Progetto di Reti Logiche

Divisore Intero a 32 bit

Andrea Paparella

Introduzione

Descrizione del progetto

Il progetto ha come obiettivo la realizzazione di un divisore intero a 32 bit basato sul metodo di "divisione lunga".

Si definiscono le seguenti variabili:

- N, il dividendo
- D, il divisore
- Q, il quoziente
- R, il resto

Si ipotizza che:

- N e D siano parole da 32 bit, di conseguenza lo saranno anche Q ed R
- D sia stabile sull'ingresso del dispositivo e non cambi per l'intera esecuzione del calcolo
- Tutti i numeri siano rappresentati in codifica binaria naturale, dunque non sono ammessi valori negativi.

Il metodo iterativo della divisione può essere brevemente descritto tramite il seguente pseudocodice:

```
if ( D == 0 ) {
        error();
}
Q = 0
R = 0
for ( n = 31; n >= 0; n-- ) {
        R = R << 1
        R[0] = N[n]
        if( R ≥ D ) {
                R = R − D
                Q[i] = 1
        } else {
                Q[i] = 0
        }
}
```

In accordo con la definizione di divisione, la rete da progettare deve segnalare errore qualora il divisore assuma valore nullo. Se ciò non accade, l'algoritmo prevede n iterazioni, con n pari al numero di bit della parola in ingresso.

Ad ogni passo, è necessario shiftare di una posizione il resto e porre la sua cifra meno significativa al valore della cifra più significativa del dividendo. Successivamente, si procede alla sottrazione tra resto parziale e divisore. Se la sottrazione ha risultato non negativo, si pone l'i-esimo bit del quoziente ad 1, 0 altrimenti.

Approccio risolutivo

Il primo passo da fare è snellire l'algoritmo.

Si osserva che lo statement (R ≥ D) è riscrivibile come R – D ≥ 0. Inoltre, introducendo una variabile T, è possibile scrivere T = R – D ≥ 0. Questo permette di calcolare T prima di valutare l'*if-else* statement e riutilizzare tale valore qualora si entri nel *if* branch.

```
T = R – D
if( T ≥ 0 ) {
        R = T
        Q[i] = 1
} else {
        Q[i] = 0
}
```

Un ulteriore semplificazione viene introdotta ricordando che, se il risultato della sottrazione è negativo allora il bit di prestito in uscita assume valore 1, 0 altrimenti. Pertanto, è possibile sostituire l'*if-else* branch con due statements introducendo:

- Bout, il bit di prestito in uscita dal sottrattore
- ext(Bout), l'estensione a n bit di Bout
- not, operatore di negazione
- x, operatore AND bit a bit
- +, operatore OR bit a bit

L'algoritmo ridotto assume la seguente forma:

```
if ( D == 0 ) {
        error();
}
Q = 0
R = 0
for ( n = 31; n >= 0; n-- ) {
        R = R << 1
        R[0] = N[n]
        T = R – D
        R = not ( ext(Bout) ) x T  +  ext(Bout) x R
        Q[i] = not(Bout)
}
```

Finalmente è possibile dedurre che:

1. Qualunque sia l'esito dell'*if* statement, la sottrazione tra resto parziale e divisore deve essere eseguita;
2. R verrà aggiornato con il risultato della sottrazione o con R stesso;
3. L'i-esima cifra di Q assumerà un valore pari al negato del bit di prestito in uscita dalla operazione di sottrazione.

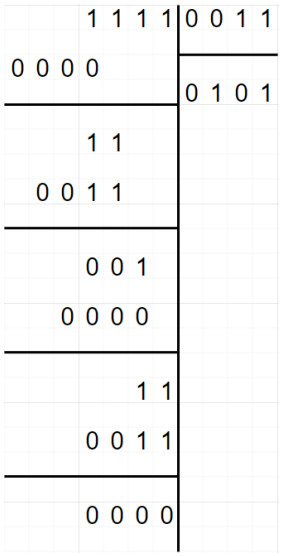
Il secondo passo da fare è osservare una regolarità che emerge nel calcolo del resto parziale. Ciò permetterà di capire come realizzare parte dell'architettura del sistema.

