Katia Maschi

VR376761

Progettazione di sistemi embedded

UniversitA’ degli studi di Verona

DIPARTTIENTO DI iNFORMATICA

A.A. 2013/2014

Indice

1. Lezione 2 4

1.1 Interfaccia del modulo 4

1.2 Funzionalità 5

1.2.1 EFSM 5

1.2.2 Implementazione 7

2. Lezione 3 10

2.1 RTL 10

2.1.1 La simulazione di SystemC 10

2.1.2 File VDC 12

2.2 TLM UT 12

2.3 TLM LT 12

2.4 TLM AT4 13

2.5 Comparazione dei tempi 13

3. Lezione 4 14

3.1 Untimed (UT) 14

3.2 Loosley Timed (LT) 14

3.3 APPROXIMATELY TIMED (AT4) 14

3.4 Comparazione dei tempi 15

4. Lezione 5 - Assertion 15

4.1 Proprietà 1 15

4.2 Proprietà 2 16

4.3 Proprietà 3 16

5. Lezione 5 - Transactor 17

5.1 Mapping di porte e segnali tra TLM e RTL 17

5.2 Traduzione da funzioni a operazioni 17

5.3 Comportamento del transattore 18

6. Lezione 6 20

6.1 Codice in C++ 20

6.2 Schedule 20

6.3 Output RTL 21

6.4 Confronto 22

7. Lezione 7 22

7.1 Device driver 22

8. Lezione 10 24

8.1 VHDL 24

8.2 Simulazione 25

9. Lezione 11 27

9.1 Ex\_1 27

9.1.1 Ex\_1 stimuli\_1 27

9.1.2 Ex\_1 stimuli\_2 27

9.1.3 Ex\_1 stimuli\_3 28

9.1.4 Confronti 28

9.2 Ex\_2 30

9.2.1 Ex\_2 stimuli\_1 30

9.2.2 Ex\_2 stimuli\_2 30

9.2.3 Ex\_2 stimuli\_3 31

9.2.4 Confronti 31

9.3 Ex\_3 31

9.3.1 Versione a) 32

9.3.2 Versione b) 32

9.3.3 Versione c) 34

9.3.4 Versione d) 35

9.4 Ex\_4 36

9.4.1 Signal drivers 36

9.5 Ex\_5 37

9.5.1 Signal drivers 37

9.6 Ex\_6 38

10. Lezione 12 39

10.1 Compilazione e sintesi 39

10.2 Area 40

10.3 FSM 41

Modelling al livello RTL

# Lezione 2

Il progetto consiste nella realizzazione a livello RTL un sistema per la divisione. Il modulo verrà testato attraverso il testbench che simulerà l’esecuzione del modulo forzando i segnali con i valori richiesti.

Il sistema è suddiviso in cinque file:

* **main\_divisore\_RTL.cc:** rappresenta l’interfaccia di collegamento tra il divisore\_RTL e il testbench;
* **divisore\_RTL.hh:** rappresenta il file di dichiarazione dell’EFSM che descrive come è costituito il modulo TEA RTL;
* **divisore\_RTL.cc**: rappresenta il file elaborativo dell’EFSM, ovvero il vero e proprio modulo divisore\_RTL;
* **divisore\_RTL\_testbench.hh:** rappresenta la libreria del testbench, descrive come è costituito il modulo di simulazione.
* **divisore\_RTL\_testbench.cc:** rappresenta il modulo di simulazione, contiene al suo interno l’implementazione del testing di sistema con le modalità richieste.

## Interfaccia del modulo

L’interfaccia del mio modulo DIVISORE è costituita da 8 porte:

* **number\_isready:** è un segnale di 1 bit messo a 1 dal testbench quando due nuovi numeri sono disponibili nelle divisor\_port e dividend\_port. Il modulo divisore deve attendere finché number\_isready non è messo a 1 per iniziare il calcolo della divisione intera.
* **divisor\_port:** è un segnale a 16 bit che è settato dal tetstbench. Rappresenta il divisore dato in input su cui il modulo esegue la divisione.
* **dividend\_port:** è un segnale a 16 bit che è settato dal tetstbench. Rappresenta il dividendo dato in input su cui il modulo agisce per eseguire la divisione.
* **quotient\_port:** è un segnale a 16 bit che è settato dal modulo divisore. Rappresenta il quoziente dato in output dopo che il modulo ha eseguito la divisione.
* **reminder\_port:** è un segnale a 16 bit che è settato dal modulo divisore. Rappresenta il resto dato in output dopo che il modulo ha eseguito la divisione.
* **result\_isready:** è un segnale di 1 bit messo a 1 dal modulo divisore quando ha calcolato la divisione e il risultato è disponibile nelle quotient\_port e reminder\_port.
* **reset:** è un segnale di 1 bit usato per resettare il modulo divisore.
* **Clk:** è il clock.

clk

result\_isready

reminder\_port

quotient\_port

reset

dividend\_port

divisor\_port

number\_isready

**DIVISORE**

## Funzionalità

Le funzionalità che devo implementare sono quelle del divisore seguendo il seguente algoritmo:

SET quoziente to 0.

Allinea a sinistra le cifre in dividendo e divisore.

RIPETI

IF la porzione di dividendo sopra il divisore è maggiore o uguale del divisore

THEN sottrai divisore alla porzione di dividendo e concatena 1 alla parte destra del quoziente.

ELSE concatena 0 alla parte destra del quoziente.

Shifta del divisore di un posto a destra.

FINCHE’ il dividendo è minore del divisore.

Il quoziente è corretto, il resto è uguale al divisore.

Queste funzionalità sono facilmente implementate con C++ con una semplice divisione. Al livello RTL le divisione deve essere però calcolata lavorando con i segnali senza supporto di librerie. Questo algoritmo viene quindi calcolato nel seguente modo:

void divisione(unsigned int Divisor, unsigned int Dividend, unsigned int \*quotient, unsigned int \*reminder){

unsigned Divisor\_Shift=Divisor;

if(Dividend < Divisor){

Quotient=0;

}

if(Divisor\_Shift.test(Divisor.length() -1 != true) {

Divisor\_Shift=Divisor\_Shift << 1;

}

Quotient=Quotient << 1;

if(Dividend >= Divisor\_Shift) {

Dividend=Dividend - Divisor\_Shift;

Quotient=Quotient + 1;

}

Divisor\_Shift=Divisor\_Shift >> 1;

Reminder=Dividend;

}

### EFSM

Un modulo può essere rappresentato da un clocked EFSM. Io ho pensato di strutturarlo nel seguente modo:

Dividend < Divisor

Divisor > Dividend

Remider = Dividend;

Divisor\_Shift = Divisor\_Shift >> 1;

if(Dividend >= Divisor\_Shift){

Dividend = Dividend - Divisor\_Shift;

Quotient = Quotient + 1;

}

Quotient = Quotient << 1;

if(Divisor\_Shift.test(Divisor.length()-1) != true)

{

Divisor\_Shift = Divisor\_Shift << 1;

}

if(Dividend < Divisor){

Quotient = 0;

}

Divisor\_Shift = Divisor;

Dividend = dividend\_port;

Divisor = divisor\_port;

Divisor\_Shift = 0;

quotient\_port = 0; reminder\_port = 0;

result\_isready = 0;

Divisor = 0; Dividend = 0; Quotient = 0; Reminder = 0;

quotient\_port = 0; reminder\_port = 0;

result\_isready = 0;

quotient\_port = Quotient;

reminder\_port = Reminder;

result\_isready = 1;

number\_isready=1

number\_isready=0

Divisor != 0

Divisor = 0

Divisor\_Shift.test(Divisor.length()-1)=true

Divisor = Divisor\_Shift

Divisor != Divisor\_Shift

### Implementazione

Il mio modulo è codificato in due SC\_METHODS, uno per calcolare i valori interni delle variabili e uno per calcolare lo stato prossimo.

SC\_METHOD(elaborate\_DIVISION\_FSM);

sensitive << reset.neg();

sensitive << clk.pos();

SC\_METHOD(elaborate\_DIVISION);

sensitive << STATUS << number\_isready << dividend\_port << divisor\_port;

Il primo SC\_METHODS è sensibile alle variazioni di reset e clock. Il secondo invece è sensibile allo stato, le porte divisore e dividendo e al segnale number\_isready.

