Tesina di Sistemi Embedded
Gruppo IV

Colella Gianni - Mat. M63/670 Guida Ciro - Mat. M63/592
Lombardi Daniele - Mat. M63/576

18 luglio 2017

Indice

Capitolo 1

Progetto finale Task 4

1.1 Premessa

Il Task 4 rappresenta il progetto finale da realizzare per l'esame di Sistemi Embedded del corso di Laurea Magistrale in Ingegneria Informatica, a. a. 2016/2017. Esso è inserito all'interno di un progetto più ampio, sviluppato in collaborazione con l'azienda Aster, che prevede l'implementazione su FPGA di un Radar Bistabile Passivo. A differenza dei radar attualmente utilizzati, detti attivi, questo tipo di radar non fa uso dell'apparato trasmissivo. Per adempiere alle proprie funzioni, questo dispositivo, non trasmettendo segnali, fa uso di segnali già presenti nell'etere. In particolare, sono stati scelti in fase progettuale i segnali utilizzati dal sistema **G**lobal **N**avigation **S**atellite **S**ystem **G**alileo. Il sistema si compone di 3 moduli principali:

1) Fase di Aquisizione;
2) Fase di Tracking;
3) Fase di Compressione.

Il Task in esame, incapsulato nella fase di Tracking, utilizzando i segnali **EaryGate** e **LateGate**, rappresentazioni del segnale primario anticipato e ritardato, fig. 1.1, fornisce una stima del valore **r**. Analizzando quest'ultimo e confrontandolo con una opportuna soglia, si è in grado di stabilire se l'oggetto, di cui si vuole conoscere la posizione, si sta allontanando o avvicinando dalla fonte trasmissiva del segnale. Viene, per questo motivo, utilizzato per correggere i valori di Delay, Frequenza Doppler e Fase.

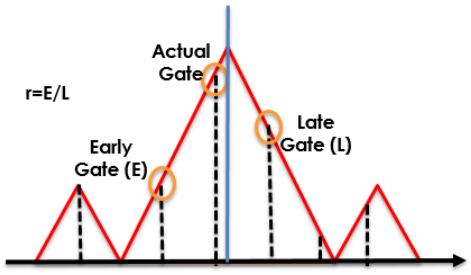

Figura 1.1: Early Gate e Late Gate

1.2 Traccia

Si realizzi un IP core che implementi il raffinamento del calcolo del delay che avviene durante la fase di Tracking, relativamente alla seconda parte durante la quale bisogna effettuare i moduli delle sommatorie, ottenute durante lo step precedente, calcolarne il rapporto e ricavarne la radice quadrata.

Per la realizzazione del task si richiede l'implementazione di:

1. moltiplicatore;
2. sommatore;
3. divisore;
4. radice quadrata.

Il Task, dunque, può essere rappresentato seguendo lo schema di principio di fig.1.2