A tal proposito segue un esempio semplificato con parole da 4 bit. Il procedimento è estendibile ad n bit.

Si ipotizza di dover calcolare la divisione tra:

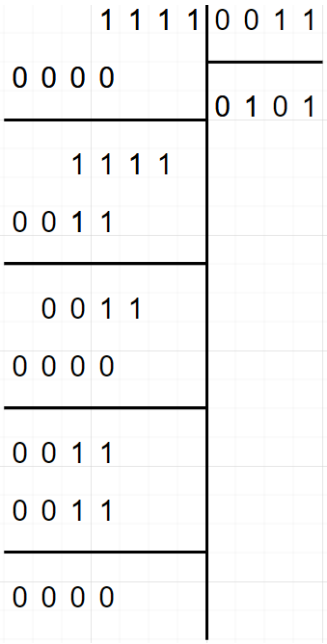
- N = 1111 (=15 in base 10)
- D = 0011 (=3 in base 10)

La classica divisione in colonna si ottiene tramite una serie di sottrazioni tra resto parziale e divisore.



Si nota che, ad ogni iterazione, D viene spostato a destra di una posizione rispetto al dividendo.

Cosa succederebbe se, ad ogni iterazione e prima della sottrazione, invece di spostare il divisore a destra si spostasse il resto parziale a sinistra? Il risultato sarebbe lo stesso e la seguente figura lo dimostra.



Nell'esempio di fianco, alla prima iterazione si esegue il confronto con la cifra più significativa del dividendo: il divisore non "sta" in '1' dunque si sottrae "0000" e si pone uno '0' nella sezione dedicata al quoziente. Si abbassano tutte le cifre del dividendo e si shifta il nuovo resto parziale a sinistra. Alla seconda iterazione si esegue il confronto con le due cifre più significative del resto parziale: il divisore "sta" in "11" dunque si sottrae "0011" e si pone un '1' nella sezione dedicata al quoziente. Ancora una volta, si abbassano le cifre del dividendo e si shifta il nuovo resto parziale a sinistra. E così via.

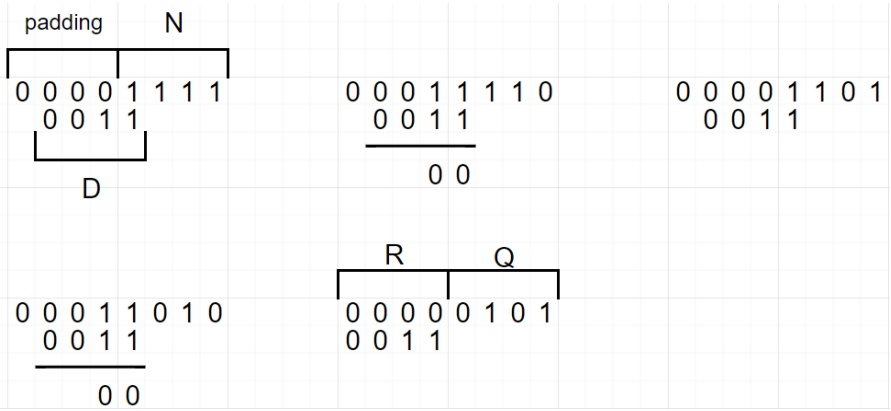
A questo punto si introduce un'ulteriore modifica. Si prova ad effettuare la divisione non in colonna ma "in riga". Questo metodo prevede che ci siano due righe:

- la prima di 2*n = 2*4 bit = 8 bit divisa in due parti di dimensione n: la zona destra contenente il dividendo, la sinistra dei bit '0' di padding;
- la seconda riga rappresenta il divisore.

Per ogni iterazione:

1. Si preleva, dalla prima riga, il resto parziale a partire dal bit (2*n – 2) fino al bit (n – 1);
2. Si confronta resto parziale e divisore;
3. Si sposta N a sinistra;
4. Nel bit meno significativo della prima riga si inserisce il bit del quoziente.
5. Se il bit del quoziente vale '1' si sostituiscono gli n bit più a sinistra della prima riga con il risultato della sottrazione tra resto parziale e divisore.