#### Elaborate\_DIVISION\_FSM

void divisore\_RTL :: elaborate\_DIVISION\_FSM(void){

if (reset.read() == 0){

cout<< "\t" <<sc\_time\_stamp()<< " - divisore: reset" << endl;

STATUS = Reset\_ST;

}

else if (clk.read() == 1) {

STATUS = NEXT\_STATUS;

switch(STATUS){

case Reset\_ST:

quotient\_port.write(0);

reminder\_port.write(0);

result\_isready.write(0);

break;

case ST\_0:

quotient\_port.write(0);

reminder\_port.write(0);

result\_isready.write(0);

Divisor.write(0);

Dividend.write(0);

Quotient.write(0);

Reminder.write(0);

break;

case ST\_1:

Dividend.write(dividend\_port.read());

Divisor.write(divisor\_port.read());

Divisor\_Shift.write(0);

break;

case ST\_2:

Divisor\_Shift.write(Divisor.read());

break;

case ST\_3:

if(Dividend.read() < Divisor.read()){

Quotient.write(0);

}

break;

case ST\_4:

if(Divisor\_Shift.read().test(Divisor.read().length()-1) != true) {

Divisor\_Shift.write(Divisor\_Shift.read() << 1);

}

break;

case ST\_5:

Quotient.write(Quotient.read() << 1);

break;

case ST\_6:

if(Dividend.read() >= Divisor\_Shift.read()) {

Dividend.write(Dividend.read() - Divisor\_Shift.read());

Quotient.write(Quotient.read() + 1);

}

break;

case ST\_7:

Divisor\_Shift.write(Divisor\_Shift.read() >> 1);

break;

case ST\_8:

Reminder.write(Dividend.read());

break;

case ST\_9:

quotient\_port.write(Quotient.read());

reminder\_port.write(Reminder.read());

result\_isready.write(1);

break;

}

}

}

#### Elaborate\_DIVISION

void divisore\_RTL :: elaborate\_DIVISION(void){

cout<<"\t"<<sc\_time\_stamp()<<" - divisore: DIVISION"<<endl;

NEXT\_STATUS = STATUS;

switch(STATUS){

case Reset\_ST:

NEXT\_STATUS = ST\_0;

break;

case ST\_0:

if (number\_isready.read() == 1){

NEXT\_STATUS = ST\_1;

} else {

NEXT\_STATUS = ST\_0;

}

break;

case ST\_1:

NEXT\_STATUS = ST\_2;

break;

case ST\_2:

if(Divisor.read() != 0) {

NEXT\_STATUS = ST\_3;

} else {

NEXT\_STATUS = ST\_9;

}

break;

case ST\_3:

if(Dividend.read() < Divisor.read()){

NEXT\_STATUS = ST\_8;

} else {

NEXT\_STATUS = ST\_4;

}

break;

case ST\_4:

if(Divisor\_Shift.read().test(Divisor.read().length()-1) != true) {

NEXT\_STATUS = ST\_3;

} else {

NEXT\_STATUS = ST\_5;

}

break;

case ST\_5:

NEXT\_STATUS = ST\_6;

break;

case ST\_6:

NEXT\_STATUS = ST\_7;

break;

case ST\_7:

if (Divisor.read() == Divisor\_Shift.read()){

NEXT\_STATUS = ST\_8;

} else {

NEXT\_STATUS = ST\_5;

}

break;

case ST\_8:

NEXT\_STATUS = ST\_9;

break;

case ST\_9:

NEXT\_STATUS = ST\_0;

break;

}

}

SystemC timing evolution

# Lezione 3

L’obiettivo di questa lezione è studiare la simulazione del kernel di SystemC e l’algoritmo di scheduling che regola la simulazione.

## RTL

### La simulazione di SystemC

Durante l’esecuzione del codice opportunamente commentato noi possiamo distinguere alcune fasi:

* Inizializazione dello scheduling: tutti SC\_METHODS eseguiti una volta.
* il costrutto wait: sospende il testbench da 0s a 40 ns.
* la sequenza di esecuzione del divisor:
  + Il processo DIVISION è svegliato dal processo DIVISION\_FSM, che scrive nei segnali interni del modulo divisore.
  + Il processo DIVISION\_FSM è svegliato dal clock ad ogni ciclo di clock

Qui si può vedere l’output dell’esecuzione con le varie fasi evidenziate:

0 s - divisore: DIVISION\_FSM

Inizializzazione

0 s - divisore: reset

0 s - divisore: DIVISION

0 s - tb: begin run()

Reset the design!

0 s - tb: reset

0 s - divisore: DIVISION\_FSM

0 s - divisore: reset

10 ns - divisore: DIVISION\_FSM

10 ns - divisore: reset

20 ns - divisore: DIVISION\_FSM

Wait(5)

20 ns - divisore: reset

30 ns - divisore: DIVISION\_FSM

30 ns - divisore: reset

40 ns - divisore: DIVISION\_FSM

40 ns - divisore: reset

40 ns - tb: after wait(5)

The division of 183 for 86

40 ns - tb: write 183

40 ns - tb: write 86

40 ns - divisore: DIVISION

50 ns - divisore: DIVISION\_FSM

DIVISION\_FSM 🡪 DIVISION

Clock 🡪 DIVISION\_FSM

50 ns - divisore: Reset\_ST

50 ns - tb: wait result

50 ns - divisore: DIVISION

60 ns - divisore: DIVISION\_FSM

…

600 ns - divisore: ST\_9

600 ns - divisore: DIVISION

610 ns - divisore: DIVISION\_FSM

610 ns - divisore: ST\_0

is: -- quotient: 2 --reminder: 11

610 ns - tb: result available --quotient: 2 --reminder: 11

610 ns - tb: reset

Info: /OSCI/SystemC: Simulation stopped by user.

L’esecuzione può essere schematizzata nel modo seguente:

TB

Divisore

DIVISION

FSM

Divisore

DIVISION

Divisore

DIVISION

FSM

TB

Divisore

DIVISION

FSM

…

TB

Divisore

DIVISION

Divisore

DIVISION

FSM

Divisore

DIVISION

Divisore

DIVISION

FSM

TB

Divisore

DIVISION

…

Divisore

DIVISION

FSM

Divisore

DIVISION

Divisore

DIVISION

FSM

**STOP**

0

10

40

50

60

610

600

Wait result

Stop simulation

### File VDC

Come si può vedere da questa immagine tutti i segnali sono bassi fino al tempo 40ns. Al tempo 40ns, il testbench mette reset e number\_isready a 1 e setta dividend e divisor ai valori di input per il design. Al tempo 600ns, il divisore mette result\_isready a 1 e setta i valori di output quotient e reminder.

## TLM UT

Con il codice TLM UT opportunamente commentato con le sc\_time\_stamp ottengo il seguente risultato:

0 s [TB:] Calculating The division of 86 for 183

0 s [TB:] Invoking the b\_transport primitive - write

0 s [DIVISORE:] Received invocation of the b\_transport primitive - write

0 s [DIVISORE:] Invoking the divisore\_function to calculate the divisor

0 s [DIVISORE:] Calculating divisore\_function ...

0 s [TB:] Invoking the b\_transport primitive - read

0 s [DIVISORE:] Received invocation of the b\_transport primitive - read

0 s [DIVISORE:] Returning result: --quotient: 0 --reminder: 0

0 s [TB:] TLM protocol correctly implemented

0 s [TB:] Result is: -- quotient: 0 --reminder: 0

Info: /OSCI/SystemC: Simulation stopped by user.

Come si può notare la notazione di tempo non è necessaria, i tempi del simulatore sono infatti tutti a zero.

## TLM LT

Con il codice TLM LT opportunamente commentato con le sc\_time\_stamp ottengo il seguente risultato:

Info: /OSCI/SystemC: Simulation stopped by user.

0 s - top.initiator - run

0 s [TB:] Calculating The division of 86 for 183

0 s [TB:] Invoking the b\_transport primitive - write

0 s [DIVISORE:] Received invocation of the b\_transport primitive - write

0 s [DIVISORE:] Invoking the divisore\_function to calculate the divisor

0 s [DIVISORE:] Calculating divisore\_function ...