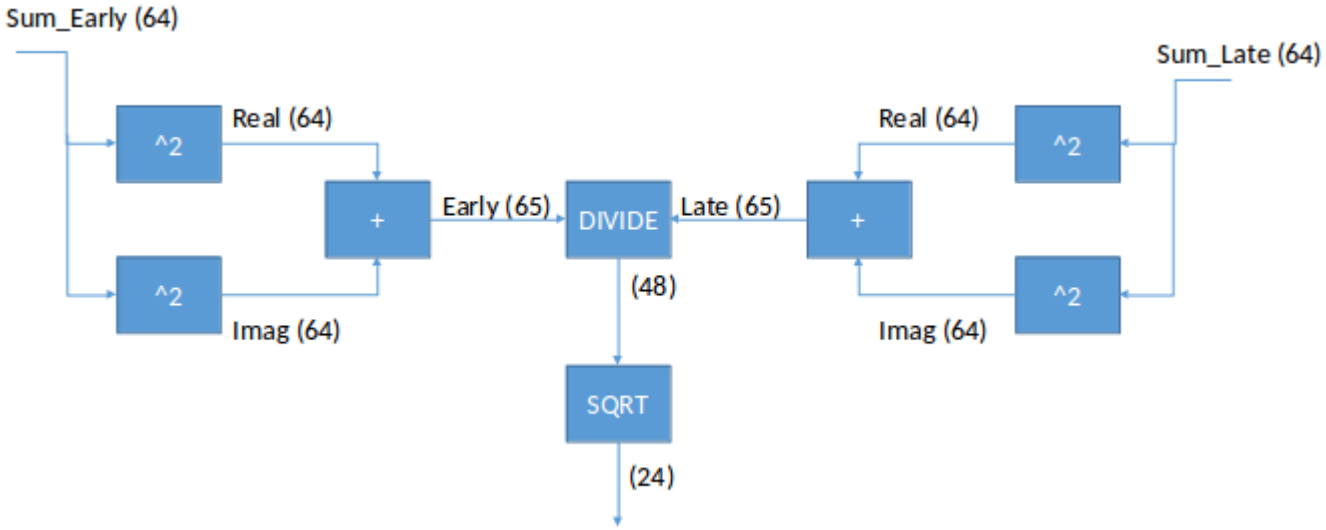

Figura 1.2: Schema di principio

L'obiettivo è quello di dispiegare tale componente su board FPGA Zybo.

1.3 Rappresentazione segnali e codice MATLAB

Il Task in esame accetta in ingresso due numeri complessi signed espressi su 64 bit che rappresentano i segnali **Sum_Early** e **Sum_Late** provenienti dal Task precedente. Essi devono subire una serie di manipolazioni per restituire in uscita un segnale signed fixed point espresso su 24 bit. Con riferimento allo schema di principio visto in precedenza, fig.1.2, il primo stadio è rappresentato da due componenti **Modulo Quadro**, fig. 1.4, che effettuano parallelamente il calcolo per i segnali Sum_Early e Sum_Late. Più precisamente, presi gli ingressi, ogni componente divide il segnale in parte reale e parte immaginaria, fig. 1.3. Ogni parte è trattata separatamente da un moltiplicatore.

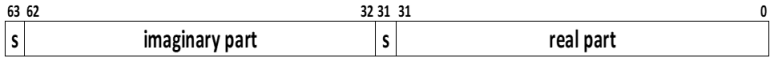

Figura 1.3: Rappresentazione dei segnali in ingesso

In uscita ogni moltiplicatore restituisce un segnale signed espresso su 64 bit. Le uscite dei due moltiplicatori sono gli ingressi di un sommatore che restituisce un segnale signed, espresso su 65 bit. Esso rappresenta il valore del modulo quadro del numero complesso. Tale segnale è signed, ma, considerato il fatto che la somma di due valori positivi è sicuramente positiva, si può elidere il bit più significativo del risultato che rappresenta il segno, e, dunque, viene considerato un intero unsigned espresso su 64 bit.

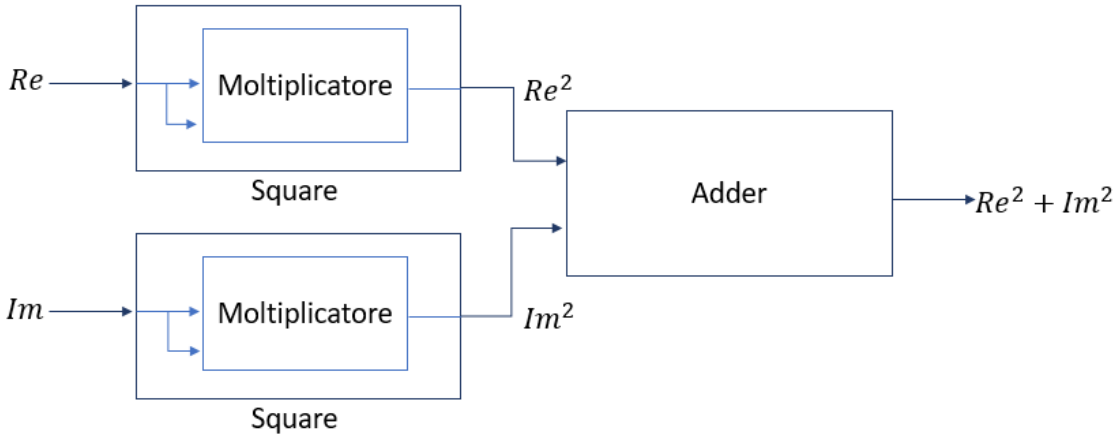

Figura 1.4: Componente Modulo Quadro

Succesivamente, i due segnali in uscita al primo stadio vengono processati da un divisore, fig. 1.5, il quale, secondo le specifiche fornite, restituisce un segnale rappresentato come fixed point a 48 bit, di cui gli 8 più significativi rappresentano la parte intera, i restanti 40 la parte decimale.