Alla fine dell'n-esima iterazione, la prima linea conterrà resto e quoziente rispettivamente nella metà sinistra e destra. Segue un esempio esplicativo.



Si conclude che il risultato è analogo a quello ottenuto col metodo della divisione in colonna.

Tale regolarità implica che i registri del resto e del quoziente potranno essere collegati in modo da eseguire con maggior semplicità l'operazione di shifting e assegnamento, ovvero:

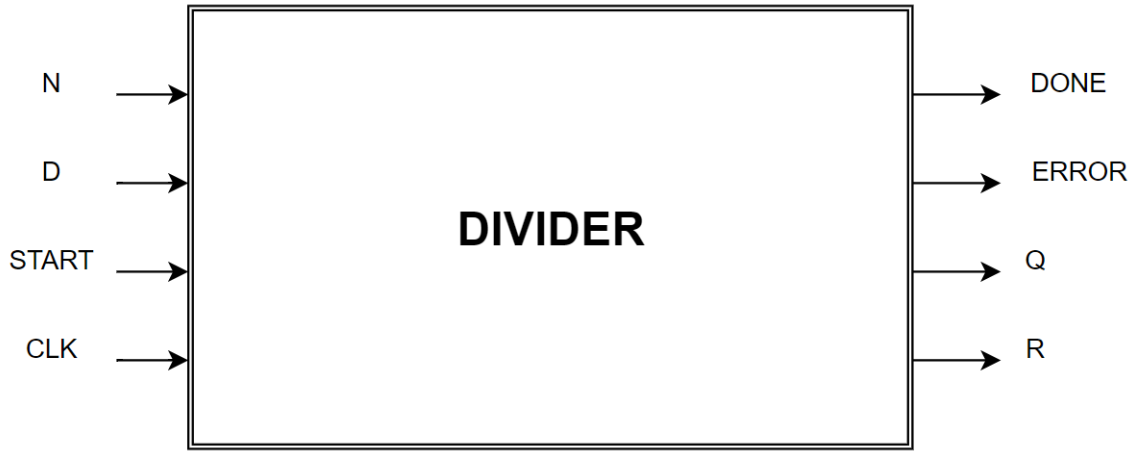
- R = R << 1
- R[0] = N[n]
- Q[i] = not(Bout)

La riduzione dell'algoritmo e la considerazione su R e Q introducono un vantaggio considerevole rispetto ad una diretta traduzione dell'algoritmo in componenti hardware. Non sono necessarie componenti combinatorie per eseguire lo shifting, né per prelevare l'i-esimo bit di una parola. L'architettura è dunque semplificata.

Specifica

Nel seguito viene descritta l'architettura del sistema tramite un approccio top-down..

Interfaccia del sistema



Segnali di ingresso:

- N, dividendo a n bit in codifica binaria naturale
- D, divisore a n bit in codifica binaria naturale
- CLK, segnale che rappresenta il clock del sistema
- START, segnale sincrono per far iniziare il calcolo; per il corretto funzionamento, si assume che esso varrà mantenuto alto per un tempo maggiore uguale al tempo di clock
- RESET, segnale sincrono per portare il sistema in uno stato noto, cioè tutti i segnali in uscita assumo valore '0' e tutti i registri sono riempiti con degli '0'

Segnali di uscita:

- Q, quoziente a n bit in codifica binaria naturale; assume un valore corretto alla fine della n-esima iterazione
- R, resto a n bit in codifica binaria naturale; assume un valore corretto alla fine della n-esima iterazione
- DONE, segnale sincrono che vale '1' quando l'algoritmo termina, ovvero alla fine della n-esima iterazione; vale '0' altrimenti
- ERROR, segnale sincrono che vale '1' quando si pone in ingresso un divisore nullo, '0' altrimenti

