PROVA FINALE RETI LOGICHE

[Divisore Intero 32 bit]

Tutor

Prof. Carlo Brandolese

Candidato

Giuseppe Vitello

Indice

Sommario

Introduzione

Il progetto consiste nella realizzazione di un modulo che permette di effettuare la divisione intera su 32 bit basandosi sul metodo detto di “divisione lunga”.

Pseudocodice

Sono partito dal seguente pseudocodice per effettuare un’analisi accurata dove D è il Divisore, N il Dividendo, R il resto e Q il Quoziente:

```
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;
}
```

Analisi pseudocodice:

1. Viene fatto in principio un controllo sul Dividendo, se fosse uguale a zero segnala l’errore.
2. Vengono inizializzati Resto e Quoziente a 0.
3. Inizia un ciclo della durata di 32 iterazioni (32 perché stiamo lavorando con Divisore e Dividendo a 32 bit).
4. Ad ogni passo dell’iterazione il resto viene shiftato a sx di un bit ed il suo ultimo bit viene sostituito con il bit i-esimo di N con
$$i = 31 - Numero\ Iterazione\ corrente$$
5. Viene fatta la verifica R ≥ D:
   - se vera al Resto viene sottratto il Dividendo e viene settato ad 1 il bit i-esimo del Quoziente.
   - se falsa il Resto rimane invariato e viene settato a 0 il bit i-esimo del Quoziente.

Al termine delle 32 iterazioni, se il Dividendo è diverso da 0, R e Q avranno i valori corretti di resto e quoziente della divisione intera, ovvero:

$$N = Q * D + R$$

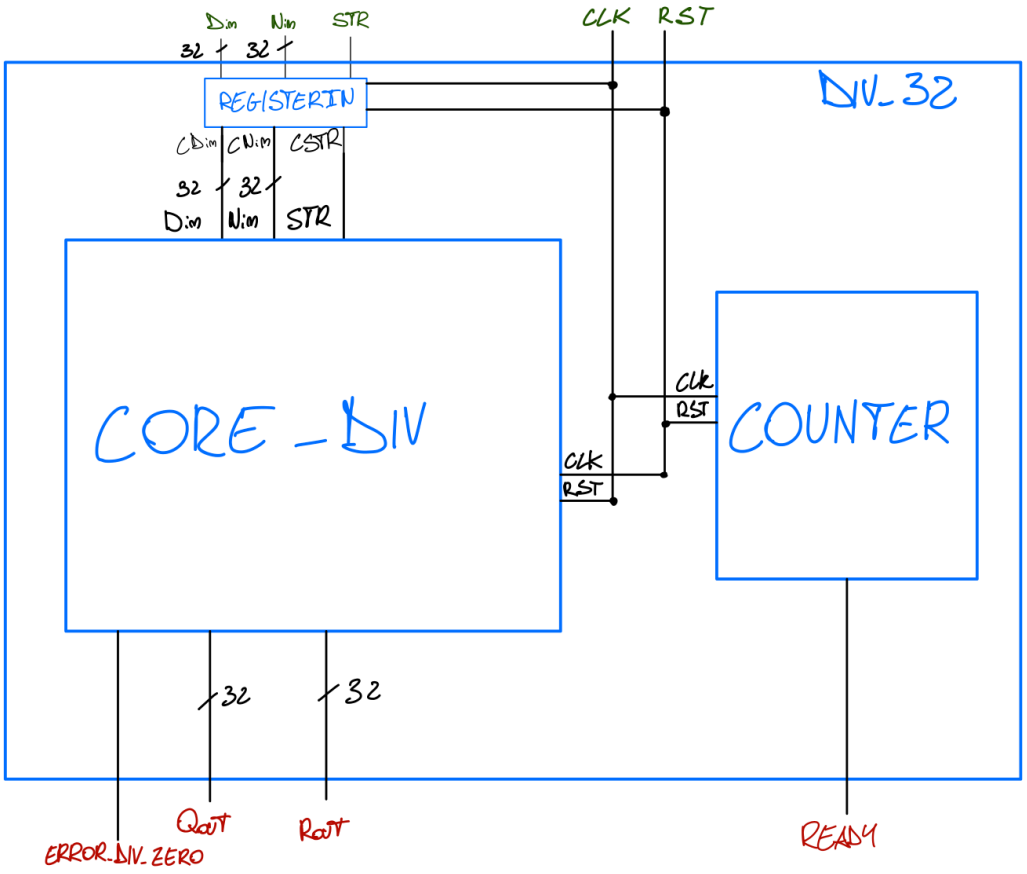
Specifica

In questa sezione è riportata la descrizione completa dell’architettura con approccio top-down.

Per ogni modulo verrà descritto il suo funzionamento, i sotto moduli che lo compongono, ingressi, uscite, segnali interni (di cui viene omessa la codifica poiché tutti i segnali sono in codifica binaria naturale).

Infine viene riportata anche un’immagine per avere un riferimento visivo dei collegamenti.

DIV_32 (TOP MODEL)



Sotto moduli

REGISTERIN: campiona **Nin** e **STR,** nega bit a bit **Din** e campiona il risultato;
CORE_DIV: il nucleo del modello;
COUNTER: blocco contatore porta a 1 il segnale **READY** dopo 32 cicli di **CLK**.

Segnali

Ingressi:

Din: Nuovo valore del Divisore 32 bit;
Nin: Nuovo valore del Dividendo 32 bit;
STR: Segnale di inizio nuova divisione 1 bit;
CLK: clock del sistema 1 bit;
RST: reset del sistema 1 bit.

Uscite:

Qout: Quoziente 32 bit;
Rout: Resto 32 bit;
ERROR_DIV_ZERO: Segnale di errore divisione per zero 1 bit;
READY: Segnala la correttezza dei valori di Qout e Rout 1 bit.

Interni:

CDin: Din' campionato;
CNin, CSTR: valori degli ingressi campionati.

Descrizione

Ipotesi e istruzioni per utilizzatore

Tutti i segnali sono da considerarsi in codifica binario naturale, dunque il modulo permette di calcolare qualunque divisione intera per dividendo e divisore con valori positivi a 32 bit.

In principio deve essere portato il segnale di RST a 1 per inizializzare il circuito, dopo ciò l'utilizzatore potrà inserire i valori **Din** e **Nin** e la divisione inizierà solo quando il segnale **STR** verrà portato ad uno.

Una volta che **STR** viene portato ad uno, il modulo impiegherà 33 cicli di **CLK** per eseguire la divisione, 1 per campionare **Din, Nin e STR** e 32 per la divisione. Solo al 33-esimo le uscite **Qout** e **Rout** saranno gli effettivi quoziente e resto della divisione, prima di ciò avranno risultati intermedi, **CLK=15ns**.

L'utilizzatore non dovrà contare da sé i cicli di **CLK**, bensì potrà utilizzare il segnale d'uscita **READY** che si porta ad 1 quando le uscite sono pronte, dunque potrebbe utilizzare tale segnale come CLK-ENABLE.