Figura 1.5: Componente Divisore

Quest'ultimo, dunque, diviene l'input da fornire all'ultimo stadio della catena, rappresentato da un componente che effettua il calcolo di **r**, 1.6. Il componente Radice quadrata fornisce in uscita un segnale unsigned rappresentabile su 24 bit dove i 4 bit più significativi rappresentano la parte intera, gli ultimi 20 quella decimale, fig. 1.3. Tale rappresentazione del segnale in uscita è stata presa in considerazione nella prima versione del Task proposta. Nelle successive versioni, a causa di un adeguamento delle specifiche di progetto, viene presa in considerazione una diversa rappresentazione del segnale di uscita basata su 24 bit signed, di cui 13 parte intera e 11 parte decimale, fig. 1.8.

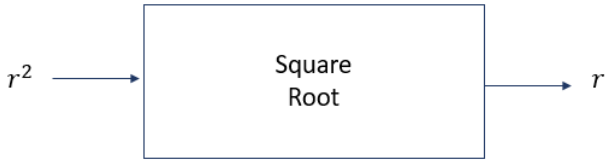

Figura 1.6: Componente Radice Quadrata

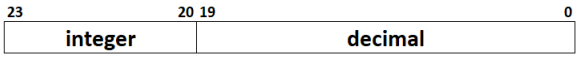

Figura 1.7: Rappresentazione di r <24,20>

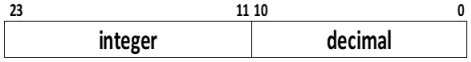

Figura 1.8: Rappresentazione di r <24,11>

Di seguito, si riportano due frammenti di codice MATLAB in cui sono inglobate le operazioni appena descritte.

```
64 %% Delay Deviation Estimation
65 for bbb = 1:nr_block
66 index_P=(1+(bbb-1)*sample_in_P:bbb*sample_in_P);% indeces of the samples in
       the primary code
67 Data_preConditioned(index_P)= Data_plus_Noise_Block(index_P).*DRR_in_P.*
       DRR_in_B(bbb)*SS_a_p(bbb); % secondary code stripping and doppler
       removal
68 r(bbb) = abs(sum(Data_preConditioned(index_P).*S_P_Early_1))/ ...
69         abs(sum(Data_preConditioned(index_P).*S_P_Late_1));  % gating
70 end
71 r_avg(bb) = mean(r);
```

Codice 1.1: Test2DelayDeviationAndAlignment.m

```
62 % Compute operations
63 reE2=real(sigEarly).^2;
64 imE2=imag(sigEarly).^2;
65 reL2=real(sigLate).^2;
66 imL2=imag(sigLate).^2;
67 sE=reE2+imE2;
68 sL=reL2+imL2;
69 d1=sE./sL;
70 R=sqrt(d1);
```

Codice 1.2: "T4dataGenerator.m"

1.4 Task 4 v1.0

Tale soluzione, che è anche quella più immediata da realizzare, consiste nell'utilizzare solamente IP core realizzati da terze parti: essi vengono forniti direttamente da Xilinx, gratuitamente e presenti nella suite di sviluppo Vivado.

1.4.1 Design

La soluzione prevede di istanziare i seguenti IP core :

- 4 **Multiplier** e 2 **Adder/Subtractor**, per realizzare i primi due componenti paralleli;
- 1 **Divider Generator**, per realizzare l'operatore di divisione;
- 1 **Cordic**, per realizzare l'operatore di radice quadrata.

La fig. 1.9 mostra le istanze di tali componenti e relativi collegamenti tra essi.