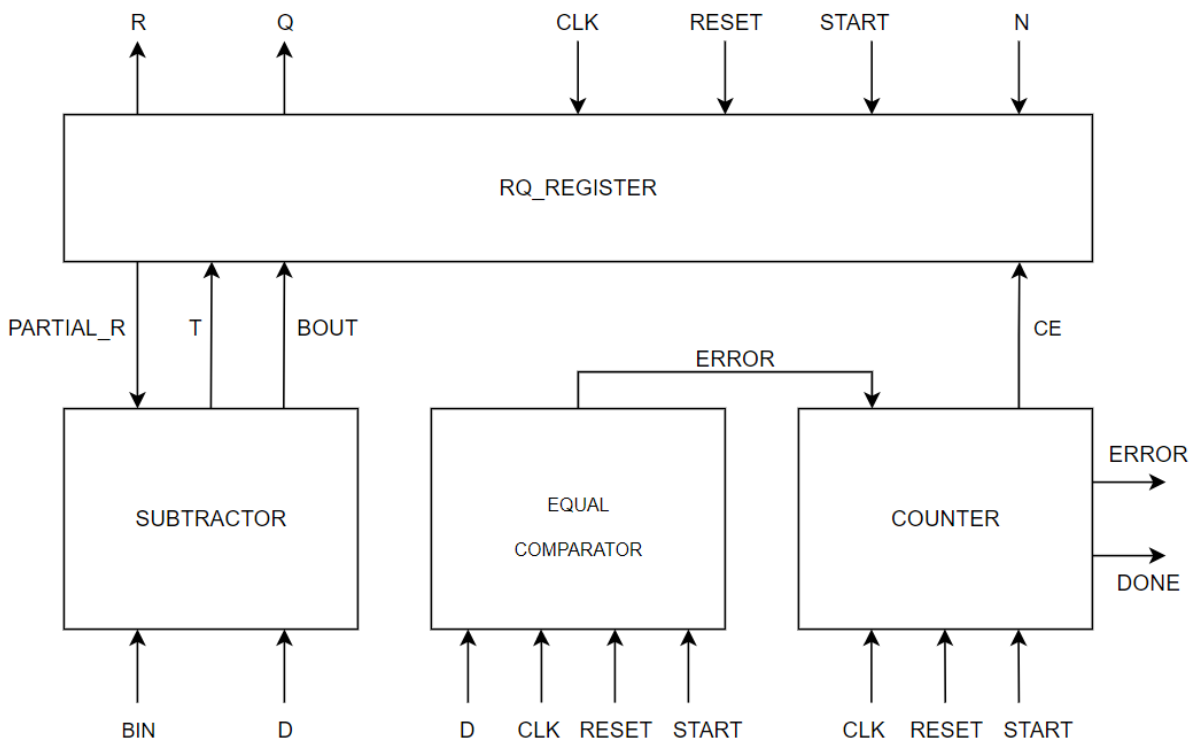
All'inizio del tempo il *Divider* può essere portato in uno stato noto tramite il segnale di RESET.

Per calcolare una divisione è necessario porre in input i valori del dividendo e del divisore, e mantenere alto il segnale di START per un ciclo di clock. Quindi il sistema esegue i calcoli. Finchè non viene raggiunta la n-esima iterazione, le uscite Q ed R assumono un valore diverso dal risultato finale atteso. Alla fine dell'n-esima iterazione, il segnale DONE è portato ad '1' e i risultati di quoziente e resto sono pronti sulle rispettive uscite.

Nel caso in cui si ponga in input un divisore nullo, al fronte di clock successivo al campionamento del segnale di START, ERROR sale ad '1'.

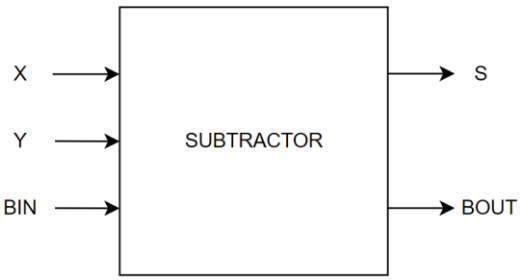
Per eseguire un'ulteriore divisione basta porre in ingresso i nuovi valori di N e D e portare START ad '1'.

Architettura del sistema



Subtractor

Questo modulo esegue la sottrazione binaria tra due numeri in ingresso.