Nel caso in cui l'utilizzatore inserisse il **Din=0** dopo due cicli di **CLK** il sistema segnalerà la divisione per zero portando ad 1 il segnale di **ERROR_DIV_ZERO.** In questo caso una volta che il segnale

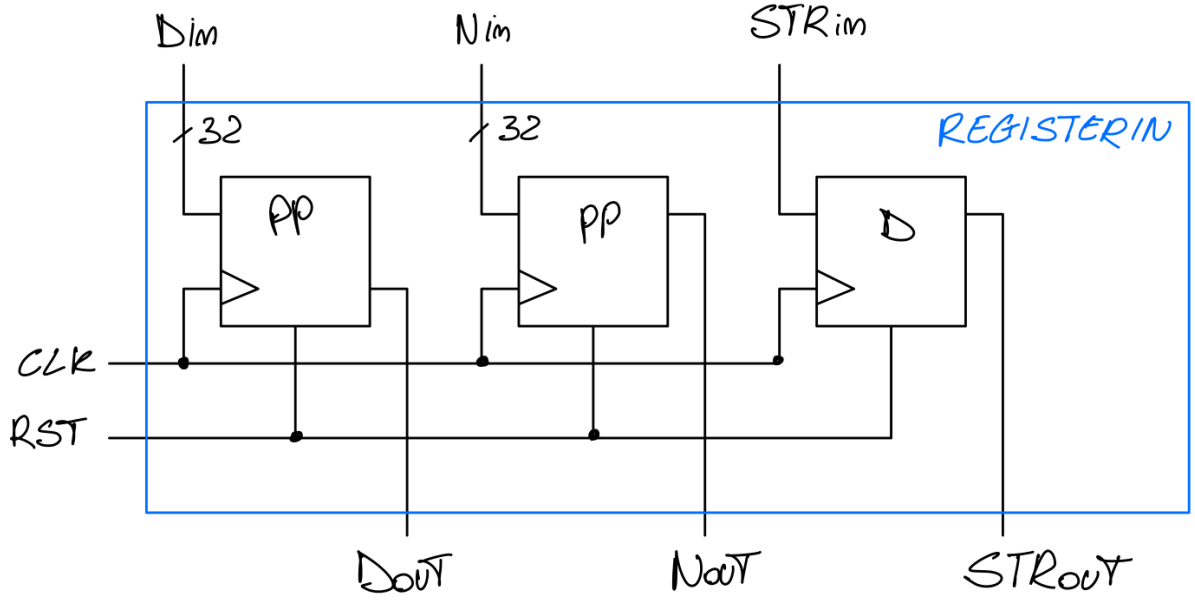
READY sarà pari ad 1, il **Qout** sarà al valore intero massimo a 32 bit ($2^{32} - 1,\ 4.294.967.295$) mentre **Rout** sarà uguale al dividendo **Nin,** mostrati nell'eventualità l'utilizzatore volesse sfruttare anche questa configurazione.

I segnali di ingresso devono essere mantenuti stabili per il tempo necessario al campionamento (circa 7 ns). Il segnale **STR** una volta portato ad 1 deve essere riportato a 0, altrimenti il sistema continuerà a riiniziare una nuova divisione con i valori campionati di **Din** e **Nin.**

Di conseguenza se in qualsiasi momento venisse portato ad 1 il segnale **STR,** verrà persa la divisione corrente e inizierà quella nuova**.**

REGISTEROUT

REGISTERIN



Segnali

Ingressi:

Din: segnale da campionare negato 32 bit;
Nin: segnale da campionare 32 bit;
STR: segnale da campionare 1 bit;
CLK: clock del sistema 1 bit;
RST: reset del sistema 1 bit.

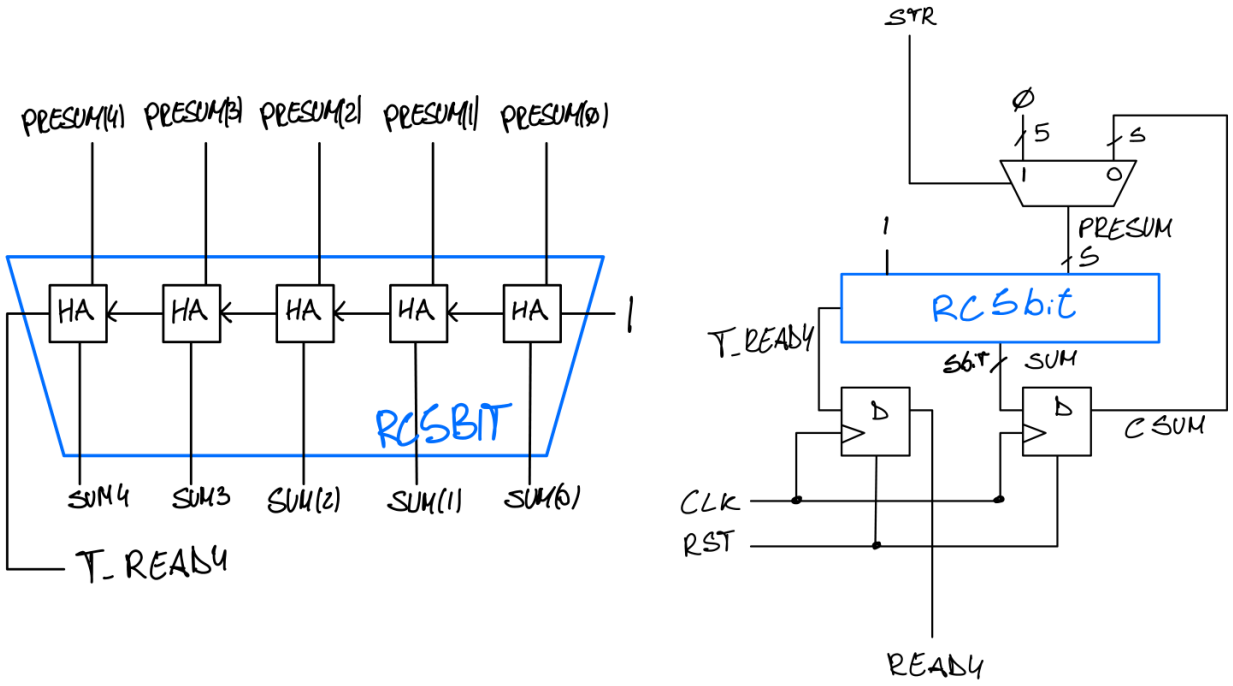
Uscite:

Dout: segnale campionato 32 bit;
Nout: segnale campionato 32 bit;
STRout: segnale campionato 1 bit.

Descrizione

Semplicemente due registri PP a 32 bit e un FF con reset sincrono.

COUNTER



Sotto moduli

RC5BIT: sommatore ripple carry a 5 bit, composto da 5 blocchi HA attaccati in cascata;

Segnali

Ingressi:

CLK: clock del sistema 1 bit;
RST: reset del sistema 1 bit;
STR: segnale da campionare 1 bit.

Uscite:

READY: segnale che indica quando sono trascorsi 32 cicli di CLK 1 bit

Interni:

SUM: valore della somma attuale 5 bit:
CSUM: valore della somma attuale campionato 5 bit;
PRESUM: valore prima dell'incremento 5 bit;
T_READY: segnale READY non ancora campionato 1 bit.

Descrizione

Il modulo alza il segnale di **READY** ad 1 una volta che sono trascorsi 32 cicli di **CLK** da quando è entrato **STR** a 1.

Troviamo immediatamente un MUX avente **STR** come segnale di scelta:

- se **STR=1** setta la sua uscita **PRESUM=0,** cioè inizializza il contatore a 0;
- se **STR=0** setta l'uscita con il valore del contatore precedente, cioè **PRESUM = CSUM,** permettendo cosi di incrementare di nuovo.

