà due file di testo, *infile.txt* e *outfile.txt*, che contengono i singoli bit inviati e ricevuti, separati da uno spazio. La scelta di questa formattazione è stata fatta per facilitare, poi, l'acquisizione da parte del programma di verifica.

Listing 5.1: Frammento di codice del Testbench utilizzato per la simulazione del modulo *ldpc_ encoder*.

```
1  stim_proc: process
2  begin
3      -- Creazione dei file di ingresso e di uscita
4    file_open(outfile, "outfile.txt", write_mode);
5    file_open(infile, "infile.txt", write_mode);
6    -- Attesa di inizializzazione del sistema
7    wait for 100 ns;
8    -- Invio del code rate 1/2
9    CODE_RATE <= "1001";
10   wait for 100 ns;
11   -- Abilitazione del modulo
12   ENABLE <= '1';
13   FULL_FIFO <= '0';
14   -- Se venie acquisito un frame l'encoder viene bloccato
         simulando che la FIFO in ingresso sia vuota
15   if(stop_acquire_data = '0') then
16     EMPTY_FIFO <= '0';
17   else
18     EMPTY_FIFO <= '1';
19   end if;
20   wait;
21 end process;
22
```

```
23 test_proc: process(CLK)
24   variable seed1: positive;
25   variable seed2: positive;
26   variable rand: real;
27   variable int_rand: integer;
28   variable f_status: FILE_OPEN_STATUS;
29   variable value1, value2: integer := 0;
30   variable data: std_logic_vector(31 downto 0);
31   variable buf1, buf2: line;
32 begin
33   if (stop_acquire_data = '0') then
34     if (CLK'event and CLK = '1') then
35       -- Simulazione della FIFO in ingresso
36       if(RD_EN = '1') then
37         -- Se in numero di bit inviati è minore di un frame
               allora genera un numero pseudo casuale da inviare all'
               ingresso dell'encoder
38         if(bit_cnt_tb1 < K) then
39           -- Contatore bit inviati
40            bit_cnt_tb1 <= bit_cnt_tb1 +1;
41           -- Generazione numero pseudo casuale
42           UNIFORM(seed1, seed2, rand);
43           UNIFORM(seed1, seed2, rand);
44           int_rand := INTEGER(TRUNC(rand*2.0));
45           data := std_logic_vector(to_unsigned(int_rand, data'
                 length));
46           -- Invio all'encoder del nuovo bit
47           DATA_IN <= data(0);
48           -- Salvataggio del bit nel buffer del file
49           value1:= conv_integer(data(0));
50           write(buf1, value1);
51           write(buf1, " ");
52         -- Altrimenti se ha già acquisito tutti i bit di un frame
               allora salva il file
53         elsif( bit_cnt_tb1 = K) then
54           writeline(infile, buf1);
55           file_close(infile);
56         end if;
57       end if;
58         -- Simulatore della FIFO in uscita
59       if (WR_EN = '1') then
60         -- Se in numero di bit di ricevuti è minore di un frame
               codificato allora acquisisce il nuovo bit
61         if(bit_cnt_tb2 < N) then
62           bit_cnt_tb2 <= bit_cnt_tb2 +1;
63           value2 := conv_integer(DATA_OUT);
64           write(buf2, value2);
65           write(buf2, " ");
66         -- Altrimenti se ha già acquisito tutti i bit di un frame
               allora salva il file
67         elsif(bit_cnt_tb2 = N) then
68           writeline(outfile, buf2);
69           file_close(outfile);
70           stop_acquire_data <= '1';
71         end if;
72       end if;
73           -- Contatore del numero di cicli di clock impiegati per
                 ottenere la codifica
```

```
74        if(bit_cnt_tb1 > 0 and bit_cnt_tb2 < N) then
75          count <= count +1;
76        end if;
77      end if;
78    end if;
79  end process;
```

Commento del Testbench. Nel primo processo, *stim_proc*, si ha l'inizializzazione dei file, e quindi anche l'eventuale cancellazione di quelli ottenuti da simulazioni precedenti, e viene inviato il segnale relativo alla codifica da utilizzare. Come si può notare si impone che il segnale `EMPTY_FIFO` sia sempre disasserito, cioè che la memoria in ingresso non sia mai vuota, a differenza di `FULL_FIFO` che viene controllato dal segnale `stop_acquire`.

Quest'ultimo segnale andando ad asserire il flag di memoria piena in uscita non permette all'encoder di procedere a nessuna acquisizione o invio di dati perché è discriminante sia nello stato `acquireBit` che `bufferedOutBit`, quindi permette di interrompendo l'acquisizione dei valori da parte dei file.

Il processo `test_proc`, invece, ha il compito di simulare il comportamento delle FIFO e salvare quello dell'encoder. Come si può osservare nel caso che il segnale `stop_acquire` venga asserito non viene eseguita nessuna operazione, è come se si scollegasse il modulo.