- X, parola a n bit corrispondente al minuendo
- Y, parola a n bit corrispondente al sottraendo
- BIN, bit di prestito in ingresso
- S, parola a n bit pari alla differenza tra X e Y
- BOUT, bit di prestito di uscita

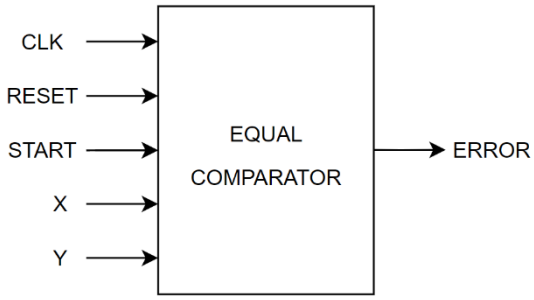
È composto da n sotto-moduli *Full Subtractor* (FS) che realizzano la sottrazione tra tre bits (minuendo, sottraendo e prestito in ingresso).

Nell'ambito del progetto, l'uscita S rappresenta il resto parziale. Secondo lo pseudocodice, se X è il resto parziale e Y il divisore, S corrisponde a T = R – D. Il segnale BIN è posto a '0' dato che si assume che non ci sia prestito in ingresso.

BOUT è il bit che permette di capire se modificare il valore di R con il risultato della differenza.

Equal Comparator

Questo modulo prende in input due numeri e setta ERROR ad ‘1’ se sono uguali, ‘0’ altrimenti.

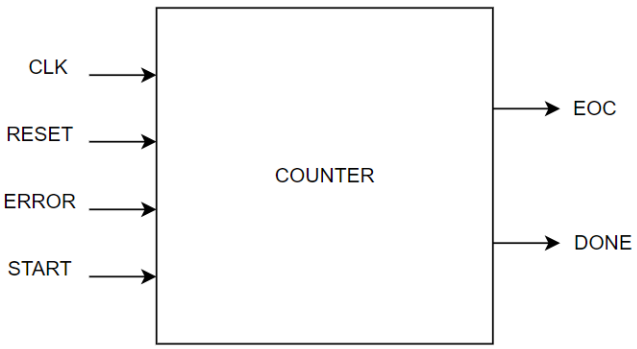


- X, parola a n bit
- Y, parola a n bit
- CLK, segnale di clock
- RESET, segnale sincrono che pone ERROR a ‘0’
- START, segnale sincrono che dà avvio alla comparazione tra X ed Y
- ERROR, segnale sincrono per comunicare l’eventuale uguaglianza delle due parole in ingresso

Nell’ambito del progetto, X è il divisore e Y è il numero zero. Di conseguenza, una volta messi i dati in input e alzato il segnale di START, ERROR vale ‘1’ se il divisore è nullo, ‘0’ altrimenti. La comparazione è eseguita un ciclo di clock dopo che START viene messo ad ‘1’ in modo da consentire la propagazione dell’input nel dispositivo.

Counter

Questo modulo conta le iterazioni dell’algoritmo e segnala la conclusione del calcolo. In riferimento allo pseudo-codice, realizza il *for loop*: for(n = 31; n >= 0; n−−).

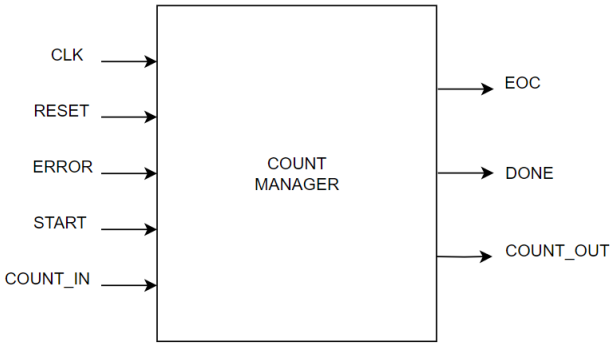


- CLK, segnale di clock
- RESET, segnale sincrono per portare il conteggio a 2^n – 1.
- ERROR, segnale per disabilitare il conteggio in caso di errore
- START, segnale sincrono per iniziare il conteggio
- EOC (End Of Count), segnale sincrono per comunicare la fine del conteggio
- DONE, segnale sincrono per comunicare il termine dell’algoritmo

Counter ha al suo interno due sotto-moduli: un *Subtractor* ed un *Count Manager*.