0 s [TB:] Invoking the b\_transport primitive - read

0 s [DIVISORE:] Received invocation of the b\_transport primitive - read

0 s [DIVISORE:] Returning result: --quotient: 0 --reminder: 86

0 s [TB:] TLM protocol correctly implemented

0 s [TB:] Result is: -- quotient: 0 --reminder: 86

Time: 0 s + 100 ns

Info: /OSCI/SystemC: Simulation stopped by user.

Come può notare anche qui il tempo del Simulatore è pari a zero. La notazione di tempo è data dal programmatore attraverso Timing annotation. Il simulatore continua a dare tempo zero per un quanto di tempo cioè fino a che non si sincronizza e prende il tempo del timimg annotion, ciò avviene per il temporal decoupling.

## TLM AT4

Con il codice TLM AT4 opportunamente commentato con le sc\_time\_stamp ottengo il seguente risultato:

0 s 0 s - top.initiator - run

0 s [TB:] Calculating The division of 86 for 183

0 s [TB:] Invoking the nb\_transport\_fw primitive of divisor - write

0 s [DIVISORE:] Received invocation of the nb\_transport\_fw primitive

0 s [DIVISORE:] Activating the IOPROCESS

0 s [DIVISORE:] End of the nb\_transport\_fw primitive

0 s [TB:] Waiting for nb\_transport\_bw to be invoked

0 s [DIVISORE:] IOPROCESS has been activated

100 ns [DIVISORE:] Invoking the divisore\_function to calculate the divisor

100 ns [DIVISORE:] Calculating divisore\_function ...

100 ns [TB:] Invoking the nb\_transport\_bw primitive - write

100 ns [TB:] Invoking the nb\_transport\_fw primitive of divisore - read

100 ns [DIVISORE:] Received invocation of the nb\_transport\_fw primitive

100 ns [DIVISORE:] Activating the IOPROCESS

100 ns [DIVISORE:] End of the nb\_transport\_fw primitive

100 ns [TB:] Waiting for nb\_transport\_bw to be invoked

100 ns [DIVISORE:] IOPROCESS has been activated

200 ns [DIVISORE:] Returning result: --quotient: 0 --reminder: 86

200 ns [TB:] Invoking the nb\_transport\_bw primitive - write

200 ns [TB:] TLM protocol correctly implemented

200 ns [TB:] Result is: -- quotient: 0 --reminder: 86

Info: /OSCI/SystemC: Simulation stopped by user.

Come si può notare qui il tempo del simulatore avanza grazie a wait() messe appositamente dal programmatore.

## Comparazione dei tempi

Come abbiamo notato abbiamo un avanzamento del tempo dato dal simulatore solo su RTL. Per i TLM le nozioni di tempo sono più astratte e date da annotazioni del programmatore.

SystemC Modelling al livello TLM

# Lezione 4

Il SystemC TLM standard definisce i protocolli di comunicazione per sviluppo di design al livello più astratto di RTL. Il livello TLM prevede l’utilizzo di due tipologie di modulo per il funzionamento di un sistema, queste due tipologie sono rispettivamente: Initiator e Target. Initiator e Target comunicano utilizzando specifiche primitive e sincronizzandosi tra loro attraverso socket di comunicazione.

## Untimed (UT)

Nella tipologia di TLM level UT il modulo Initiator è implementato nel file “divisore\_UT\_testbench.cc”, mentre il modulo Target nel file “divisore\_UT.cc”. Il modulo Target rappresenta nel nostro sistema, l’elaborazione vera e propria dell’algoritmo, il modulo Initiator richiama l’elaborazione per eseguire determinate simulazioni. L’Untimed utilizza la primitiva “b\_transport()”, differenziata in modalità Read e Write, per la comunicazione tra i due moduli. La b\_transport è implementata nel modulo Target e viene richiamata dal modulo Initiator per effettuare una simulazione (dopo ovviamente aver preparato adeguatamente i segnali di ingresso al modulo Target) in modalità Write, successivamente il Target comincerà ad elaborare i segnali in ingresso per calcolarne il risultato dell’algoritmo di divisione. Siccome Target e Initiator sono concorrenti, l’Initiator richiama una seconda volta la primitiva b\_transport in modalità Read per poter leggere il risultato del modulo Target. Questo darà la disponibilità alla lettura del payload, (pacchetto di dati che trasferisce la b\_transport), solo successivamente al “TLM\_OK\_RESPONSE” che rappresenta la fine dell’elaborazione del Target.

## Loosley Timed (LT)

Anche in questo caso l’elaborazione si divide in modulo Target e Initiator che comunicano tra loro attraverso i rispettivi socket. Come per l’UT l’Initiator richiama la primitiva b\_transport, implementata nel modulo Target, per richiedere al Target l’esecuzione dell’elaborazione. Il nome stesso del coding style identifica come il tempo giochi un ruolo chiave nell’esecuzione del sistema, infatti, si differenzia dall’Untimed in quanto è necessario considerare una sincronizzazione a livello temporale tra i due moduli Target e Initiator. Anche qui l’Initiator richiama la b\_transport in modalità Write per inviare il payload contenente gli input per l’elaborazione al Target. Il Target ricevuto il payload comincia la sua elaborazione e una volta terminata risponde all’Initiator con il messaggio “TLM\_OK\_RESPONSE”. Siccome i moduli sono concorrenti l’Initiator richiama una seconda volta la b\_transport in modalità Read per leggere i risultati dell’esecuzione che saranno disponibili solo dopo il “TLM\_OK\_RESPONSE”. Il tempo viene gestito in modalità di sincronizzazione dei due moduli e visualizzato in fase di esecuzione, in modo da determinare chi, e in quale istante di tempo, sta facendo cosa.

## APPROXIMATELY TIMED (AT4)

L’Approximately Timed si distingue dagli stili precedenti per l’utilizzo di due primitive:

* nb\_transport\_fw: implementata nel Target viene invocata dall’Initiator per richiedere l’esecuzione dell’algoritmo di divisione al Target o richiederne i risultati (sostituisce approssimativamente la b\_transport nei coding styles precedenti);
* nb\_transport\_bw: implementata nell’Initiator, viene invocata dal Target quando il Target ha terminato il calcolo dell’algoritmo e per segnalare all’Initiator la possibilità di ricevere i risultati dell’algoritmo.

L’Approximately Timed distingue quattro fasi operative che identificano richiesta e risposta della transport forward e richiesta e risposta della transport backward. Il tempo anche qui gioca un ruolo fondamentale.

## Comparazione dei tempi

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | TLM UT | TLM LT | TLM AT4 | RTL |
| Real Time | 0.064 | 0.099 | 0.275 | 21.032 |
| User Time | 0.060 | 0.092 | 0.268 | 20.441 |
| System Time | 0.000 | 0.004 | 0.004 | 0.548 |

Nel TLM la simulazione è molto veloce e legata solo alla complessità della funzionalità del modulo, infatti abbiamo tempi molto bassi. Il TLM – UT è quello che ci impiega il tempo minore. Invece in RTL la simulazione è più lenta infatti con la stessa quantità di divisioni impiega 20.441 s.

Assertion

# Lezione 5 - Assertion

L’obbiettivo di questa consegna è creare un simulatore che verifichi la corretta, o meno, esecuzione del sistema, valutando se esso soddisfa o meno tre proprietà. Le proprietà, come da richiesta, devono rispettivamente: le prime due definire una caratteristica sempre vera del sistema e l’ultima una caratteristica sempre falsa.