Si prenda in considerazione la parte di ingresso al modulo, righe da 36 a 57: ad una richiesta da parte dell'encoder (RD_EN = 1) si controlla che il numero di bit inviati sia minore o uguale della lunghezza del frame K_{ldpc}, solo in tal caso si procede alla generazione di un numero casuale mediante la funzione UNIFORM della `ieee.math_real` che restituisce un numero reale compreso tra 0 e 1 con legge uniforme. Moltiplicandolo per 2 e grazie alla funzione TRUNC, che restituisce il numero intero più piccolo al numero dato come argomento, si ottiene che il segnale `data` può essere rappresentato come una variabile aleatoria discreta uniforme di alfabeto $\{0, 1\}$. La variazione del segnale `DATA_IN` in accordo con il nuovo valore del segnale `data(0)` avviene nel periodo successivo dall'abilitazione alla lettura, come in una memoria FIFO. Il valore inviato viene anche immesso nel buffer per la successiva scrittura che avverrà quando il numero di bit inviati coincide con K_{ldpc} e la relativa chiusura del file.

Nel caso, invece, che sia l'encoder a richiedere la scrittura di un dato nella FIFO in uscita, il Testbench lo salva in un altro buffer se il numero di bit ricevuti è minore o uguale N_{ldpc}; in caso contrario scrive il contenuto del buffer nel file e lo chiude, e asserisce il segnale `stop_aquire`. In tal modo qualsiasi operazione richiesta dall'encoder viene ignorata.

Per ottenere una stima delle prestazioni si è andati a misurare il tempo che intercorre tra il primo bit inviato e l'ultimo bit ricevuto, andando a incrementare un contatore. Avviando la simulazione, alla fine dell'elaborazione del frame è possibile andare a rilevarne il tempo impiegato mostrato in Tabella 5.1.

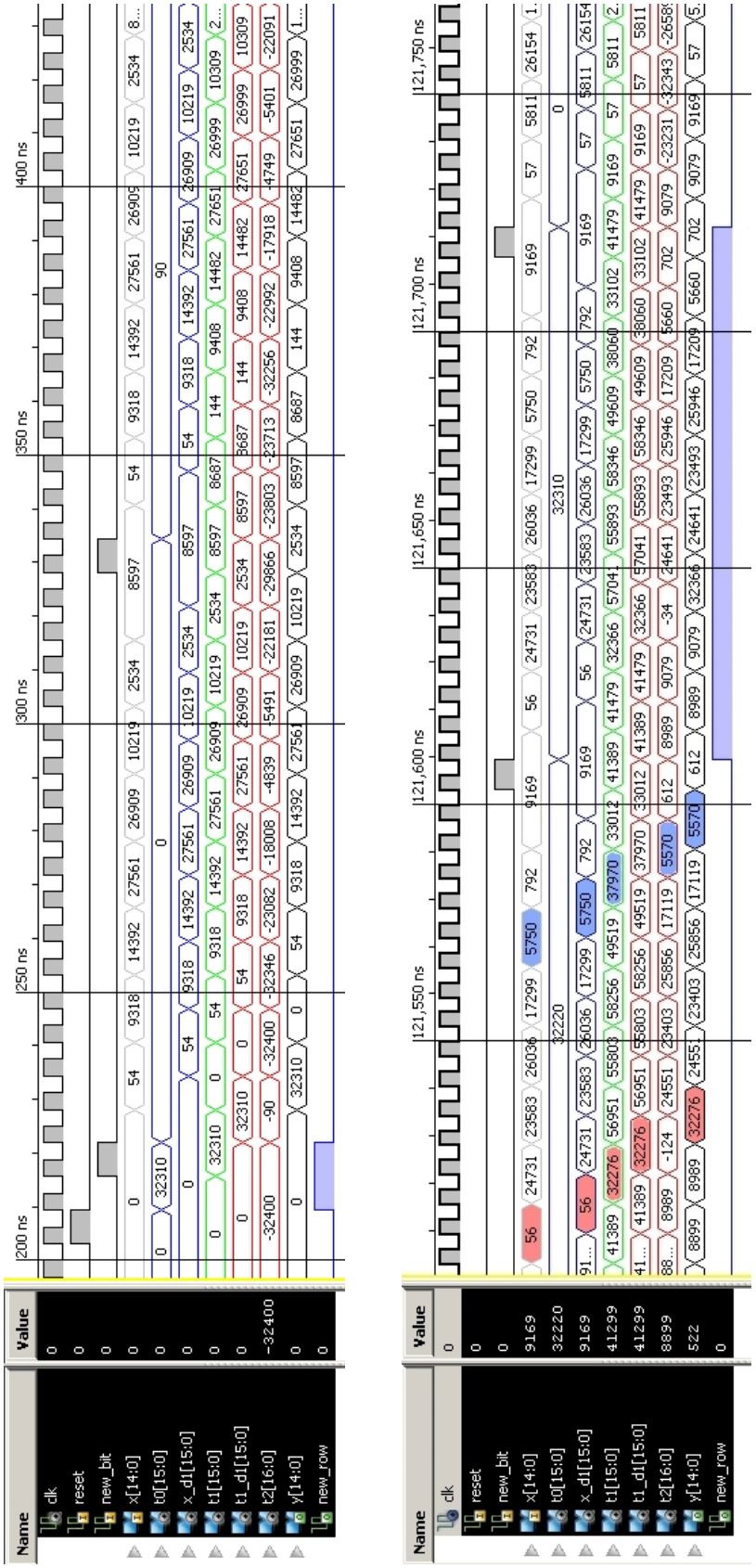
Volendo verificare il funzionamento dei moduli è preferibile avviare una simulazione Behavioral perché in questo modo i segnali interni alle entity vengano mantenuti, altrimenti nelle fasi successive si ha la realizzazione strutturale dei segnali all'interno dell'FPGA e quindi anche la perdita di eventuali segnali che son stati ottimizzati.