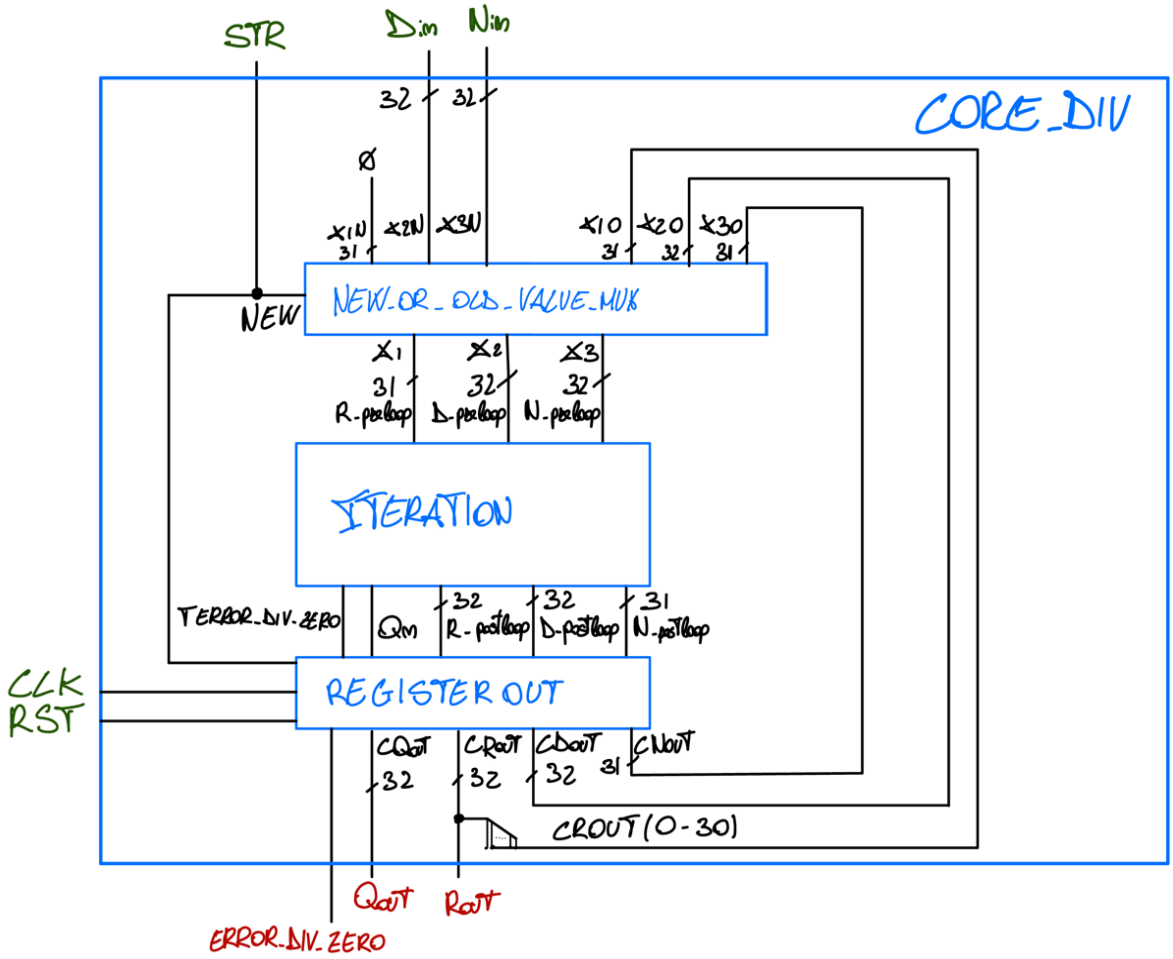
COUNTER

RC5BIT è un sommatore ripple carry a 5 bit, realizzato con 5 half adder, avente come primo operando **PRESUM** e come riporto 1; in questo modo lo si utilizza come contatore.

Per segnalare il 32-esimo **CLK**, il modulo usa il riporto di **RC5BIT**, sarà quest'ultimo il segnale di **T_READY**.

T_READY e il risultato dell'incremento **SUM** vengono entrambi campionati da FF con reset sincrono ottenendo rispettivamente **READY** e **CSUM**.

CORE_DIV



Sotto moduli

NEW_OR_OLD_VALUE_MUX: catena di multiplexer;
ITERATION: traduzione circuitale di un passo dell'algoritmo della lunga divisione;
REGISTEROUT: catena di registri.

Segnali

Ingressi:

Din: Nuovo valore del Divisore negato bit a bit e campionato 32 bit;
Nin: Nuovo valore del Dividendo campionato 32 bit;
STR: Segnale di inizio nuova divisione campionato 1 bit;
CLK: clock del sistema 1 bit;
RST: reset del sistema 1 bit.

Uscite:

Qout: Quoziente della i-esima iterazione 32 campionato bit;
Rout: Resto della i-esima iterazione 32 campionato bit;
ERROR_DIV_ZERO: Segnale di errore divisione per zero campionato 1 bit.

Interni:

Qn: bit i-esimo del Quoziente bit;
R\N\D_preloop: valori entranti nell'iterazione;
R\N\D_postloop: valori uscenti dall'iterazione;
TERROR_DIV_ZERO: Segnale di errore divisione per zero 1 bit;
CQout, CRout, CNout, CDout: Valori campionati uscenti dall'iterazione.

Descrizione

Compiti del modulo

1. Rilevare divisione per zero;
2. Verificare se si deve iniziare una nuova divisione oppure se si deve continuare il calcolo della precedente, nel caso di divisione nuova settare i segnali **Qin** e **Rin** a 0;
3. Shiftare il segnale **Rin** di un bit a sx e concatenarlo con il bit i-esimo di **Nin**, con $i = 31 - Numero\ Iterazione\ corrente$.
4. Verificare se **Rin >= Din**, se vero **Rin = Rin- Din** e il bit i-esimo di **Qout** deve essere settato ad 1, altrimenti **Rin** rimane invariato e il bit i-esimo di **Qout** deve essere settato a 0.
5. Continuare a ripetere i passi 1, 2, 3 e 4 ad ogni ciclo di **CLK.**

Premesse

Una volta fatta l'analisi dei compiti vi sono elencate alcune premesse:

CORE_DIV non si arresta dopo 32 cicli di **CLK,** bensì continua ad eseguire i passi 1,2,3 e 4.

Nonostante i passi evidenzino la necessità dell'operatore shift, non si trova all'interno del modulo nessuna parte specifica che permetta ciò. Il motivo sta nel fatto che in tale sistema, ogni volta che si ha la necessità di eseguire uno shift, rimarrebbe un bit del segnale shiftato inutile. Senza tale bit si ha la possibilità di avere moduli più piccoli (per quanto un bit possa fare la differenza).

Dunque ogni qual volta il sistema deve effettuare uno shift l'operazione viene sostituita da segnali interni che contengono i 31 bit interessati (sono sempre i primi 31 bit). Quando tale segnale dovrà essere utilizzato, verrà concatenato con un bit con il valore corretto per l'operazione da eseguire e nella posizione corretta (è sempre l'ultima).