Ho implementato tre proprietà:

## Proprietà 1

La proprietà 1 è la seguente:

void root\_RTL :: property1(void) {

  // if number\_isready == 1 then result\_isready == 1

  // in less than 100 clock cycles

  int count = 0;

  bool true\_property = false;

  while(true){

    if (number\_isready.read() == 1){

      while((count < 100)&(!true\_property)){

      wait();

      if (result\_isready.read() == 1)

    true\_property = true;

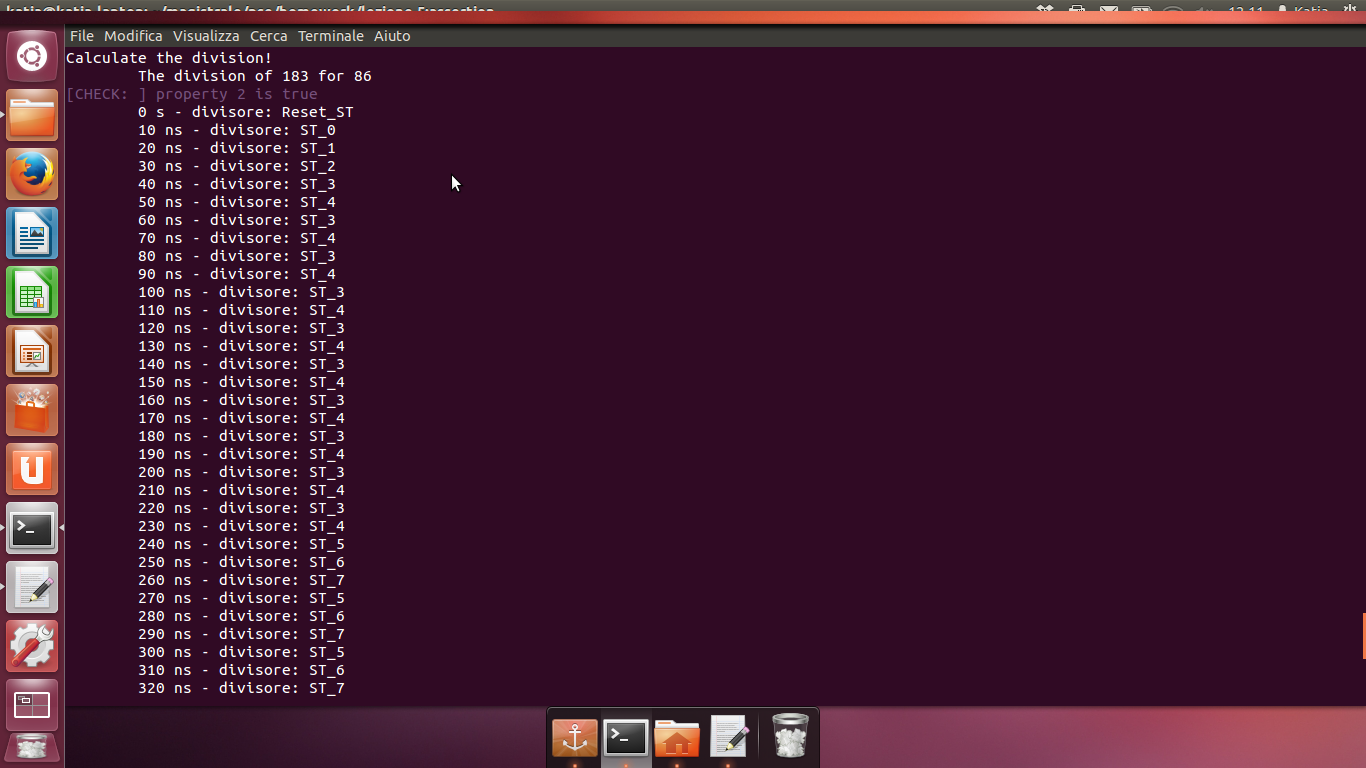
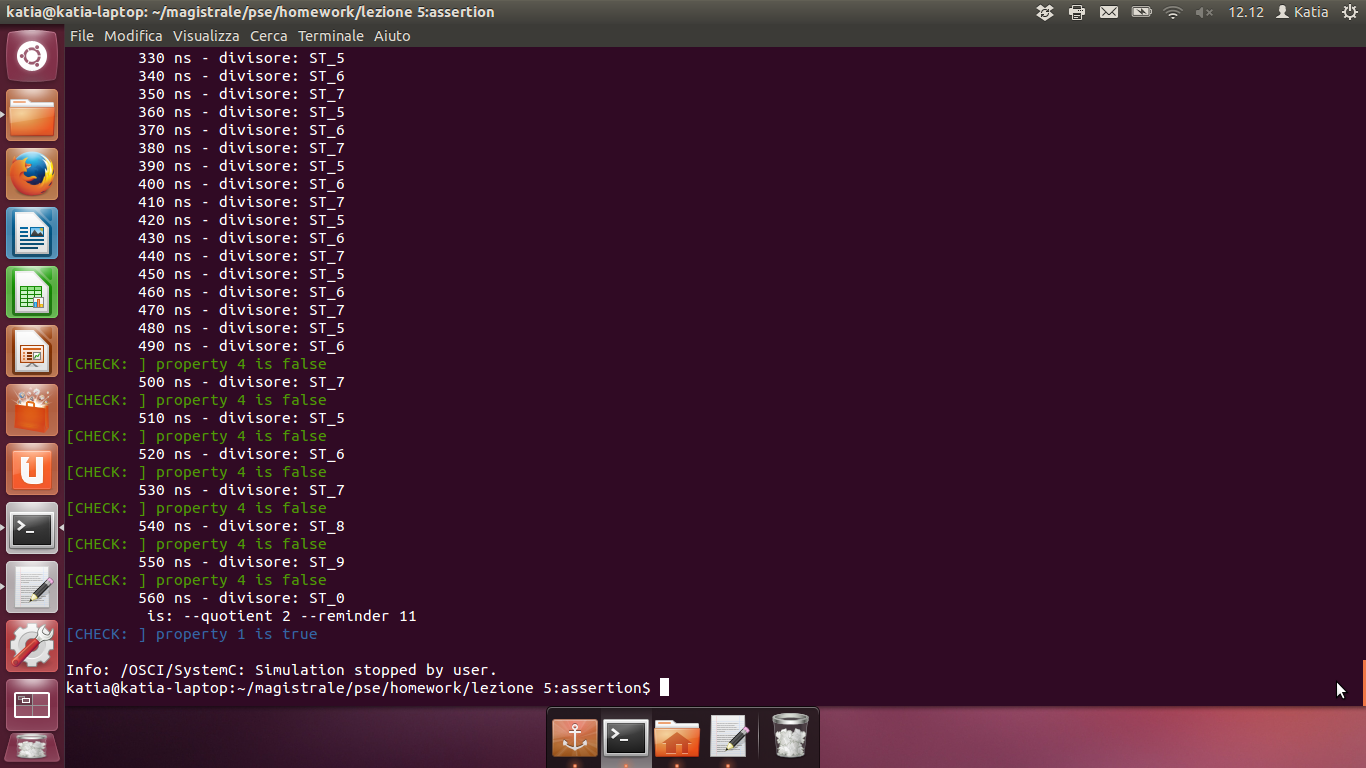
      }

      if (true\_property)

    cout<<"\033[34m"<<"[CHECK: ] property 1 is true"<<"\033[0m"<<endl;

      else {

    cout<<"\033[34m"<<"[CHECK: ] property 1 is false"<<"\033[0m"<<endl;



    sc\_stop();

      }

      count = 0;

      true\_property = false;

    }

    wait();

  }

}

Tale proprietà è sempre vera nel mio codice. Come si vede nella immagine a fianco.

## Proprietà 2

La proprietà 2 è la seguente:

void divisore\_RTL :: property2(void){

// if number\_isready == 0 then quotient\_port == 0 and reminder\_port == 0 and result\_isready == 0

while(true){

if (number\_isready.read() == 0){

if (((quotient\_port.read() == 0)&&(result\_isready.read() == 0))&&(reminder\_port.read() == 0))

cout<<"\033[35m"<<"[CHECK: ] property 2 is true"<<"\033[0m"<<endl;

else{

cout<<"\033[35m"<<"[CHECK: ] property 2 is false"<<"\033[0m"<<endl;

sc\_stop();

}

}

wait();

}

}

Tale proprietà è sempre vera nel mio codice. Come si vede nella immagine a fianco.

## Proprietà 3

La proprietà 3 è la seguente:

void divisore\_RTL :: property3(void){

// if Dividend < Divisor then STATUS = ST4

while(true){

if (Dividend.read() < Divisor.read()){

if (STATUS.read() == ST\_4) {

cout<<"\033[32m"<<"[CHECK: ] property 4 is true"<<"\033[0m"<<endl;

sc\_stop();

}else{

cout<<"\033[32m"<<"[CHECK: ] property 4 is false"<<"\033[0m"<<endl;

//sc\_stop();

}

}

wait();

}

}

Tale proprietà è sempre falsa nel mio codice. Come si vede nella immagine a pagina precedente.