Per questo motivo, si riporta l'analisi del comportamento di alcuni moduli con questo tipo di simulazione.

Analisi del modulo *datapath*. Nella Figura 5.1[a] viene visualizzata la fase di inizializzazione del modulo nel quale viene inviato il segnale di reset. Infatti, dopo averlo ricevuto porta il segnale `t0` si porta al valore di Q_{MAX} in modo tale che quando venga asserito il flag di ricezione di un nuovo bit, `NEW_BIT`, questo si porti nullo. Si può osservare che il segnale `NEW_ROW` viene alzato in concomitanza con la transizione di `t0` a Q_{MAX}. Ciò è dovuto alla definizione propria del segnale dato che questo risulta asserito quando il contatore implementato arriva al suo valore massimo. Nella simulazione sono identificati con lo stesso colore i segnali con la medesima latenza, ma risulta più interessante se ci si sposta nelle elaborazioni successive, come in figura 5.1[b].

In questa immagine son stati evidenziati i percorsi che compiono due valori diversi e si verifica che effettivamente il tempo di latenza del modulo è pari a quattro, dato che impiegano quattro periodi di clock per esser riportati in uscita. Si osserva, anche, che i due segnali ad un certo punto compiono percorsi differenti: infatti nel primo si ha che `y = t1_d1` (versione ritardata di t1), mentre nel secondo si ha che `y = t2`. Il discriminante della scelta è lo stesso `t2` dato che se è non negativo vuol dire che `t1` supera il valore $N_{ldpc} - K_{ldpc}$ e, quindi, rappresenta il risultato dell'operazione, mentre se è negativo vuol dir che `t1` è il risultato finale. Ultime osservazioni che si possono fare è notare il giusto incremento di `t0` di Q_{ldpc} all'arrivo dell'impulso su `NEW_BIT` e l'asserzione del `NEW_ROW` per tutto il periodo in cui `t0` assume il suo valore massimo.

Passando all'analisi dell'intero dispositivo è necessario verificare l'evoluzione dei segnali che controllano la RAM, in modo da poter osservare come vengono elaborati i bit di parità. Per far ciò si presentano le due simulazioni di Figura 5.2 che rappresentano il funzionamento nei due stati principali in modo da comprendere meglio quanto detto nel capitolo precedente.



(a) Inizializzazione

(b) Elaborazione durante lo stato *acquireBit*

Figura 5.1: Estratto della simulazione del modulo *datapath* da iSim. Parametri: Table = A1, $Q_{ldpc} = 90$, $Q_{MAX} = 32310$.

Analisi dello stato `bufferedOutBit`. Si ricorda che in questo stato veniva implementata la struttura che generava i bit di parità da riportare in uscita: la RAM veniva percorsa fino all'indirizzo K_{ldpc}-esimo ed eseguito l'or esclusivo tra il contenuto della cella di memoria e il precedente bit se non per il primo.

I processi di color rosso rappresentano il registro interposto tra `new_DATA_OUT` e `s_DOA_d1`, visibile in Figura 4.4, ed esso impone la sua uscita nulla quando `rst_DOA` è asserito, altrimenti campiona il valore in uscita della porta xor con frequenza imposta dal segnale `wr_en_i`.

La configurazione dei segnali è tale che alla porta arrivino contemporaneamente gli ingressi perché sono sincronizzati i segnali di abilitazione del registro sopra citato e della PORTA della RAM, comportando che il segnale `new_DATA_OUT` sia anch'esso sincrono con essi.

Analisi dello stato `acquireBit`. Meno intuitivo è il funzionamento in questo stato, nel quale prendono parte tutti le altre entity istanziate. Per aver una maggior chiarezza i processi di color magenta sono inerenti al modulo *table*, quelli di color verde al modulo *datapath* e quelli in rosso alla RAM.

Prima di partire con la descrizione è appropriato vedere che il tipo di simulazione è cambiato, infatti è stata inserita una Post-Place&Route, con lo scopo di evidenziare tutti i glitch[2] e ritardi, determinati dalle linee di propagazione all'interno degli FPGA. È importante osservare come non sia possibile trovare i segnali associati alle entity come si era andati a trovarli in quelle precedenti. Esempio più lampante è il segnale `s_p_end_row_1` che è una versione negata di quello uscente dal modulo *table* definito nel codice, ma è l'unico disponibile per la visualizzazione.

Esaminando la figura, si vede come la tabella venga percorsa sulla stessa riga ricaricando il contenuto del registro `s_ADDR_reg` sul segnale `s_ADDR` e andando a incrementarlo fino a quando il flag dell'ultimo valore non venga asserito (in questo caso è portato basso). Il segnale in uscita è molto disturbato e in ritardo e questo è dovuto al modello con il quale si è realizzato il modulo *table*: una volta ricevuto l'indirizzo esso si deve propagare su tutte le ROM e successivamente i valori devono essere selezionati mediante un multiplexer. Infatti, non bisogna essere ingannati dall'immagine: sembrerebbe che i valori letti dalle tabelle vengano inviati al modulo *datapath* collegando il segnale `s_p_ADDR` al segnale `s_X`, ma si deve ricordare che tra i due vi è un registro e, quindi, approssimativamente è possibile dedurre che il periodo di propagazione è superiore perfino a metà periodo di clock.