Il *Subtractor* ha il compito di prendere in input il valore del conteggio e restituire il valore decrementato di un’unità. Si noti che il valore di conteggio non è conosciuto al di fuori del *Counter*, se non quando si è in uno stato noto (per esempio, quando EOC o DONE sono alti, il conteggio vale zero).

Il *Count Manager* fa’ in modo che il conteggio sia compreso tra 2^n – 1 e 0, gestisce situazioni eccezionali e segnala la terminazione del conteggio. Esso è schematizzato di fianco.



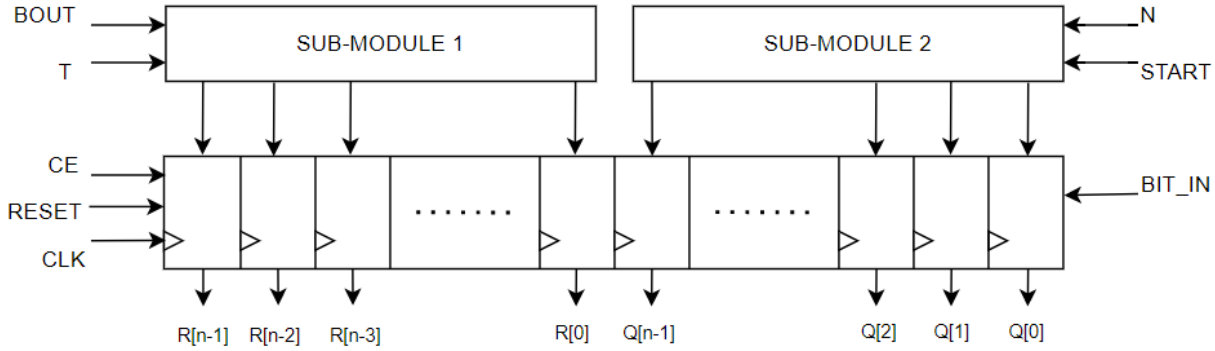
Quando RESET è alto, il contatore assume valore di conteggio nullo e tutti i segnali logici in uscita valgono ‘0’. Il *Count Manager* disabilita il conteggio quando si verifica una condizione di errore, ovvero quando ERROR è alto. È bene sottolineare che abilitare non corrisponde ad iniziare. Ciò che fa iniziare il conteggio è il segnale di START, ciò che lo fa abilitare è l’assenza di errore.

Il *Count Manager* pone il segnale EOC ad ‘1’ quando il conteggio raggiunge il valore zero. Nel caso in esame, la terminazione del conteggio non coincide con la terminazione dell’algoritmo. Infatti, una volta raggiunto il valore zero, è necessario un ulteriore ciclo di clock per campionare valori stabili e corretti di resto e quoziente. Motivo per il quale esiste un segnale di DONE che vale ‘1’ quando i dati in output sono pronti.

RQ Register

L’obiettivo di questo modulo è replicare la divisione “in riga” introdotta nel paragrafo **Approccio risolutivo** dell’**Introduzione**.

Per una maggiore chiarezza espositiva, *RQ Register* viene rappresentato frammentandolo in un blocco di registri e 2 sotto-moduli.



- N, parola a n bit
- T, parola a n bit
- BOUT, segnale sincrono per attivare il funzionamento del *Sub-Module 1*
- START, segnale sincrono per attivare il funzionamento del *Sub-Module 2*
- CLK, segnale di clock
- RESET, segnale sincrono per portare il contenuto dei registri a ‘0’
- CE, segnale sincrono per abilitare la scrittura nei registri
- BIT_IN, bit in ingresso nel registro più a destra del blocco

Il blocco di registri è suddiviso in due zone: la prima a sinistra destinata a contenere R, la seconda a destra Q. In accordo con la divisione “in riga”, N viene memorizzato nella zona più a destra.

I registri sono collegati tra di loro in modo tale da consentire lo scorrimento dei bit. Formalmente, la lettura è parallela, mentre la scrittura è sia seriale che parallela.