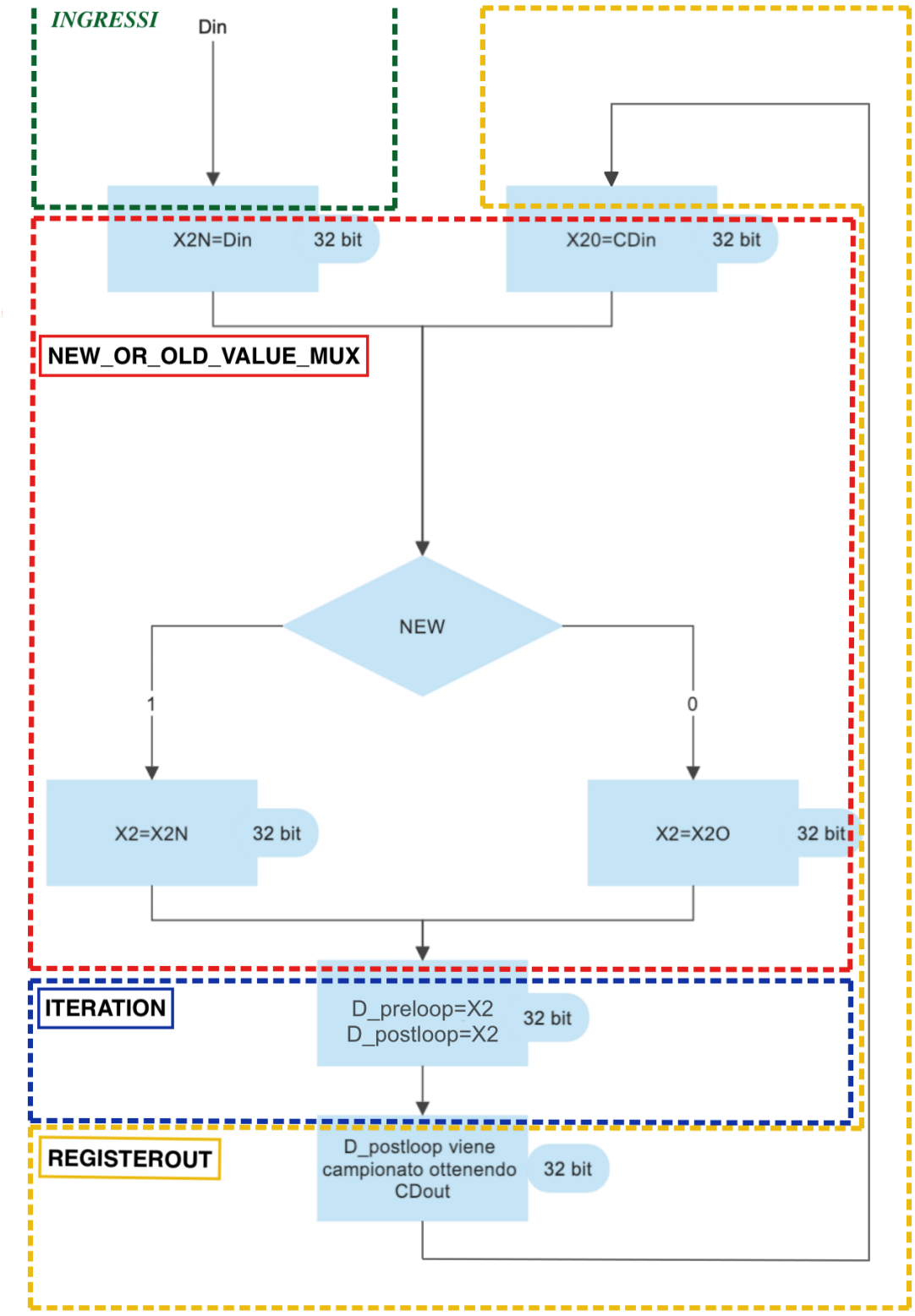
Per tali motivi ritengo che la spiegazione di come vengano effettuate le operazioni di shift conviene affrontarla in questo modulo.

Funzionamento

Per comprendere a pieno cosa accade all'interno del modulo, la descrizione del funzionamento avverrà attraverso dei flowchart e/o spiegazioni testuali per ogni segnale (D Divisore, N Dividendo, Q Quoziente, R Resto e ERROR_DIV_ZERO)

Segnale D Divisore

Ricordo che il segnale **Din** è il Divisore negato bit a bit.

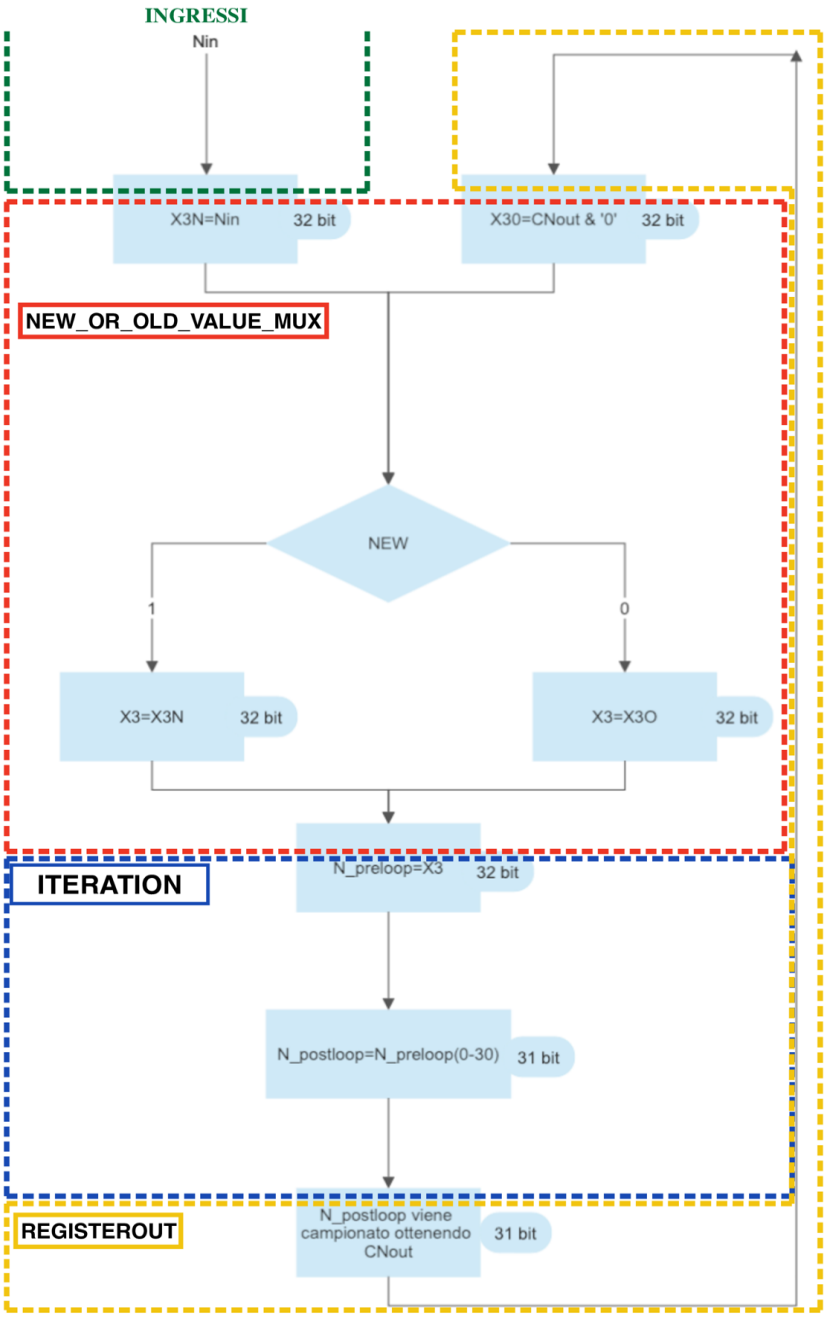


Segnale N Dividendo

Il modulo **ITERATION** necessita di prendere ad ogni iterazione il bit i-esimo del Dividendo per concatenarlo al Resto con

$$i = 31 - Numero\ Iterazione\ corrente$$

Se si shifta di una posizione il Dividendo ad ogni iterazione il bit corretto sarà sempre l'ultimo di **N_preloop**.
Notiamo come viene effettuato lo shift del Dividendo grazie a tutti e 3 i moduli.
Nella prima iterazione **NEW_OR_OLD_VALUE_MUX** fa entrare nel modulo **ITERATION** il Dividendo originale (ricordo che il segnale di **NEW** è interno al sotto modulo e corrisponde a **STR**).
ITERATION fa uscire i primi 31 bit del segnale N che vi è entrato e **REGISTEROUT** li campiona ottenendo **CNout.**
Nelle seguenti iterazioni **CNout** rientra nel modulo **NEW_OR_OLD_VALUE_MUX,** di fatti troviamo **X3O= CNout & 0.**
X3, così facendo, corrisponde al Dividendo shiftato di "NUMERO ITERAZIONI - 1" posizioni.