Mixed RTL/TLM modelling : Transactor

# Lezione 5 - Transactor

L’obbiettivo del progetto è realizzare un’interfaccia di collegamento tra il modulo RTL e TLM-LT.   
Tale interfaccia prende il nome di transattore**.** Il transattore è descritto e implementato dai file.

Porte RTL

TLM socket

DIVIDER

Testbemch

DIVIDER

design

Transactor

TLM-LT

RTL

## Mapping di porte e segnali tra TLM e RTL

Il transattore, in quanto modulo di collegamento tra i moduli testbench, a livello TLM-LT e divisore RTL. Deve mappare correttamente tra loro ingressi e uscite del TLM e dell’RTL. Il transattore riceve i segnali di input dal TLM quando questo invoca la b\_transport implementata all’interno del transattore stesso. L’implementazione dei segnali che il TLM passa al transattore, è contenuta nel payload che la b\_transport richiede come parametro. Il transattore a sua volta collega il valore dei segnali passati dal TLM con i valori di input dei segnali del modulo RTL, in più pone il segnale number\_isready uguale a 1 così da permettere l’esecuzione del modulo RTL. A esecuzione terminata il modulo RTL risponde con i segnali di quotient e di reminder e il segnale di result\_isreadyinizializzato a 1, (indicante la fine dell’esecuzione RTL), al transattore, il quale notifica al modulo TLM attraverso lo status “TLM\_OK\_RESPONSE”. Come si può vedere le flags di collegamento tra testbencher e RTL non sono più necessarie in quanto il transattore (modulo intermedio) è in grado di stabilire, e notificare ai rispettivi moduli, partenza e termine dell’esecuzione.

## Traduzione da funzioni a operazioni

Il transattore, oltre a mappare i segnali, deve tradurre funzioni a livello TLM in operazioni a livello RTL. Il transattore implementa due thread che identificano rispettivamente la Write e la Read.   
Tali thread vengono attivate quando il modulo TLM invoca la b\_transport settando il parametro trans\_command rispettivamente a Write o a Read. Sostanzialmente, quando il TLM richiama la b\_transport implementata nel transattore, all’interno della b\_transport vi sarà una funzione notify che, a seconda che il trans\_command sia Read o Write, “sveglia” la thread Read o Write. Così facendo il transattore fa da perfetto bridge tra primitive ad alto livello TLM e operazioni a basso livello RTL.

reset\_to\_RTL.write(1);

p\_Out\_data\_dividend.write(ioDataStruct.datain\_dividend);

p\_Out\_data\_divisor.write(ioDataStruct.datain\_divisor);

p\_Out\_enable.write(1);

while(p\_In\_enable.read() != 1)

wait();

ioDataStruct.quotient=p\_In\_data\_quotient.read();

ioDataStruct.reminder=p\_In\_data\_reminder.read();

Number\_isready

Dividend\_port

Divisor\_port

resett

Result\_isready

quotient\_port

reminder\_port

Set\_write

b\_transport

Set\_read

b\_transport

testbench

transactor

RTL

## Comportamento del transattore

Come già detto in precedenza il transattore è un modulo che si interpone tra testbench TLM e divisore\_RTL. Il suo compito è quello di fare il mapping tra modulo TLM e i segnali del modulo RTL, inoltre deve tradurre le funzioni TLM in operazioni RTL. Il TLM, che è il testbench, fornisce al transattore i valori dei segnali di ingresso per il calcolo dell’algoritmo. L’implementazione di questi segnali è passata al transattore attraverso il payload della b\_transport. Il payload, che conterrà quindi i segnali sopra elencati, sarà una struttura i cui campi sono i valori dei segnali da inviare, tale struttura è definita nel file define.h. Successivamente, la b\_transport, richiamata dal TLM in modalità di scrittura, esegue un notify a WRITEPROCESS che “sveglia” la thread e esegue l’RTL. Tale thread, che è associata alla funzione di scrittura della b\_transport, ha il compito di implementare le porte in ingresso al modulo RTL con i valori del payload e di “lanciare” l’esecuzione del modulo RTL ponendo il segnale number\_isready uguale a 1. A questo punto il modulo RTL può procedere con il calcolo dell’algoritmo. In contemporanea il TLM richiama nel transattore la b\_transport in modalità Read.   
Analogamente per il caso Write, la b\_transport conterrà una notify che “sveglia” un READPROCESS che ha il compito di prelevare i risultati dell’algoritmo calcolato dal modulo RTL e inerirli nella structure predefinita. Fatto ciò il READPROCESS notifica alla b\_transport che i risultati sono pronti ed essa a sua volta invia al TLM il comando “TLM\_OK\_RESPONSE”, il quale consente al TLM di leggere i risultati contenuti nel payload. Tutto ciò si può vedere nell’output seguente:

0 - topl.i\_divisore\_RTL\_transactor - reset

0 s - divisore: DIVISION\_FSM

Inizializzazione

0 s - divisore: reset

0 s - divisore: DIVISION

0 s - topl.i\_src\_LT - run

Calculating The division of 86 for 183

Invoking the b\_transport primitive - write

0 - topl.i\_divisore\_RTL\_transactor - b\_transport

0 s - divisore: DIVISION\_FSM

0 s - divisore: reset

write notified

0 - topl.i\_divisore\_RTL\_transactor - notify received

0 - topl.i\_divisore\_RTL\_transactor - b\_transport ended

Invoking the b\_transport primitive - read

0 - topl.i\_divisore\_RTL\_transactor - b\_transport

0 s - divisore: DIVISION

read notified

0 - topl.i\_divisore\_RTL\_transactor - notify received

10 ns - divisore: DIVISION\_FSM

10 ns - divisore: Reset\_ST

Elaborazione RTL

…

70 ns - divisore: DIVISION

80 ns - divisore: DIVISION\_FSM

80 ns - divisore: ST\_0

80 - topl.i\_divisore\_RTL\_transactor - b\_transport ended

TLM protocol correctly implemented

Result is: -- quotient: 0 --reminder: 86

Time: 80 ns + 0 s

Trans WRITE + ret

Divisore

DIVISION

FSM

Divisore

DIVISION

Divisore

DIVISION

FSM

TB

Divisore

DIVISION

FSM

Trans B\_TR

Divisore

DIVISION

Trans READ

Trans READ

Divisore

DIVISION

FSM

TB

Divisore

DIVISION

…

Divisore

DIVISION

FSM

Divisore

DIVISION

Divisore

DIVISION

FSM

**STOP**

0

10

Stop simulation

Trans RESET

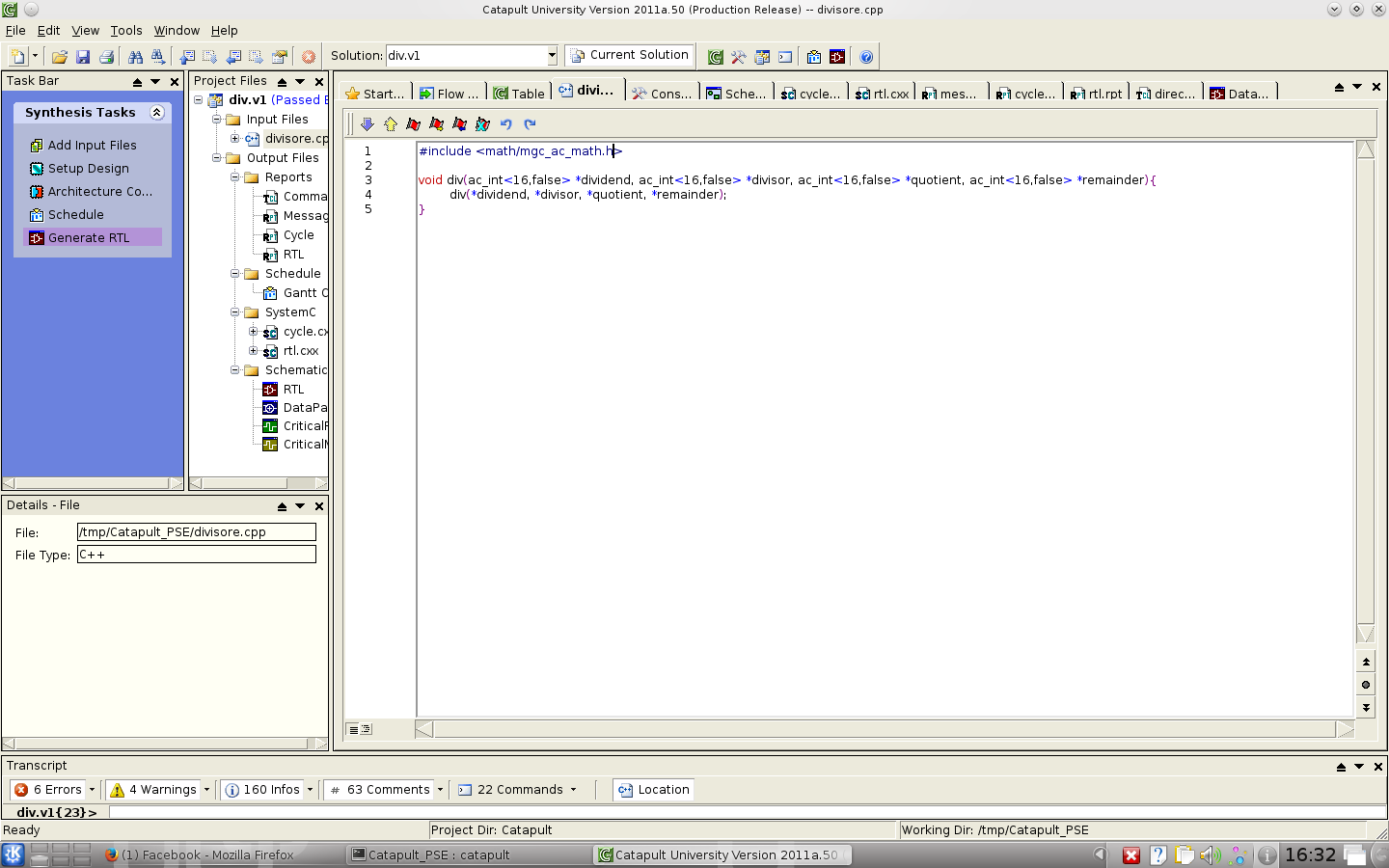
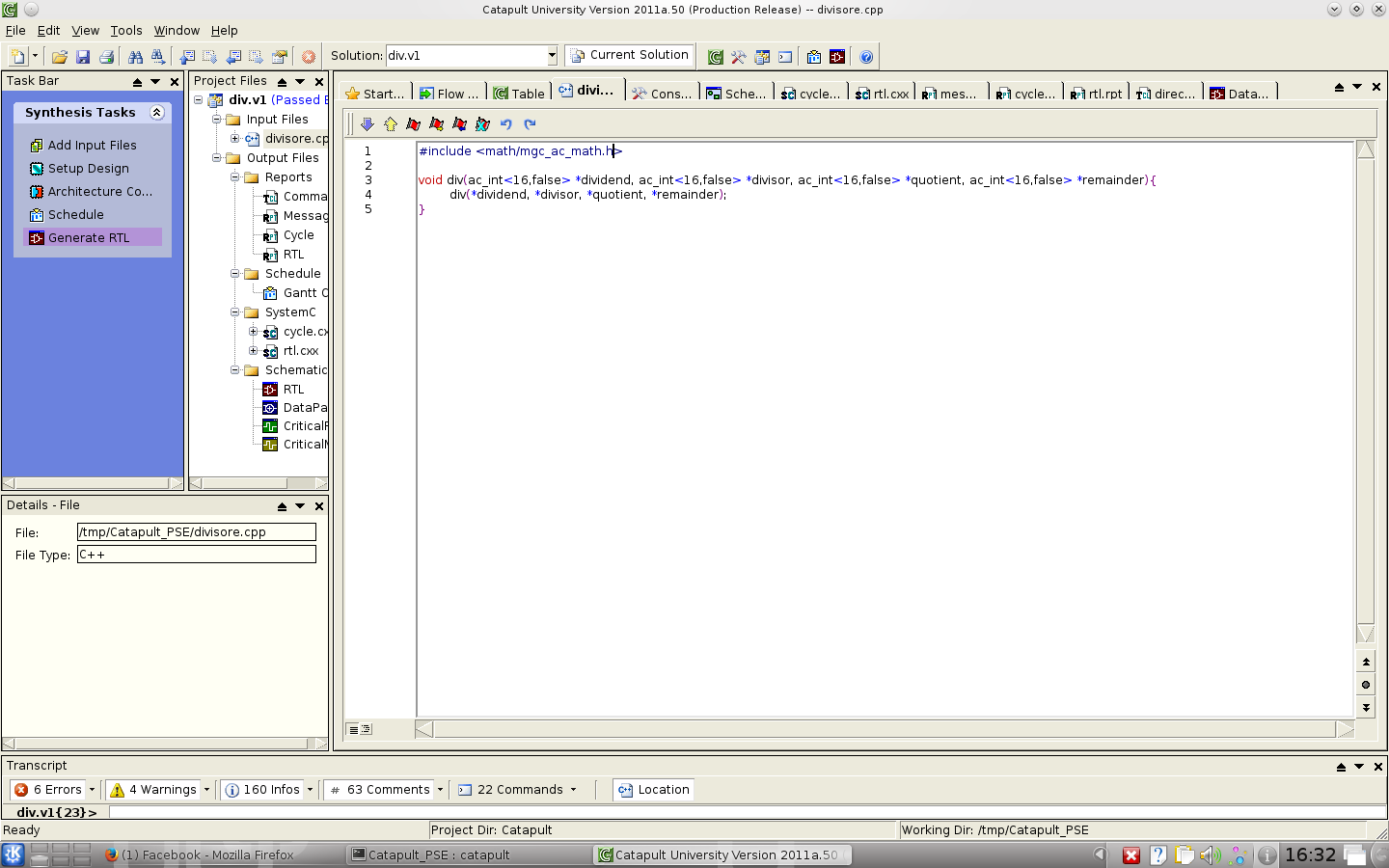
Trans B\_TR

Sintesi automatica dalla descrizione C++

# Lezione 6

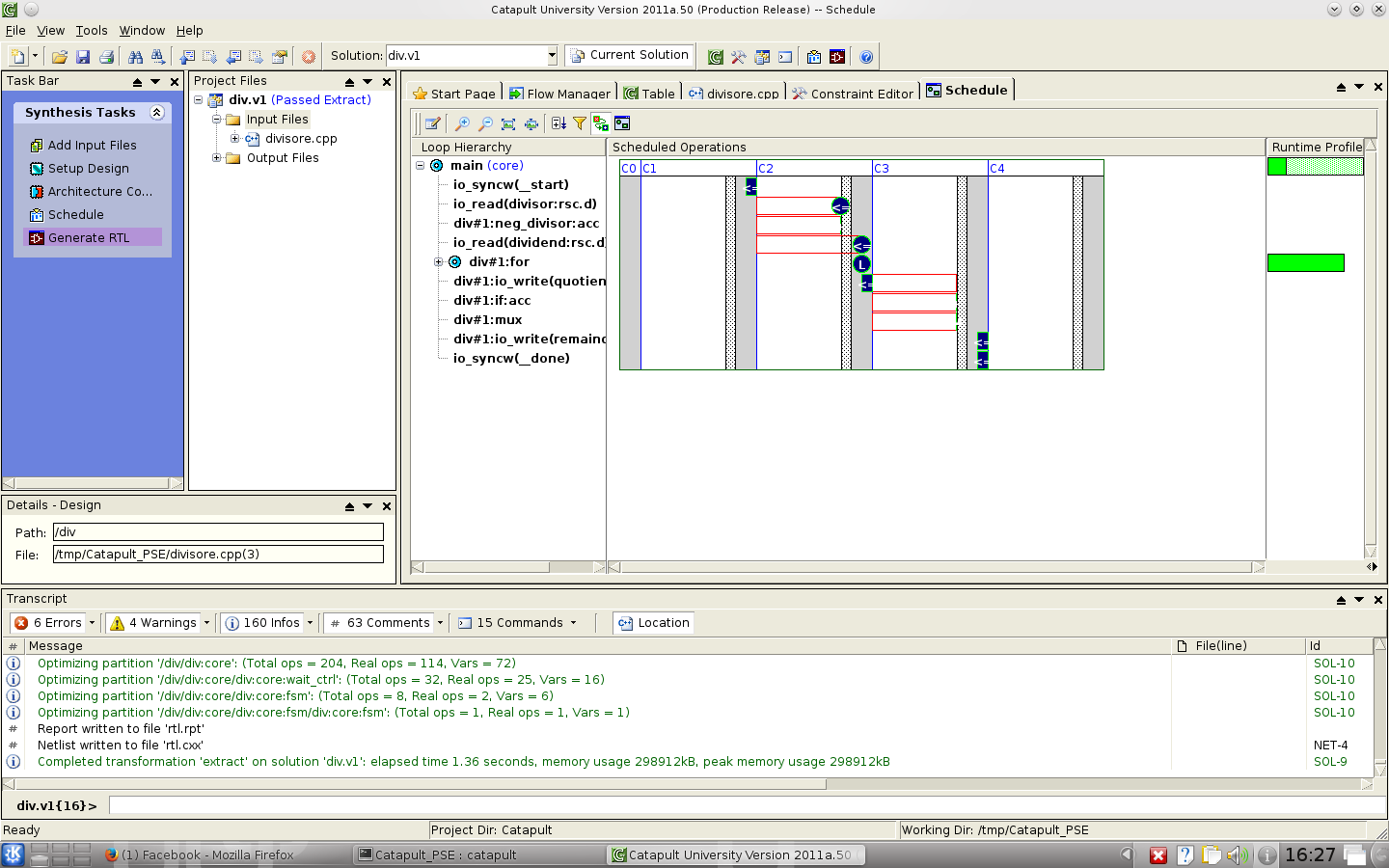
L’obiettivo di questa lezione è quello di applicare il processo di sintesi di Catapult C al device Driver.