[2]Glitch: impulso indesiderato su un segnale o un pattern errato su di un bus di breve durata.

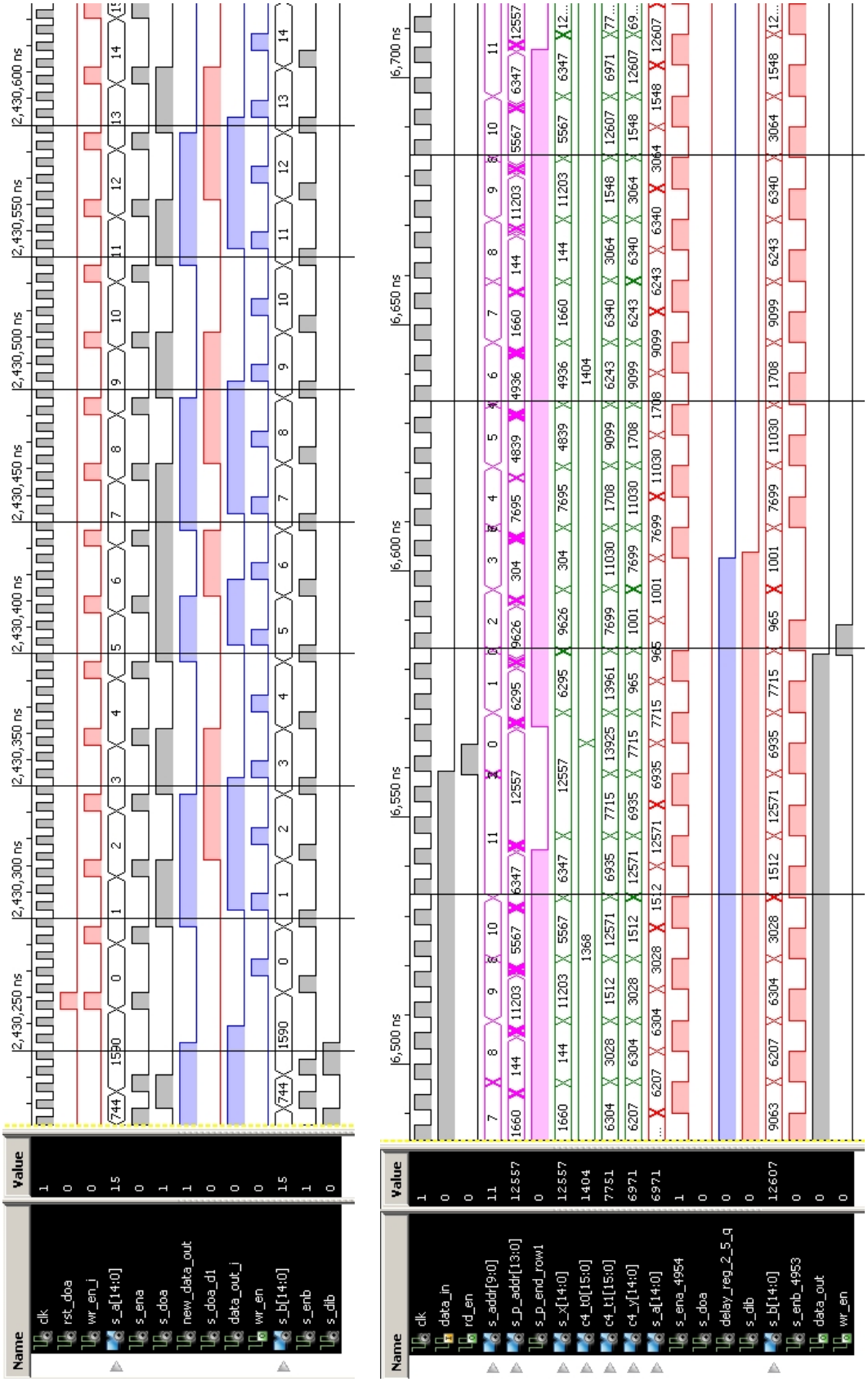
Passando i segnali all'unità aritmetica essi procedono nell'elaborazione senza particolari ritardi e glitch, arrivando alla RAM dove vengono utilizzati per indirizzare la PORTA e successivamente la PORTB dopo un ciclo di clock, in tal modo prima si legge il contenuto, poi si esegue la xor con il bit in ingresso ritardato, `delay_reg_2(5)`, e riportato in ingresso della seconda porta per essere scritto. Si nota un ritardo tra gli ingressi e l'uscita e ciò è dato dalla logica combinatoria, che consultando il design finale è realizzata mediante una LUT, dato che viene anche condizionato dallo stato in cui si trova e, quindi, non viene realizzato con una semplice porta xor.

Da questi diagrammi è possibile intuire se il comportamento del dispositivo sia quello desiderato o meno, ma per esser certi bisogna controllare se i file di uscita corrispondano ad una codifica corretta. Per far ciò si utilizza il programma *Matlab*, che fornisce un insieme di funzioni utili per eseguire la codifica, data la matrice di parità **H** e anche alcuni esempi di utilizzo forniti in [8].