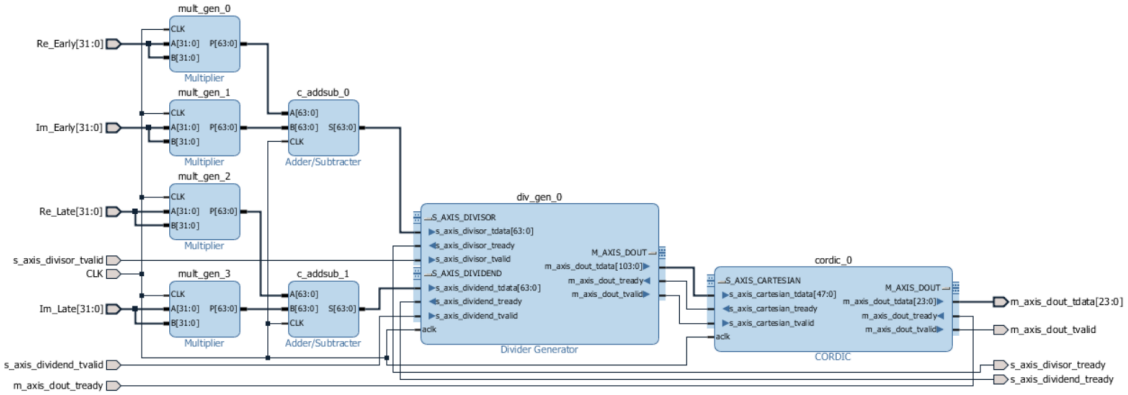

Figura 1.9: Block Diagram relativo ad una prima soluzione

1.4.2 Testbench

Per verificare il funzionamento, viene eseguito un semplice testbench, fig. 1.10, fornendo in input al Task una coppia di dati generati in MATLAB. Si può notare come il risultato finale viene restituito in uscita dopo 136 cicli di clock, impostando idealmente un clock in ingresso a 100 Mhz. In questa implementazione, l'uscita r è espressa in una forma fixed point unsigned <24,20>.

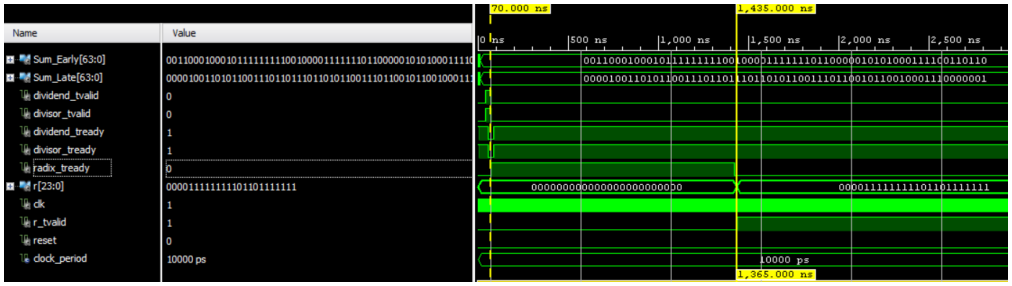

Figura 1.10: Testbench

1.4.3 Analisi soluzione

La soluzione adottata può essere analizzata considerando 2 parametri fondamentali: area occupata, frequenza massima di lavoro del circuito.

Per quanto riguarda l'area occupata dal componente complessivo, viene fornita la seguente tabella che riassume l'occupazione dovuta ai singoli IP core dispiegati, nonché quella relativa all'intero Task.

Componente	LUT	Slice Register	DSP48
Multiplier	1103	64	0
Adder/Subtractor	63	64	0
Divider generator	2036	4474	0
Cordic	673	892	0
Totale	7248(41%)	5750(16%)	0(0%)

Tabella 1.1: Occupazione d'area post-sintesi

Effettuando un'analisi post-implementation, fornendo ad ogni componente un constraint fisico temporale (clock dell'FPGA) si scopre che il modulo che lavora a più basse frequenze, è quello che implementa l'operazione di modulo quadro, che accetta una frequenza massima poco inferiore ai 70 Mhz.

1.4.4 Vantaggi e svantaggi

Essendo la soluzione totalmente composta da IP core Xilinx, si ottengono una serie di vantaggi come la velocità di dispiegamento della soluzione, l'affidabilità di ogni componente utiizzato, infine, trattandosi appunto di proprietà intellettuali è significativa la facilità nel riuso dei componenti.

Di contro, poiché il primo stadio della catena, a differenza dei successivi, non è dotato di interfaccia AXI Stream, è necessario gestire esternamente i segnali di tvalid e tready sull'interfaccia slave del Divider Generator. Inoltre, dalla tab.1.1, è possibile notare come la soluzione proposta occupa molte risorse.