Ad ogni ciclo di clock, se CE vale ’1’ avviene lo shifting a sinistra dei registri ed il registro corrispondente a Q[0] prende in input BIT_IN.

Nell’ambito del progetto, CE è alto quando EOC del *Counter* è basso, ciò accade quando il conteggio non è ancora terminato. Mentre BIT_IN costituisce la negazione del prestito in uscita dal *Subtractor*.

La modalità di scrittura nei registri è scelta dai sotto-moduli.

Il *Sub-Module 1* inserisce il dividendo N nei registri di Q quando START vale ‘1’.

Il *Sub-Module 2* inserisce il resto parziale T nei registri di R quando BOUT vale ’0’. Nel caso in esame, ciò si verifica quando $R - D \geq 0$, ovvero il prestito in uscita dal *Subtractor* è ‘0’. Si noti che i registri scorrono indipendentemente dal valore di BOUT. Se quest’ultimo vale ’1’, R[0] assume il valore di Q[n-1]; se altrimenti vale ’0’, Q[n-1] diretto in R[0] viene perso poiché sostituito dal LSB di T.

In conclusione, la particolare struttura dei registri implica che:

1. Q[0] all’iterazione i-esima coincide con Q[i] ad algoritmo terminato;
2. Le operazioni R << 1 e R[0] = N[n] sono direttamente ottenute prelevando n bit dai registri R[n-2], R[n-3], … R[0], Q[n-1]. Tali bit rappresentano il resto parziale da inserire nel *Subtractor*.

Alla fine dell’iterazione n, i registri contengono il risultato della divisone.

Verifica

Per il testing del sistema viene utilizzato un approccio funzionale.

"Program testing can be used to show the presence of bugs, but never to show their absence." (Dijkstra, 1972)

Test-bench

Il test-bench del sistema top-level è ottenuto ponendo in input particolari dati, e.g. divisore nullo, dividendo nullo, dividendo e divisore di dimensioni molto differenti, ed altre ancora.

Casi d'uso

Questa sezione descrive brevemente i casi d'uso del sistema, ovvero le condizioni di utilizzo che il test-bench riproduce per il testing del *Divider*.

All'inizio del tempo, si pone RESET ad '1' per un tempo sufficiente a portare il dispositivo a regime. Successivamente, si portano RESET e START a '0' per un tempo pari al precedente. A questo punto, il sistema è pronto a ricevere gli input ed effettuare la divisione. Per ogni caso d'uso, una volta posti N e D in input, START assume valore '1' per un periodo di clock. Quindi si attendono 32 cicli di clock in modo da dare tempo al *Divider* di calcolare la divisione.

Seguono otto casi d'uso:

1. Caso N>D con N e D numeri piccoli
   N <= "00000000000000000000000000001111"; --15
   D <= "00000000000000000000000000000011"; --3
   Valori attesi: Q=15 R=1
2. Caso N>>D con N molto grande e D piccolo
   N <= "11111111111111111111111111111111"; --4294967295
   D <= "00000000000000000000000000000011"; --3
   Valori attesi: Q=1431655765 R=0
3. Caso N=D
   N <= "00000000000000000000000001001101"; --77
   D <= "00000000000000000000000001001101"; --77
   Valori attesi: Q=1 R=0
4. Caso N<<D con N piccolo e D molto grande
   N <= "00000000000000000000000001001100"; --76
   D <= "00000000000000100000000001001101"; --131149
   Valori attesi: Q=0 R=76
5. Caso N<D con N e D numeri grandi
   N <= "11110000000100000000000001001100"; --4027580492
   D <= "11111111111111111111111111111111"; --4294967295
   Valori attesi: Q=0 R=4027580492
6. Caso N=0 e D>0
   N <= "00000000000000000000000000000000"; --0
   D <= "00000001011000100111100110111101"; --23230909
   Valori attesi: Q=0 R=0
7. Caso N=0 e D=0
   N <= "00000000000000000000000000000000"; --0
   D <= "00000000000000000000000000000000"; --0
   Valore atteso: ERROR=1
8. Caso N>0 e D=0
   N <= "00000000000000000000000000001001"; --9
   D <= "00000000000000000000000000000000"; --0
   Valore atteso: ERROR=1