Il problema principale si presenta quando si devono generare le matrici di parità perché il programma non fornisce nessuna funzione per richiamarle nel caso di un encoder LDPC DVB-T2, ma le fornisce solamente per la versione DVB-S2. In realtà alcune matrici sono identiche per i *normal FRAME*, ma non vi ne è nessuna per quelli di tipo *short*. È stato, perciò, necessario scrivere un programma che generasse le matrici H dato in ingresso la tabella, il cui codice è riportato di seguito in 5.2.

Listing 5.2: Codice Matlab di *CreateHFile.m* .

```
clear;
% Inserimento dei parametri
  N = 64800
  K = 32400
  Q = 90
  nbit = 15 %ceil(log2(p_MAX))
% Creazione della matrice sparsa H
H = sparse(N-K, N);
% File dove vi sono contenuti i valori della tabella corrente
Hfile = fopen('A1.txt', 'r');
% File dove sarà scritta la tabella formattata per la ROM
Table = fopen('Table_A1.txt', 'w');
% Iterazione del file di testo
i = 1;
while not(feof(Hfile))
  % Linea del file di testo
  Hstr = fgetl(Hfile);
  % Conversione della stringa in vettore numerico
  Hline =sscanf(Hstr, '%u');
  % Definizione del vettore per la ROM
  Hrow = Hline;
  % Incremento dell'ultimo valore della riga
```



(a) Tipo: Behavioral, Table = A1, $Q_{ldpc} = 90$, $Q_{MAX} = 32310$.

(b) Tipo: Post-Place&Route, Table = B1, $Q_{ldpc} = 36$, $Q_{MAX} = 12924$.

Figura 5.2: Estratto delle simulazione del modulo *ldpc_encoder* nei due stati: `bufferedOutBit`**(a)** e `aquireBit`**(b)**.

```
23    Hrow(length(Hrow)) = Hrow(length(Hrow))+ 2^nbit;
24    % Stampa della riga sul file
25    fprintf(Table, '%u, ', Hrow);
26    fprintf(Table, '\n');
27    % Composizione della matrice H con il bit i-esimo per la
          sottomatrice che concerne ai bit d'informazione
28    for n = 0:359
29      H(mod((Hline+n*Q),N-K)+1,i) = 1;
30      i = i+1;
31    end
32  end
33  % Chiusura di tutti i file
34  fclose('all');
35  % Composizione della matrice H per la sottomatrice dei bit di
       parità
36  H(1,K+1) = 1;
37  for n = 2:N-K
38      H(n,[K+n-1, K+n]) = 1;
39  end
40  % Salvataggio nel file Matlab della matrice H ottenuta
41  save A1.mat H
```

Il programma, vista la necessità anche di avere anche un elaboratore di testi per generare le tabelle da inserire nelle ROM, realizza anche un file dove si incrementa l'ultimo valore della riga secondo la convenzione descritta in 4.2. Dato che $Hc^T = 0$, dove $\mathbf{H}$ è la matrice di parità di dimensioni $(N_{ldpc} - K_{ldpc}) \times N_{ldpc}$ e $\mathbf{c}$ è il vettore riga codificato di N_{ldpc} bit, la costruzione di H deve esser fatta in modo tale che verifichi se il bit di parità sia corretto rispetto ai bit d'informazione precedenti.

Dalla prima parte dell'algoritmo utilizzato si deduce che gli indirizzi forniti dalla formula per ogni bit i-esimo corrispondano agli indici di riga nella quale la matrice $\mathbf{H}$ nella colonna i-esima è non nulla. Nella seconda, invece, si apprende che ogni bit di parità è sommato con quello precedente e per questo motivo per essere verificato nella matrice $\mathbf{H}$ sarà non nullo, sia la locazione del bit di parità corrente che quella precedente. Per esser più chiari si prenda in considerazione il bit p_1 nella codifica di tabella A1

$$p_1 = i_{3899} \oplus i_{3910} \oplus i_{7274} \oplus i_{13320} \oplus i_{14966} \oplus p_0.$$

Analogamente nella matrice di parità si avrà che $H[1, 3899] = H[1, 3910] = H[1, 7274] = H[1, 13320] = H[1, 14966] = H[1, 32400] = H[1, 32401] = 1$ dato che eseguendo il prodotto riga per colonna e ricordando la forma di $\mathbf{c}$,3.1, si ottiene :

$$\begin{aligned} H[1]c^T &= i_{3899} \cdot 1 \oplus i_{3910} \cdot 1 \oplus i_{7274} \cdot 1 \oplus i_{13320} \cdot 1 \oplus i_{14966} \cdot 1 \oplus p_0 \cdot 1 \oplus p_1 \cdot 1 \\ &= \underbrace{i_{3899} \oplus i_{3910} \oplus i_{7274} \oplus i_{13320} \oplus i_{14966} \oplus p_0}_{p_1} \oplus p_1 = p_1 \oplus p_1 = 0 \end{aligned}$$

Da queste considerazione si ottiene il codice del file *CreateHfile.m* ma bisogna tener conto che *Matlab* ha una indicizzazione delle matrici diversa dalla convenzione utilizzata dalle tabelle. Essendo che la generazione di tali matrici è abbastanza lunga come computazione, conviene salvarle per non dover generarle ogni volta che si vuole eseguire un test.