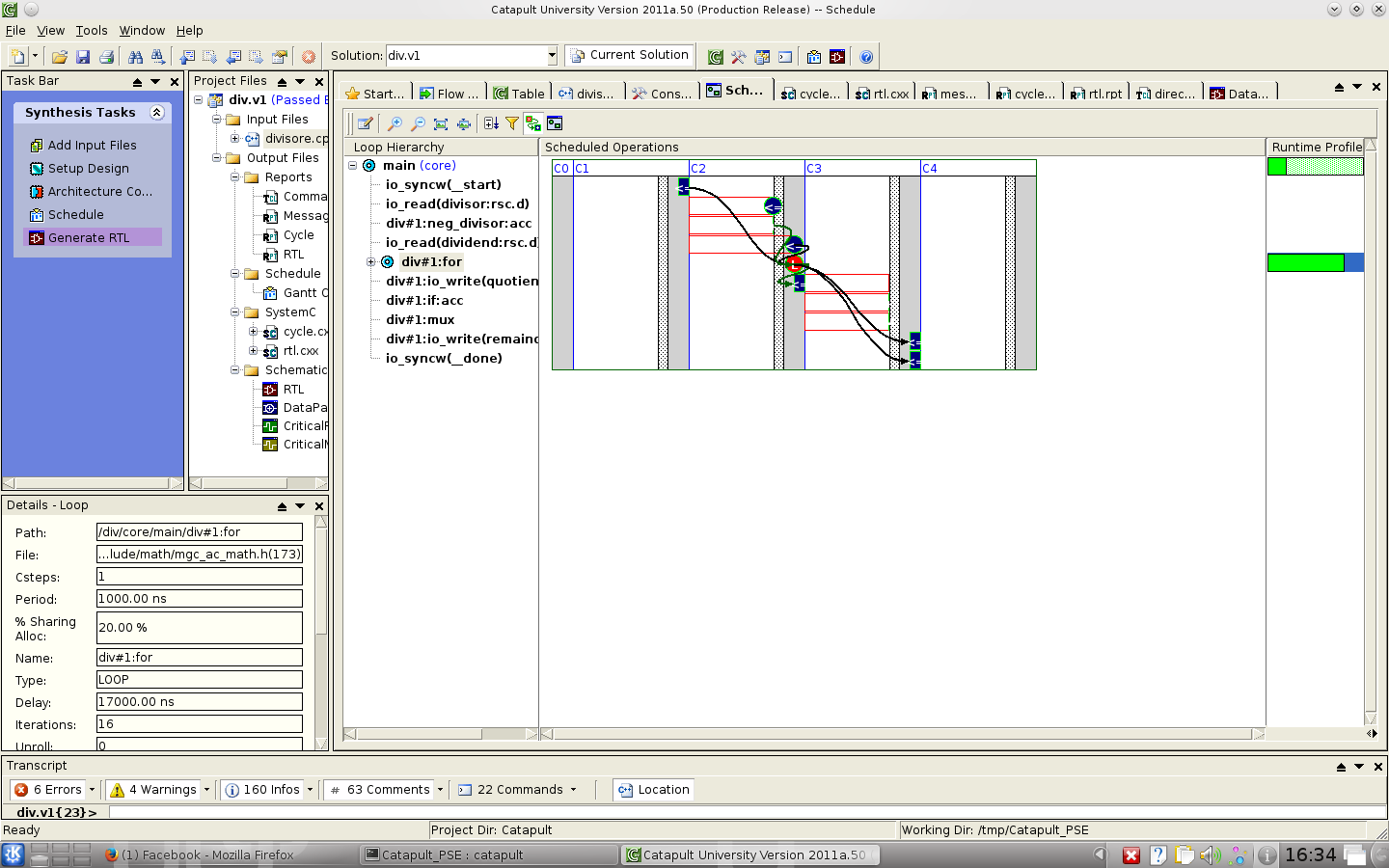
## Codice in C++

Come si può vedere nella schermata seguente abbiamo inserito il codice C++ della funzionalità da implementare in Catapul C.

## Schedule

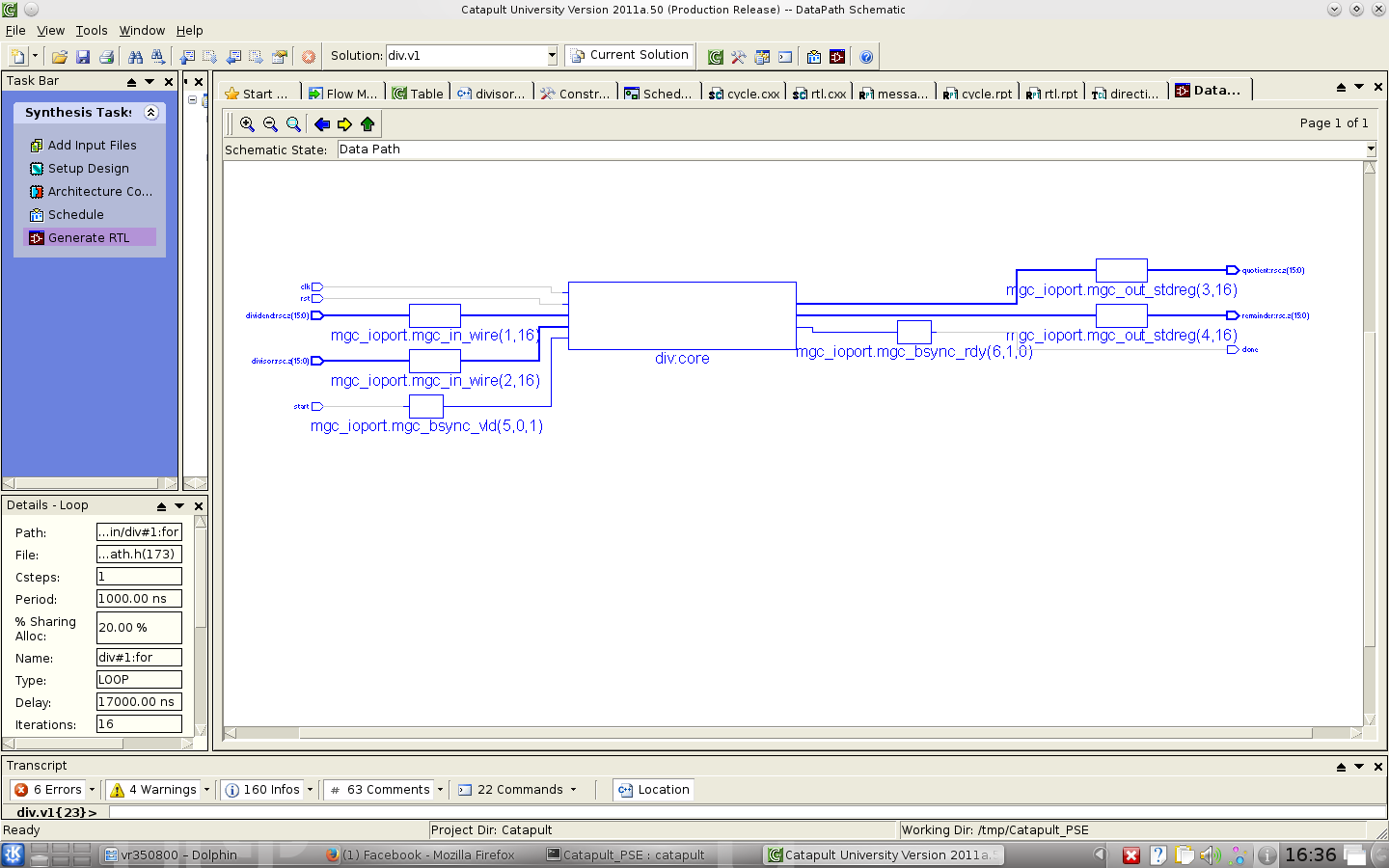
Alla fine delle devo applicare l’algoritmo di schedule. Catapult schedula il design e genera un Gantt chart per dare una visione grafica dell’algoritmo e della architettura.