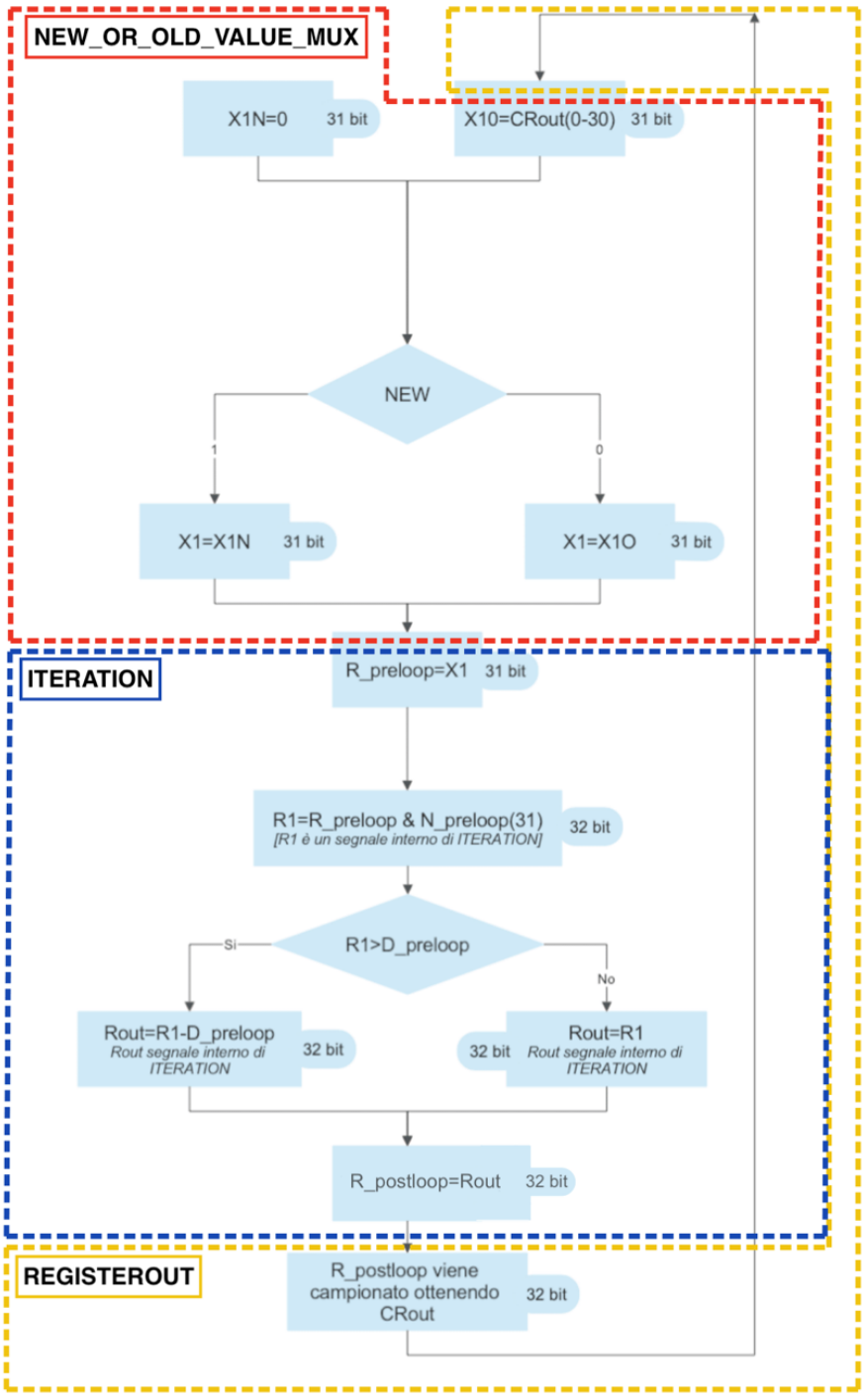
Segnale R Resto

Nella prima iterazione il modulo **NEW_OR_OLD_VALUE_MUX** porta in ingresso a **ITERATION** il Resto inizializzato a 0.

Tale segnale è a 31 bit, dato che **ITERATION** lo concatena con il bit in posizione 31 - #Iterazioni del Dividendo ad ogni iterazione. Come spiegato nel paragrafo precedente tale bit sarà sempre l'ultimo di **N_preloop.**

Una volta pronto, **R_postloop** viene campionato ottenendo **CRout.**

I primi 31 bit di **CRout** vengono riportati al modulo **NEW_OR_OLD_VALUE_MUX** e ripeterà il ciclo.



CORE_DIV

Segnale Q Quoziente

Il segnale Q viene calcolato grazie ai moduli **ITERATION** e **REGISTEROUT**.

ITERATION ha come uscita ad ogni iterazione **Qn,** ovvero il bit i-esimo del Quoziente.

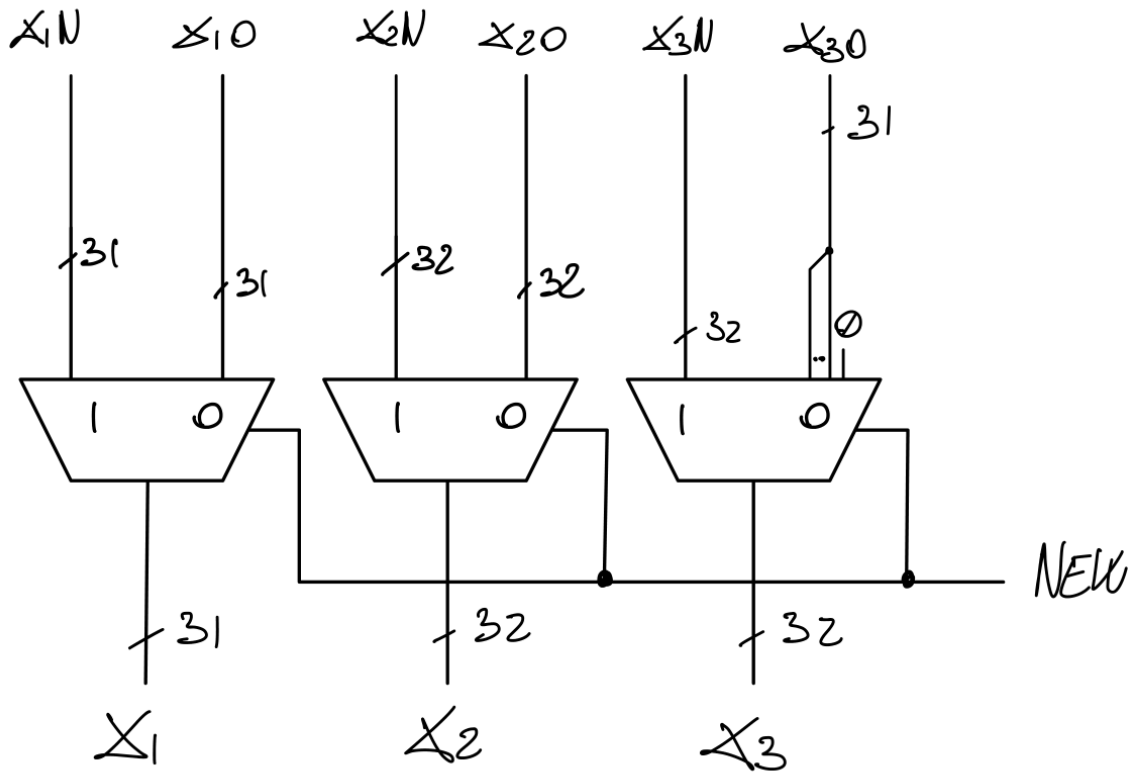
Tale segnale viene campionato da **REGISTEROUT** attraverso un registro SP che permette di inserire **Qn** nella posizione corretta.