Sfruttando l'esempio trovato in [8], adattandolo per le nuove codifiche e per l'interfaccia con i file uscenti dal Testbench, si scrive il codice del file *Test.m*. Questo genera un file contenente la codifica elaborata da *Matlab* e restituisce a video tre principali controlli: la verifica che il file generato sia corretto, che la codifica proveniente dal Testbench sia esatta e che i due file siano identici.

Listing 5.3: Codice Matlab di *Test.m* .

```
clear;
% Apertura dei file ottenuti dalla simulazione con iSim
infile = fopen('infile.txt', 'r');
outfile = fopen('outfile.txt', 'r');
% File della codica ottenuta mediante Matlab
codewordfile = fopen('codeword.txt', 'w');
% Conversione dei file di iSim in vettori
datain = fscanf(infile, '%d');
dataout = fscanf(outfile, '%d');
% Caricamento dal file ottenuto con CreateHFile.m della matrice
    di parità H
load A1.mat
% Definizione dell'encoder
l = fec.ldpcenc(H);
% Stampa del numero di bit d'informazione e della lunghezza del
    frame finale
NumInfoBits = l.NumInfoBits
BlockLength = l.BlockLength
% Codifica del frame in ingresso mediante l'encoder di Matlab e
    stampa su file
codeword = encode(l, transpose(datain));
fprintf(codewordfile, '%u', codeword);
% Verifica del controllo di parità(vettore nullo) con la codifica
     di Matlab
paritychecks = mod(l.ParityCheckMatrix * codeword', 2);
z = zeros(1, l.BlockLength-l.NumInfoBits);
ParityChecksWithCodeWord = isequal(paritychecks', z)
% Verifica del controllo di parità con la codifica ottenuta dal
    modulo ldpc_encoder
paritychecks = mod(l.ParityCheckMatrix * dataout, 2);
ParityChecksWithDataOut =isequal(paritychecks', z)
% Verifica che i due frame ottenuti siano uguali
CodeWordIsEqualDataOut = isequal(codeword, transpose(dataout))
% Chiusura di tutti i file
fclose('all');
```

Dalla verifica di tutte le possibile codifiche si ottiene che il progetto funziona correttamente. Per ulteriori informazioni sulle funzioni di *Matlab* è consigliato

consultare le guide del programma.

5.2 Commenti sulla sintesi

In questa sezione viene visualizzato il prospetto fornito dall'ISE dopo l'implementazione del modulo *ldpc_ encoder* in Figura 5.3. Questo risulta molto utile per comprendere quale sia l'entità delle risorse utilizzate per realizzare l'encoder all'interno del FPGA, dato che normalmente dovrà essere integrato con altre strutture per formare l'intero modulatore.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	293	20,480	1%	
Number of 4 input LUTs	537	20,480	2%	
Number of occupied Slices	343	10,240	3%	
Number of Slices containing only related logic	343	343	100%	
Number of Slices containing unrelated logic	0	343	0%	
Total Number of 4 input LUTs	576	20,480	2%	
Number used as logic	535			
Number used as a route-thru	39			
Number used as Shift registers	2			
Number of bonded IOBs	13	320	4%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number of FIFO16/RAMB16s	15	128	11%	
Number used as RAMB16s	15			
Average Fanout of Non-Clock Nets	2.75			

Figura 5.3: Rapporto della sintesi ottenuta dall'ambiente ISE.

Dall'analisi dei valori percentuali si può osservare che l'utilizzo del modello di FPGA scelto è minimo se non per l'uso delle RAMB16, che corrispondono alle 18Kb Block RAM del dispositivo.

Utilizzando *Xilinx FPGA Editor*, un programma che permette di vedere la reale istanziazione del modulo all'interno del FPGA, si può osservare le 15 Block RAM utilizzate sono così distribuite:

- 2 per generare la RAM del vettore di parità: con parole da un bit è possibile configurarle solo con profondità pari a 16384. Per ottenere il vettore di dimensione 32400 è necessario, quindi, metterle in cascata in modo da raggiungere una profondità di 32768. Come è possibile osservare da [13][pag. 125], se la larghezza della parola è 1,2 o 4 viene implementata per memorizzare 16Kb, altrimenti nel caso di 9,18,36 si utilizzano interamente i 18Kb di memoria disponibile.

- 13 per implementare le ROM associate alle tabelle: tutte esse vengono implementate con larghezza di parola a 18 bit e profondità 1024.

Non vengono implementate con le Block RAM le ROM associate al modulo *parameters*. Infatti nella sintesi, dato che sono di dimensione molto ridotta, vengono implementate con delle LUT. Queste ottimizzazioni sono dovute alla scelta di implementare le memorie mediante un modello VHDL, evitando i moduli IP che avrebbero forzato la loro realizzazione mediante Block RAM.

5.3 Prestazioni del modulo

Come accennato nella sezione iniziale, per stimare le prestazioni è possibile andare a determinare il tempo di elaborazione di un frame. Nella Tabella 5.1 sono riportati i valori determinati mediante Testbench.