1.5 Analisi singola su ogni componente

In questa sezione si propone una panoramica su ogni componente della pipeline, prendendo in considerazione per ognuno di essi una serie di alternative e cercando di trovare una combinazione "vincente" in termini di area e spazio. Tutto questo non implica che la prima soluzione proposta è da scartare a priori.

La seguente tabella mostra per ogni componente le soluzioni prese in considerazione.

Per ogni singola proposta, viene effettuato uno studio in termini di occupazione d'area, frequenza massima di lavoro e numero di cicli di clock necessari per avere il risultato dell'operazione.

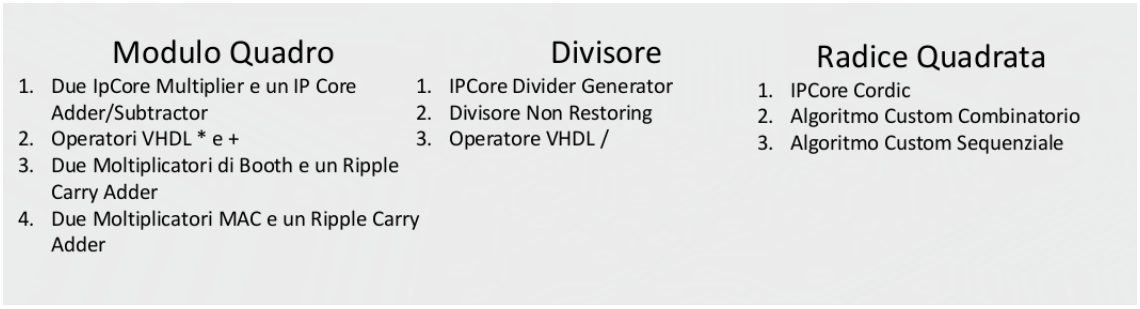

Figura 1.11: Componenti analizzati

1.5.1 Modulo quadro

Così come visto in tab.**??**, per tale componente vengono prese in considerazioni quattro possibili soluzioni. Si ricorda che tale componente, nella catena di elaborazione del task, è quello istanziato nel primo stadio della catena, dove parallelamente vengono calcolati i moduli quadri dei segnali Sum_Early e Sum_Late.

Realizzazione	LUT	Slice Register	DSP48	F.max	Cicli di clock
2 Multiplier Xil + Add/Sub Xil	1283	106	0	89.896 MHz	2
* e +	158	0	8	80.901 MHz	2
2 Booth + Ripple Carry Adder	425	264	0	82.967 MHz	128
2 MAC + Ripple Carry Adder	5468	128	0	20.927 MHz	2

Tabella 1.2: Occupazione d'area post-implementation

Dalla tabella si evince come il componente migliore in termini di occupazione sia quello relativo alla sua descrizione dataflow, in quanto il sintetizzatore **UG901** di Vivado, accorgendosi di operatori matematici, li va ad inferire sui DSP.

Per quanto riguarda invece le prestazioni, la scelta del migliore ricade su quello composto da IP core Xilinx con una frequenza massima di lavoro poco inferiore ai 90 MHz. Poiché 3 soluzioni su 4 sono puramente combinatorie, bufferizzando opportunamente ingressi e uscite, il numero di cicli di clock necessari ad avere il risultato pronto è pari a 2. Tranne per la soluzione con moltiplicatore a celle MAC che risulta essere la scelta nettamente peggiore in termini di area e prestazioni, le restanti 3 potrebbero essere prese tranquillamente in considerazione per la realizzazione del Task. Trovando un buon compromesso tra area e frequenza di lavoro, il componente migliore risulta essere quello relativo alla sua descrizione dataflow.