Segnale ERROR_DIV_ZERO

ITERATION ad ogni iterazione controlla il segnale D_preloop: se tale segnale ha tutti i bit a 1 il modulo porta a 1 l'uscita **TERROR_DIV_ZERO**, 0 altrimenti.

Tale segnale verrà campionato dal modulo **REGISTEROUT** ottenendo **ERROR_DIV_ZERO** uscita di **CORE_DIV.**

NEW_OR_OLD_VALUE_MUX



Segnali

Ingressi:

> **NEW:** segnale di scelta 1 bit;
> **X1N, X2N, X3NEW, X4NEW:** prima scelta 32 bit;
> **X1O, X2OD, X3O, X4O:** seconda scelta 32 bit.

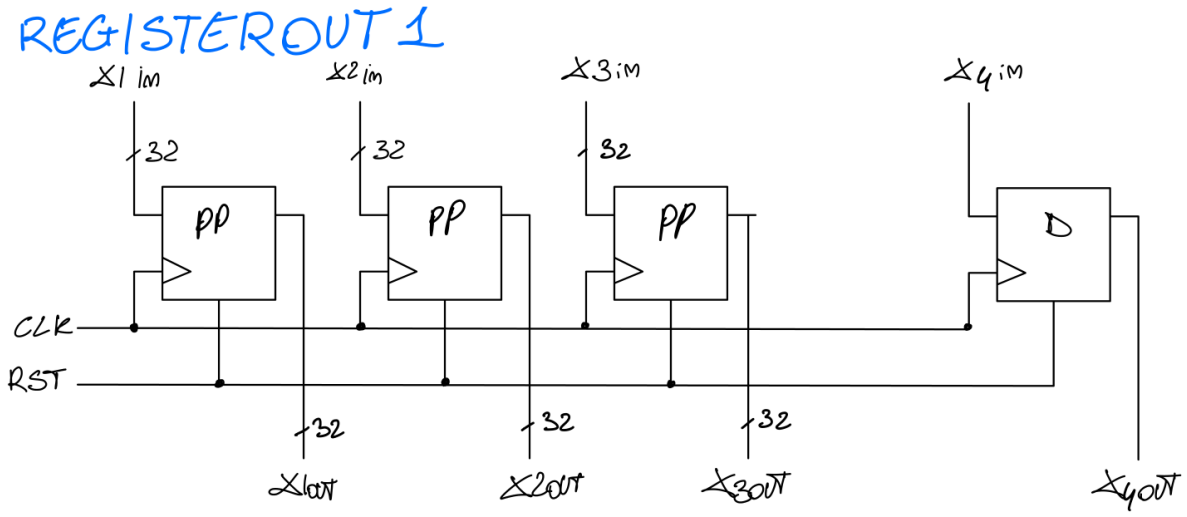
Uscite:

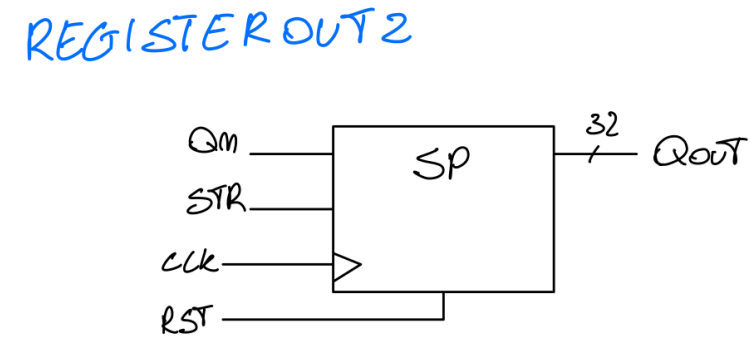
> **X1, X2, X3, X4:** 32 bit.

Descrizione

Semplice catena di mux.

REGISTEROUT





Sotto moduli

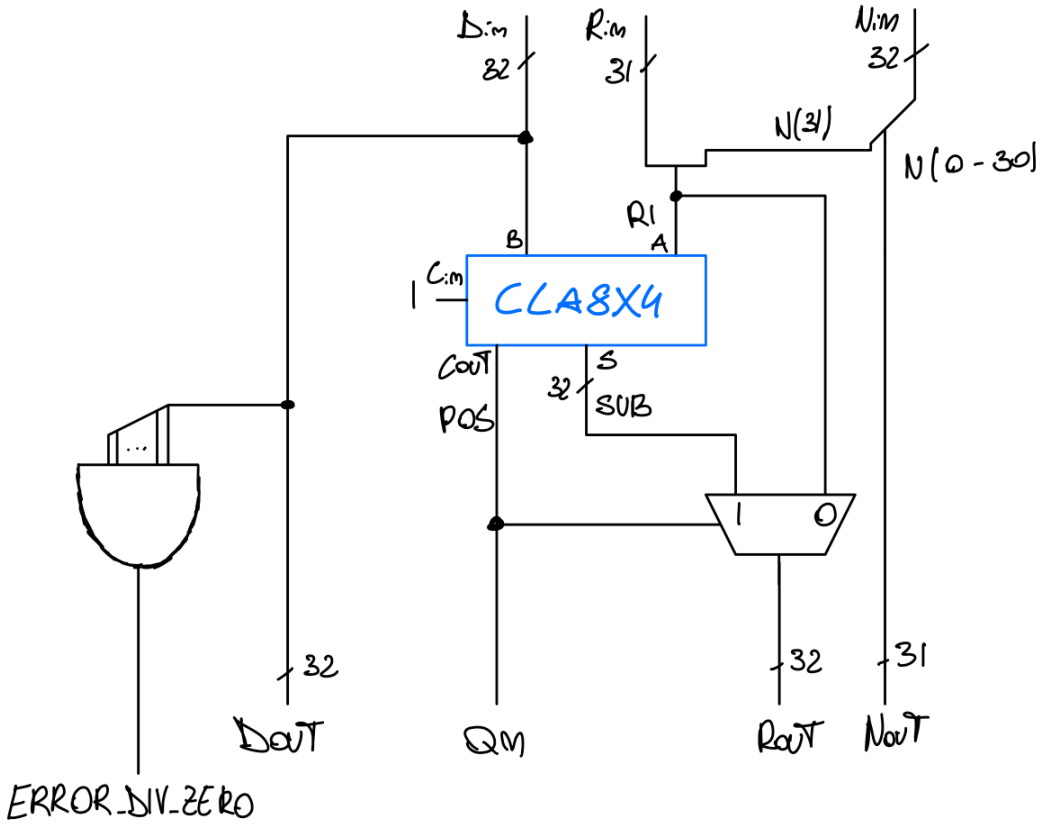
REGISTEROUT1: catena di registri PP;
REGISTEROUT2: registro SP.

Descrizione

REGISTEROUT1: è composto da 3 registri a 32 bit PP, e un FF con reset sincrono.

REGISTEROUT2: se **STR=1** shifta di un bit a sx **Qout** e gli concatena **Qn** nella prima posizione.
In questo semplice modo si inserisce nella posizione corretta il valore di **Qn:** se **STR=0** l'uscita **Qout** diventa 31 bit a 0 seguiti da **Qn** (in realtà in 31 bit a 0 potrebbero avere qualsiasi valore perché non vengono utilizzati).

ITERATION



Sotto moduli

CLA8X4: sommatore curry look ahead a 32 bit;

Segnali

Ingressi:

Din: Divisore negato 32 bit;
Nin: Dividendo shiftato di i-1((i-1)-esima iterazione)32 bit;
Rin: Primi 31 bit del resto della (i-1) -esima iterazione 31 bit;
Qin: Primi 31 bit del quoziente della (i-1) -esima iterazione 31 bit.