Stabilire un termine di paragone non è facile perché non si conoscono i relativi rate degli altri moduli, per esempio dell'encoder BCH che si trova a monte dell'encoder LDPC. In [9][pag. 386] vi è riportata una tabella con i bit rate dei principali standard video e soffermandosi su quelli utilizzati per il broadcast se ne riportano 2: MPEG2 TV con 6Mbit/s e HDTV con 24Mbit/s. Nel documento [10] si parla solo del primo standard, quindi si ha un buon margine per tutte le codifiche realizzate, a differenza del secondo che non lo è rispettabile per nessuna. Nello stesso documento però si cita anche che il massimo throughput rate[3] in ingresso che è di 50Mbit/s e ciò comporterebbe pesanti limitazioni al progetto.

Tabella 5.1: Prestazioni del modulo *ldpc_ encoder*: stima del *bitrate* in ingresso e in uscita.

Table	n_{clk}	nT_{clk} [μs]	$\widehat{bitrate_{IN}}$ [Mbit/s]	$\widehat{bitrate_{OUT}}$ [Mbit/s]
A1	518407	3370	9,6	19,2
A2	648007	4212	9,2	15,2
A3	518407	3370	12,8	19,2
A4	550807	3580	13,6	18,1
A5	570247	3707	14,0	17,5
A6	583207	3791	14,2	17,1
B1	103687	674	4,8	24,0
B2	111607	725	9,9	22,3
B3	136087	885	11,0	18,3
B4	129607	842	12,8	19,2
B5	118807	772	15,4	21,0
B6	115207	749	16,8	21,6
B7	125287	814	16,4	19,9

[3]Indica la massima quantità di dati che è possibile inviare su di un canale trasmissivo perché questi vengano ricevuti correttamente.

Capitolo 6

Conclusioni

In questo capitolo si analizzano i risultati ottenuti, verificando se vi siano delle incongruenze con gli obiettivi prefissati e si spiegano le varie difficoltà e problematiche riscontrate durante lo sviluppo del progetto. L'ultima sessione è adibita all'esposizione di eventuali linee guida per il miglioramento del progetto.

6.1 Obiettivi raggiunti

L'obiettivo prefissato consisteva nella realizzazione di un encoder LDPC DVB-T2 che potesse esser configurato per i diversi tipi di code rate disponibili per il nuovo standard, per poi esser inserito nel modulatore del digitale terrestre di seconda generazione. Vanno ricordate le scelte progettuali iniziali:

- **sviluppo del progetto interamente in codice VHDL** in modo da esser portabile su più piattaforme a differenza dei moduli IP, i quali dovrebbero essere rimplementati ogni volta che si cambiasse dispositivo;
- **scelta di architetture pipeline** per scomporre sommatori o comparatori in cascata così da poter mantenere elevata la frequenza di funzionamento;
- **architetture semplici nell'unità aritmetica** evitando l'utilizzo di moltiplicatori o divisori che potrebbero comportare latenze elevate;
- **sincronismo dei segnali utilizzando shift register** così facendo si riduce notevolmente l'intervento della FSM semplificandola notevolmente.

Tutte le specifiche son state rispettate e si è potuto ottenere un'implementazione che potrebbe soddisfare alcune esigenze non troppo performante dal punto di vista della velocità di elaborazione.

Per come è stato costruito il dispositivo, risulta molto facile anche modificarlo per essere utilizzato in altri altri ambiti, per esempio in quello satellitare DVB-S2,

dove comunque è richiesto un encoder LDPC, ma con tabelle e parametri diversi. Andando a cambiare i valori contenuti nelle tabelle e nei parametri, e adattando la larghezza dei segnali e la profondità della RAM in base al frame, si ottiene, quindi, l'implementazione per un ulteriore standard.

6.2 Problemi riscontrati

I problemi incontrati riguardano maggiormente l'aspetto di simulazione e controllo del dispositivo.

Nella simulazione è stato quasi impossibile realizzarne completamente una di tipo Post-Place&Route data la mole di tempo che richiederebbe. Considerando che per la codifica B1, cioè quella che richiederebbe meno tempo, si è stimato che un'intera simulazione durerebbe circa ventisette giorni, si è scelto di avviarne una e fermarla dopo l'acquisizione di una trentina di bit in modo da poterla confrontare con una di tipo Behavioral.

Dato che le forme d'onda corrispondevano, per le simulazioni successive ci si è basati solo su quella comportamentali.

Ulteriore problema è stato trovar il modo di eseguire la verifica e la creazione di delle tabelle. All'inizio si era pensato di creare un compilatore di testi in *Java* per generare le ROM, e utilizzare *Matlab* per svolgere il controllo. Quest'ultimo, però, non aveva le matrici di parità per lo standard utilizzato e, quindi, dovendo anche generarle si è pensato di eseguire tutto all'intero dello stesso programma.

Il progetto in sè non presenta nessuna problematica rispetto alle specifiche date, l'unica nota negativa potrebbero essere le prestazioni e la presenza di un solo tipo di interfaccia I/O.

6.3 Sviluppi ulteriori

Per migliorare le prestazioni del dispositivo è possibile eseguire alcune modifiche sul progetto in modo da ottimizzarlo, arrivando anche a prestazioni non indifferenti, a discapito però di un maggiore consumo di risorse e aumento della complessità del controllo.