1.5.1.1 Aggiunta macchina a stati

Avendo scelto un componente combinatorio per il calcolo del modulo quadro nel primo stadio, per ovviare ad uno dei problemi visti nella prima proposta di soluzione, è necessario aggiungere una parte di controllo in modo tale che il componente abbia un'interfaccia del tutto compatibile con un bus AXI Stream, almeno per quanto riguarda i segnali tdata, tvalid e tready. Dunque si wrappa il componente e si aggiungono degli ulteriori segnali, oltre a quelli già presenti. Idealmente, seguendo la filosofia di un generico componente AXI Stream, l'interfaccia black box viene differenziata in Slave e Master. Sull'interfaccia slave vengono posti 3 segnali di cui 2 in ingresso ed 1 in uscita,

rispettivamente di *tdata*, di *tvalid* e di *tready*. Il segnale di input *tvalid* indica il fatto che se esso è asserito, il dato in ingresso è valido e la macchina può iniziare ad elaborarlo; viceversa, il segnale di output *tready*, se pari ad 1, indica che la macchina è pronta ad accettare un nuovo dato, affinché possa essere processato. Sull'interfaccia master invece vi sono 2 segnali in output e 1 di input, in particolare oltre al dato in uscita vi è lo stesso segnale *tvalid* come sullo slave che indica un dato pronto in uscita ed uno *tready*, il quale indica che un'eventuale dato in uscita è pronto ad essere accettato dal componente posto a valle della catena.

Entrando nei dettagli, la parte di controllo è interpretata come FSM composta da 4 stadi e può essere descritta come segue: a partire dallo stato inziale di **RESET**, essa vi permane finché l'eventuale segnale di reset non viene posto a 1, facendo transitare la macchina nello stato di **IDLE**. In questa fase si attende che il segnale di input *tvalid* divenga 1 in modo tale che la macchina combinatoria processi un dato valido, memorizzato appositamente in un buffer di input. Nel transire la macchina passa per lo stato di **RESULT_CALCULATION**, dove viene abilitato il buffer in uscita per la memorizzazione del risultato. Successivamente, si entra nello stato di **WAIT_M_TREADY** in cui si attende che il componente a valle segnali che esso è pronto ad accettare un nuovo valore, quindi, se il segnale di ready è pari a 1, allora la macchina torna nello stato di IDLE in attesa di un nuovo dato da elaborare. Si fa notare come da qualsiasi stato è possibile tornare nello stato di RESET all'attivazione dell'omonimo segnale.

In fig. si propone il diagramma a bolle di quanto descritto sopra.

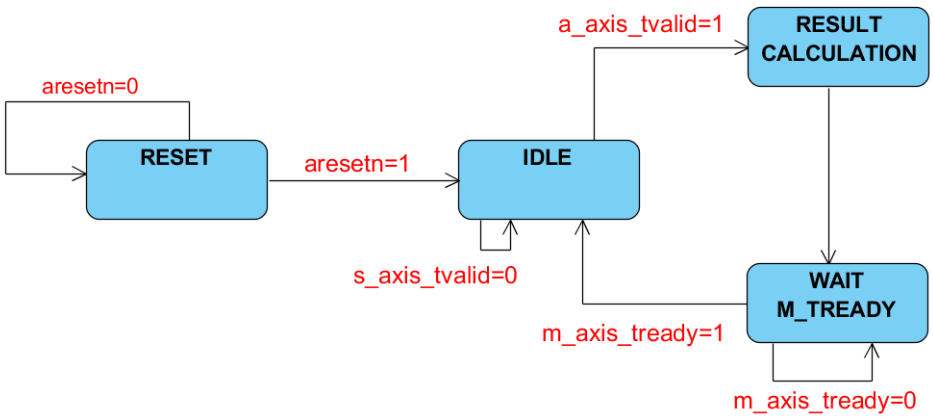

Figura 1.12: Macchina a stati

1.5.2 Divisore

Il componente preposto all'operazione di divisione nella catena di elaborazione si trova nel secondo stadio della pipe ed effettua il rapporto tra il modulo quadro di Sum_Early e Sum_Late. Per una possibile realizzazione del componente sono prese in considerazione 3 possibili soluzioni mostrate di seguito. Da sottolineare come l'IP core Divider generator offre la possibilità tramite il settaggio del parametro *Clocks per division* di ottenere una macchina diversa in termini di area e prestazioni. In via del tutto sperimentale, viene settato tale parametro con 3 valori differenti e ricavati i valori suddetti.

Divider generator	LUT	Slice Register	DSP48	F.max	Cicli di clock
Clocks per division = 1	7168	17165	0	163.514 MHz	91
Clocks per division = 2	7089	9144	0	102.776 MHz	91
Clocks per division = 8	1819	2635	0	104.493 MHz	91

Tabella 1.3: Varianti Divider generator

La scelta tra i tre ricade sulla terza opzione in quanto, al costo di una maggiore latenza, si ottiene un componente ottimizzato nell'occupazione d'area.