Uscite:

Dout: Divisore negato 32 bit;
Nout: Dividendo shiftato per la prossima iterazione 32 bit;
Rout: Resto della i-esima iterazione 31 bit;
Qn: bit i-esimo del Quoziente 31 bit;
ERRORE_DIV_ZERO: segnala che si sta dividendo per zero 1 bit.

Interni:

R1: Resto temporaneo 32 bit;
SUB: risultato sottrazione 32 bit:
POS: riporto della sottrazione 32 bit.

Descrizione

Il modulo **ITERATION** è la traduzione circuitale di un passo dell'algoritmo. In tale descrizione ometterò la spiegazione fatta dello shift dei segnali perché già approfondita nel dettaglio nel modulo **CORE_DIV**.

Prime due istruzioni dell'algoritmo

Iniziamo nell'ottenere **R1,** partendo da **Rin** e concatenandogli l'ultimo bit di **Nin**. La restante parte di **Nin** diventa **Nout** (approfondito nel **CORE_DIV**)**.**

Traduzione circuitale del blocco if dell'algoritmo

Il blocco if nel modulo **ITERATION** diventa un sommatore a 32 bit (**CLA8X4**) e un MUX a 32 bit. Utilizzeremo il sommatore come sottrattore.

Sfruttando le proprietà della somma e del complemento a 1 il riporto della somma segnalerà la condizione **R1>Din.**

Entrano nel modulo **CLA8X4** i segnali **R1, Din** e 1 come riporto. In questo modo, ricordandoci che il segnale **Din** è negato e sommandogli 1, otteniamo il complemento a 1 del Divisore; perciò, viene fatta la sottrazione tra il resto e il divisore, il risultato è **SUB**.

Non stiamo aggiungendo il bit del segno, dunque il riporto del modulo (segnale **POS**) ci permetterà di capire quale dei due operandi fosse più grande.

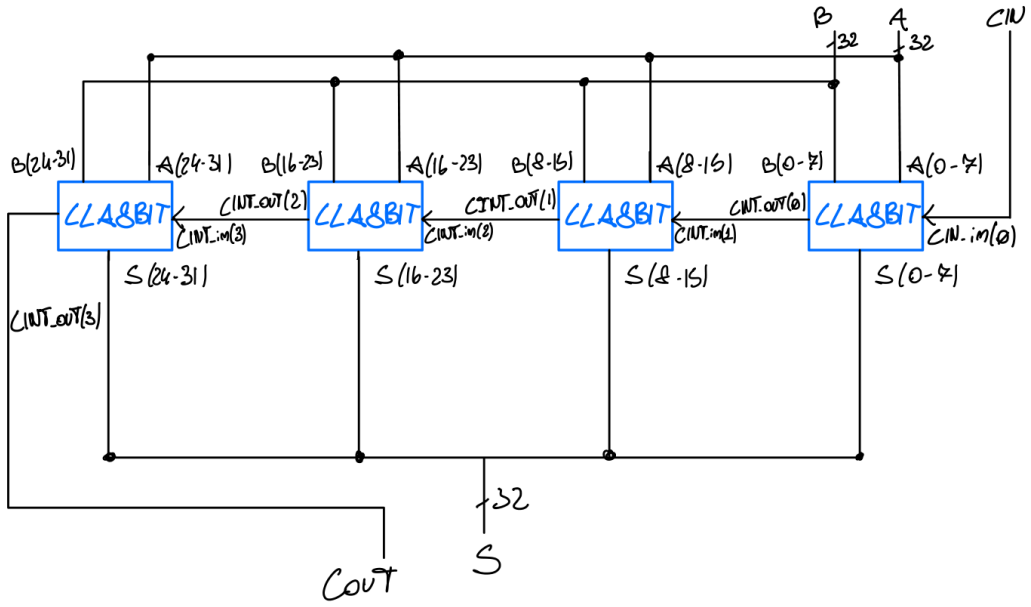
POS permette di ricavare **Rout**, se 1 **Rout=SUB,** se 0 **Rout=R1.**

POS è inoltre l'i-esimo bit da aggiungere al quoziente, dunque l'uscita **Qn.**

Controllo divisione per 0

Infine rimane **ERRORE_DIV_ZERO,** che per segnalare una divisione per zero, basta fare un AND a 32 bit del segnale **Din** (serve un AND perché il segnale è negato, dunque gli 0 sono diventati 1 e viceversa)**.**

CLA8X4



Sotto moduli

CLA8BIT: sommatore curry look ahead a 8 bit;

Segnali

Ingressi:

A: primo operando 32 bit;
B: secondo operando 32 bit;
Cin: riporto iniziale 1 bit.

Uscite:

S: Risultato 32 bit;
Cout: Riporto finale 32 bit.

Interni:

CINT_in: Riporti entranti nei sotto moduli 4 bit;
CINT_out: Riporti uscenti nei sotto moduli 4 bit.

Descrizione

Carry look ahead a 32 bit composto da 4 carry look ahead a 8 bit in cascata connessi correttamente grazie ai segnali interni **CINT_in** e **CINT_out.**

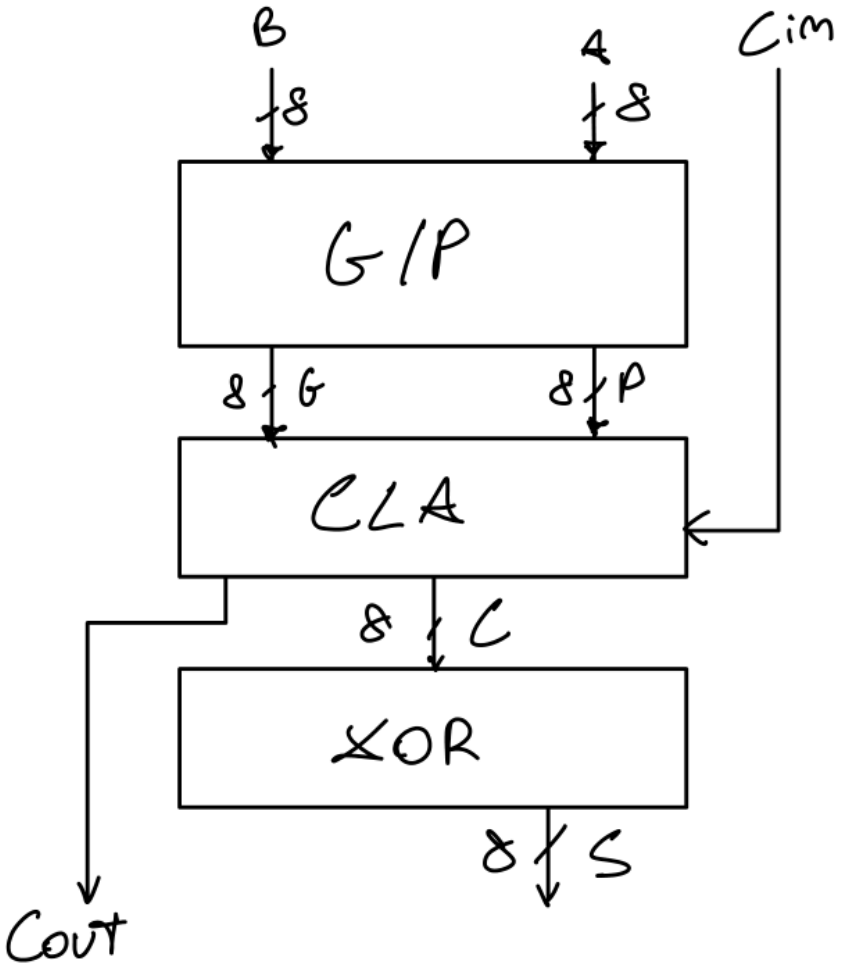
Si potevano scegliere un’infinità di sommatori.

Ho deciso di usare 4 carry look ahead perché avevano un buon rapporto area/tempo A=272(criterio porte generiche) T=11.