Innanzitutto, si deve ridurre l'impatto del modulo *table* sull'intera struttura dato che è quella con il tempo di propagazione più elevato e quindi che impedisce di diminuire ulteriormente il perdio di clock, come si è poturo vedere dalle simulazioni.

Possibili soluzioni possono esser la riduzione del Fan-out del bus indirizzi che comanda le tabelle, infatti questo deve caricarle tutte e tredici e seppur quando viene sintetizzato vengono inseriti dei buffer, sarebbe stato opportuno forse gestirli

mediante dei buffer tri-state che portassero l'ingresso e l'uscita in alta impedenza, formando così anche un bus in uscita ed evitando di inserire il multiplexer.

Altra soluzione potrebbe essere quella di inserire una memoria temporanea tra le tabelle e la FSM. Infatti, se ogni riga viene caricata in dei registri, per la prima lettura essa sarà molto lenta perché dovrà attendere la scrittura nella memoria cache rispettando le tempistiche del modulo *table*, quindi attendendo anche più periodi di clock per indirizzo, ma gli altri 359 potranno essere eseguiti accessi ad una velocità nettamente superiore.

Con queste due accortezze si potrebbe, quindi, implementare un dispositivo che lavora a frequenze più elevate di quella attuale. Spostandoci ora sul numero di operazioni da eseguire gli interventi che si possono effettuare sono due essenzialmente.

Per primo si potrebbero inserire più encoder LDPC in parallelo, eseguendo una TDM(Time Division Multiplexing) sui vari frame. Consiste nel dividere la successione di frame nei vari encorder, così mentre se ne sta elaborando uno, il successivo sta acquisendo il nuovo frame e così via, fino tanto che un encoder non ha terminato l'elaborazione e, quindi, ripartire a caricare il frame a questo. In uscita praticamente vengono svuotate le FIFO con la medesima successione.

Questa soluzione è molto dispendiosa come risorse perché richiede di implementare più encoder e anche di generare la struttura che gestisca lo scambio.

Un'altra soluzione interessante sarebbe quella di modificare la FSM di controllo delle strutture. Infatti, essa attende sempre che in uscita sia possibile inviare un dato e, quindi, deve aspettare che il bit inviato si propaghi fino alla FIFO avendo dei tempi morti.

Per ridurre questo problema si potrebbero inserire altri ingressi di controllo quali **ALMOST_FULL** o **PROG_FULL** che sono altri due flag che sono gestiti dalla FIFO. Il primo è asserito quando più della metà della memoria è piena, mentre il secondo quando si supera una soglia programmabile. Utilizzando il secondo segnale e impostando la soglia ad un valore prossimo al valor massimo, sarebbe possibile far in modo che la FSM inviasse i segnali in maniera continuativa finché tale flag non fosse asserito, così evitando di aspettare che essi si propaghino alla FIFO. Se invece si asserisse, comparirebbe la struttura di controllo realizzata in questo progetto con i relativi ritardi.

Così facendo per la maggior parte del tempo la struttura lavora continuamente, potendo diminuire anche gli stati per l'elaborazione e quindi anche il tempo di esecuzione.

Combinando queste modifiche è possibile ottenere un notevole miglioramento del dispositivo.

Bibliografia

[1] M. Barbero and N. Shpuza. (2010) Rivelazione, correzione e mascheramento degli errori. [Online]. Available: http://www.crit.rai.it/

[2] DVB, "Implementation guidelines for a second generation digital terrestrial television broadcasting system (dvb-t2)," *DVB Document A133*, 2009-02.

[3] ——. (2010) Digital terrestrial television (fact sheet). [Online]. Available: http://dvb.org/technology/fact_sheets/

[4] ——. (2011) 2nd generation terrestrial, the world's most advanced digital terrestrial tv system (fact sheet). [Online]. Available: http://dvb.org/technology/fact_sheets/

[5] ——. (2011) Dvb worldwide. [Online]. Available: http://dvb.org/about_dvb/dvb_worldwide/index.xml

[6] ——. (2011) History of the dvb project. [Online]. Available: http://dvb.org/about_dvb/history/l

[7] R. G. Gallager, *Low-Density Parity-Check Codes*. M.I.T. Press, 1963.

[8] MATLAB. (2010) Construct a ldpc encoder object. [Online]. Available: http://www.mathworks.com/help/toolbox/comm/ref/fec.ldpcenc.html

[9] N.Benvenuto, R.Corvaja, T.Erseghe, and N.Laurenti, *Communication System, Fundamentals and Design Methods*. Wiley, 2007.

[10] E. S. T. series, "Digital video broadcasting (dvb); frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (dvb-t2)," *ETSI EN 302 755 V1.1.1*, 2009-09.

[11] D. Vogrig, "Dispense del corso di laboratorio di elettronica digitale," 2009.

[12] Xilinx, *Virtex-4 Family Overview*, August 30, 2010. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-4.htm

[13] ——, *Virtex-4 FPGA User Guide*, December 1, 2008. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-4.htm

Ringraziamenti

Un particolare ringraziamento va al prof. Daniele Vogrig per la sua professionalità, disponibilità e pazienza, che sono risultate sempre costanti in questi mesi.

Desidero ringraziare la mia famiglia, gli amici e tutti quelli che sono risultati sempre presenti nel darmi sostegno e nello spronarmi a realizzare tutto ciò.