Infine, viene confrontato il suddetto IP core con due soluzioni alternative totalmente custom, come mostrato in tab.1.4.

Realizzazione	LUT	Slice Register	DSP48	F.max	Cicli di clock
IP core Divider generator	1819	2635	0	104.493	91
Divisore Non Restoring fixed	5996	428	0	83,198 MHz	104
Descrizione dataflow con operatori /	6850	171	0	2,407 MHz	2

Tabella 1.4: Occupazione d'area divisori

Analizzando la tabella si evince come il Divider generator risulta essere la scelta migliore rispetto alle altre in termini di area e frequenza di lavoro, pur pagandone il prezzo rispetto alle altre nel numero di slice register occupate.

1.5.3 Radice quadrata

Infine, per l'ultimo componente del Task vengono confrontati 3 realizzazioni diverse dell'operatore di radice quadrata. In particolare si confronta l'IP core Xilinx che sfrutta l'algoritmo Cordic con 2 soluzioni che implementano l'algoritmo digit-by-digit per il calcolo del valore di radice.

Realizzazione	LUT	Slice Register	DSP48	F.max	Cicli di clock
Cordic	741	403	0	130.056 MHz	24
Digit-by-digit combinatorio	1917	74	0	13.818 MHz	2
Digit-by-digit sequenziale	202	121	0	120.642 MHz	25

Tabella 1.5: Occupazione d'area radice quadrata

Con un buon compromesso tra area occupata e prestazioni, la scelta migliore ricade sul componente che calcola in maniera in maniera sequenziale il risultato sfruttando l'algoritmo digit-by-digit (per ulteriori informazioni si rimanda a `https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Digit-by-digit_calculation`)

1.5.3.1 Implementazione come FSM dell' Algoritmo digit-by-digit

Come da titolo, il componente che realizza la radice quadrata è implementato secondo la logica di una macchina a stati finiti. Per renderlo compatibile con bus AXI Stream ed altri componenti che vi si interfacciano con esso, vengono aggiunti gli stessi segnali di input/output come descritto in cap.1.5.1.1

Di seguito, in fig.1.13, si propone il diagramma che descrive la FSM.

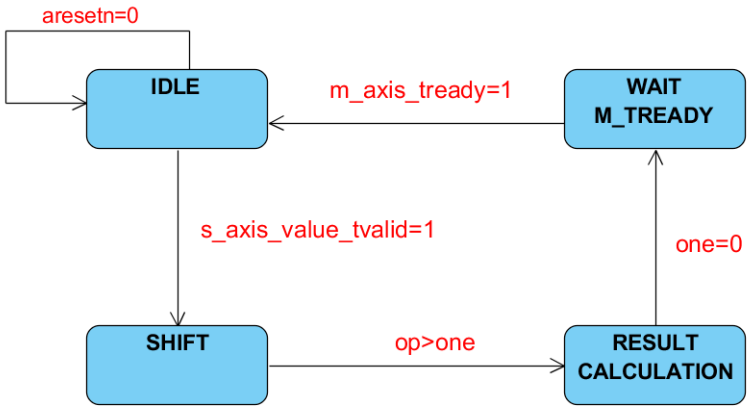

Figura 1.13: Macchina a stati digit-by-digit

A partire dallo stato inziale di **IDLE**, la macchina vi permane fintantoché il segnale di *tvalid* sull'interfaccia slave è pari a 0, oppure viene asserito il segnale di reset. Una volta che il *tvalid* diviene pari a 1, la macchina transita nello stato di **SHIFT** dove viene effettuata la seguente operazione: a partire da un valore che indica la precisione del calcolo, *one*, su di esso viene effettuato uno shift a destra di 2 posizioni fino a quando non si ottiene la più grande potenza di 4 più piccola del radicando. Una volta trovato tale valore, la macchina passa nello stato di **RESULT_CALCULATION** in cui viene effettuato il calcolo della radice secondo la logica dell'algoritmo. Una volta determinata la radice (il segnale *one* è pari a 0), si passa nello stato di **WAIT_M_TREADY** in cui in uscita viene settato ad 1 il valore di *tvalid* e si attende che il componente a valle asserisca il segnale di *tready*.