Alternative visionate:

RCA a 32 bit: A=192 T=64;
8 CLA a 4 bit: A=208 T=19;
2 CLA a 16 bit: A=400 T=7;
CLA a 32 bit: A=656 T=5;

CLA8BIT



Sotto moduli

G/P: blocco generazione/propagazione;
CLA: blocco che calcola i riporti;
XOR: catena di XOR per il calcolo dei risultati.

Segnali

Ingressi:

A: primo operando 8 bit;
B: secondo operando 8 bit;
Cin: riporto iniziale 1 bit.

Uscite:

S: Risultato 8 bit;
Cout: Riporto finale 32 bit.

Interni:

C: Riporti 8 bit;
P: Propagazioni 8 bit;
G: Generazioni 8 bit.

Test-bench

I test prevedono una partizione del dominio di INPUT e vi sono almeno 2 test per ogni partizione.

La scelta dei valori di input è stata dettata dal buon senso, dunque sono dei valori che dovrebbero maggiormente sforzare il sistema.

Oltre ciò vi è il testing del segnale STR, che se portato ad 1 in qualsiasi momento, deve arrestare la divisone corrente e iniziare quella nuova.

Test

Viene mostrato l'input **Din** e **Nin** e i segnali **Qout** e **Rout** e **ERROR_DIV_ZER0** dopo i 33 cicli di **CLK.**

D=0

Din <= "00000000000000000000000000000000"; --0
Nin <= "00000000000000000000111110101010"; --4010
Qout=--(4.294.967.295) 2^32-1
Rout = 4010
ERROR_DIV_ZER0 = 1

D=0

Din <= (others => '0'); --0
Nin <= (others => '1'); --(4.294.967.295) 2^32-1
Qout = (4.294.967.295) 2^32-1
Rout = (4.294.967.295) 2^32-1
ERROR_DIV_ZER0 = 1

D<N

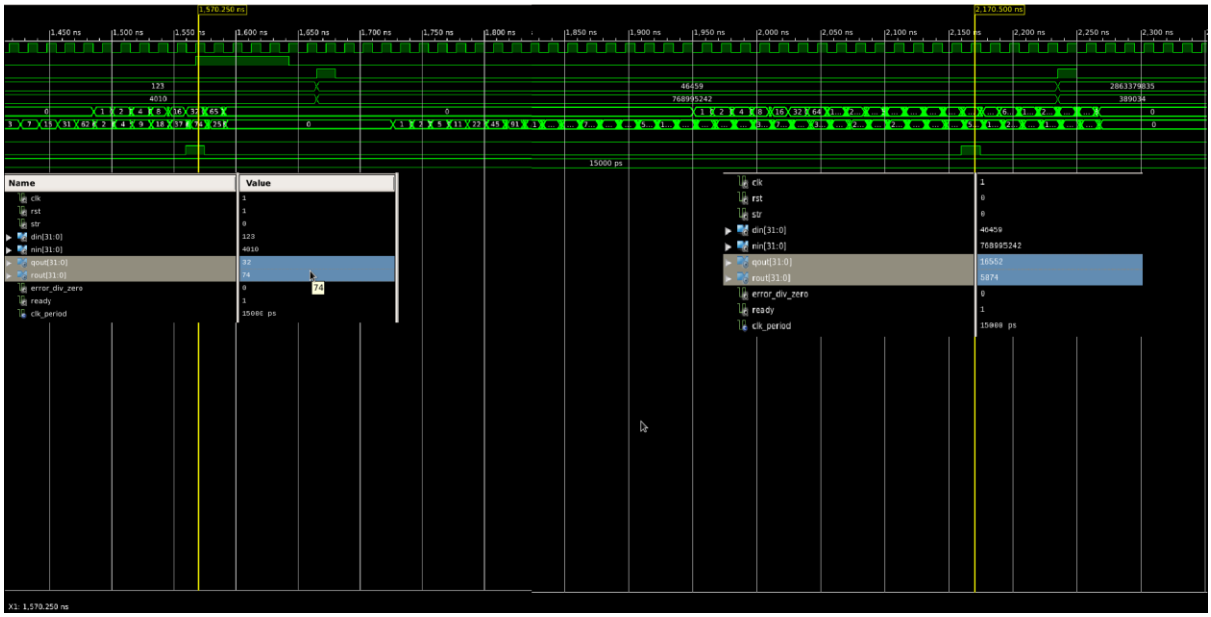
Din <= "00000000000000000000000001111011";--123
Nin <= "00000000000000000000111110101010";--4010
Qout = 32
Rout = 74
ERROR_DIV_ZER0 = 0

D<N

Din <= "00000000000000001011010101111011";--46459
Nin <= "00101101110101011110111110101010";--768995242
Qout = 16.552
Rout = 5.874
ERROR_DIV_ZER0= 0

Test reset

Tra le due divisioni precedenti viene alzato per la seconda volta il segnale di reset, per verificarne il corretto funzionamento.



D>N e test STR

Tele test porta il segnale **STR** a 1 dopo 37 cicli di **CLK** rispetto al precedente.

Din <= "10101010101010111011010101111011";--2863379835
Nin <= "00000000000001011110111110101010";--389034
Qout = 0
Rout = 389.034
ERROR_DIV_ZER0= 0

D<N

Din <= "00000000000000000000000000000001";--1
Nin <= "11111111111111111111111111111111”;--(4.294.967.295) 2^32-1
Qout = (4.294.967.295) 2^32-1
Rout = 0
ERROR_DIV_ZER0 =0

D=N

Din <= "11111111111111111111111111111111";--(4.294.967.295) 2^32-1
Nin <= "11111111111111111111111111111111";--(4.294.967.295) 2^32-1
Qout = 1
Rout = 0
ERROR_DIV_ZER0= 0

D=N

Din <= "10111010101011111011110101111011";--3132079483
Nin <= "10111010101011111011110101111011";--3132079483
Qout = 1
Rout = 0
ERROR_DIV_ZER0= 0

D>N

Din <= "00000000000000000000000000000011";--2
Nin <= "00000000000000000000000000000001";--1
Qout = 0
Rout = 1
ERROR_DIV_ZER0= 0

D<N e test STR

Porto il segnale di **STR** a 1 e entra ***input 1*** dopo 7 cicli di **CLK** riporto il **STR** a 1 e entra ***input 2.***

input 1

I punti interrogativi segnalano che non si è conclusa la divisione non essendo passati i 33 cicli di **CLK** richiesti.

Din <= "00000000000000000000000000000001";--1
Nin <= "10111010101011111011110101111011";--3132079483
Qout =??
Rout =??
ERROR_DIV_ZER0 =??

input 2

Din <= "00000000000000000000000000000001";--1
Nin <= "10111010101011111011110101111011";--3132079483
Qout = 3132079483
Rout= 0
ERROR_DIV_ZER0=0

Considerazioni finali

Alternative da citare

All'interno del blocco **COUNTER** si sarebbe potuto nuovamente usare **CLA8bit,** con delle opportune modifiche, per avere riutilizzo di moduli.

Complicando un po' l'architettura sarebbe stato possibile evitare il ricircolo del Divisore negato e del Dividendo shiftato ogni volta. Oppure senza complicare, ma calcolando la divisione con un ciclo di **CLK** in più, dato dalla sostituzione di **NEW_OR_OLD_VALUE_MUX** con dei registri.

Senza entrare troppo nel dettaglio, ma illustrando solo la logica di entrambe le alternative:

Per il Divisore negato basterebbe spostare il campionamento in **REGISTEROUT** in un nuovo registro PP prima di **NEW_OR_OLD_VALUE_MUX;** si agisce analogamente con il Dividendo utilizzando un registro PS, tale che faccia uscire solo il bit necessario.

Architettura finale