## Output RTL

Alla fine Catapult genera il seguente schema RTL:



## Confronto

Se compariamo la generazione dell’RTL automatico e quello generato manualmente si vede che con la generazione automatica è più efficiente per quanto riguarda il tempo e gli errori prodotti ma il codice è illegibile.

Testbench e device drivers

# Lezione 7

L’obiettivo di questa lezione è di scrivere un device driver per il modulo Divisore.

## Device driver

Il file divisore.cc è un driver di secondo livello. Il file sclink.cc è il driver di primo livello e fornisce al device driver di secondo livello due funzionalità:

* Sclink\_read: funzione usata per leggere i registri del device;
* Sclink\_write: funzione usata per scrivere sui registri del device.

La prima cosa che il driver deve fare quando crea un character device driver è l’inizzializzazione del device stesso. La funzione necessaria per compiere questo compito è register\_chrdev\_region. Questa funzione è usata per registrare un device nel kernel e assegnare ad esso due numeri, uno che servirà per identificarlo e l’altro che rappresenta il specifico device.

static int divisore\_init(void)

{

int err = register\_chrdev(DIVISORE\_DEV\_MAJOR\_NUMBER, "divisore", &divisore\_fops);

if (err < 0) return err;

printk(KERN\_INFO "Registered characted device divisore\n");

return 0;

}

Dopo di che è dichiarata una struttura file\_operations per definire le funzioni accessibili dal device driver.

struct file\_operations divisore\_fops = {

.ioctl = divisore\_ioctl

};

Queste operazioni sono soprattutto di implementazione delle chiamate a sistema. Ogni campo della struttura deve puntare alla funzione del device driver che implementa l’operazione specifica, o si lascia NULL per quelle operazioni non sopportate. Io ho usato solo la funzione ioctrl, che è usata per descrivere i comandi:

int (\*ioctrl) (struct inode \*inode, struct file \*filp, unsigned int cmd, unsiged long arg);

I parametri inode e file si riveriscono al device. Il parametro cmd indica il tipo di comando. Il parametro arg è usato per passare parametri al device driver. L’implementazione di ioctrl consiste in un grande blocco switch che seleziona il corretto comportamento in accordo con il parametro cmd. Diversi comandi hanno valori numerici, i quali sono spesso associati a nomi simbolici per semplificare il codice.

static int divisore\_ioctl(struct inode\* inode, struct file\* file, unsigned int

command, unsigned int arg)

{

switch (command) {

case DIVISORE\_IOCTL\_RESET:

return do\_reset();

case DIVISORE\_IOCTL\_READ:

return do\_read((read\_command\_arg\*) arg);

case DIVISORE\_IOCTL\_WRITE:

return do\_write((write\_command\_arg\*) arg);

default: // Unknown/unsupported command

return -1;

}

}

La funzione ioctrl invoca altre funzioni che implementano le singole funzionalità. Per esempio, qui c’è il codice della do\_write usata per scrive i numeri di input nel device, insieme con la dichiarazione della struttura usata per passare i dati:

int do\_write(write\_command\_arg\* arg)

{

divisore\_send\_data data;

divisore\_send\_data2 data2;

data.dividend\_port = arg->value1;

data.divisor\_port = arg->value2;

sclink\_write(DIVISORE\_ADDR, \*((unsigned int\*)&data));

data2.number\_isready = 1;

data2.reset = 1;

sclink\_write(DIVISORE\_ADDR, \*((unsigned int\*)&data2));

return 1;

}

Per implementare le funzionalità del device, questa funzione usa un certo numero di sclink\_read e sclink\_write.

VHDL modelling al livello RTL

# Lezione 10

Scrivere il modulo Driver in VHDL.

## VHDL

La descrizione del device è divisa in tre parti. La prima è la package declaration. Nel mio caso, il packages è usato per definire alcune costanti:

PACKAGE divisore\_pack IS

CONSTANT SIZE : INTEGER := 16;

CONSTANT Reset\_ST : INTEGER := 0;

CONSTANT ST\_0 : INTEGER := 1;

CONSTANT ST\_1 : INTEGER := 2;

CONSTANT ST\_2 : INTEGER := 3;

CONSTANT ST\_3 : INTEGER := 4;

CONSTANT ST\_4 : INTEGER := 5;

CONSTANT ST\_5 : INTEGER := 6;

CONSTANT ST\_6 : INTEGER := 7;

CONSTANT ST\_7 : INTEGER := 8;

CONSTANT ST\_8 : INTEGER := 9;

CONSTANT ST\_9 : INTEGER := 10;

END divisore\_pack;

La seconda parte della descrizione del mio modulo divisor è la entity definition:

entity divisore is

port ( clock, reset : in bit;

divisor\_port : in UNSIGNED (SIZE-1 DOWNTO 0);

dividend\_port : in UNSIGNED (SIZE-1 DOWNTO 0);

number\_isready : in bit;

quotient\_port : out UNSIGNED (SIZE-1 DOWNTO 0);

reminder\_port : out UNSIGNED (SIZE-1 DOWNTO 0);

result\_isready : out bit

) ;

end divisore;

Una volta che la sua interfaccia è stata specificata in una entity declaration, una o più implementazioni di una entity possono essere descritte in più architecture bodies. La dichiarazione in un architecture body definisce gli oggetti che saranno usati per costruire una descrizione strutturale.

architecture divisore of divisore is

subtype status\_t is integer range 0 to 10;

subtype internal\_t is UNSIGNED (SIZE-1 DOWNTO 0);

signal STATUS: status\_t;

signal NEXT\_STATUS: status\_t;

signal Divisor: internal\_t;

signal Divisor\_Shift: internal\_t;

signal Dividend: internal\_t;

signal Quotient: internal\_t;

signal Reminder: internal\_t;

CONSTANT ZERO : internal\_t := "0000000000000000";

CONSTANT UNO : internal\_t := "0000000000000001";

begin

…

Il modulo divisore è implementato attraverso due processi:

* Il primo è di tipo uno: esso è sensibile a tutti i senali letti e descrive un circuito combinatorio. Questo processo calcola lo stato prossimo e l’evoluzione del sistema.
* Il secondo è del quarto tipo: esso è sensibile al clock e ha un segnale di reset asincrono. Il processo aggiorna il valore dello stato corrente, esso legge dalle porte di input e scrive nelle porte di output.

|  |  |
| --- | --- |
| process(STATUS, number\_isready, Divisor, Divisor\_Shift, Dividend)  -- elaborate\_DIVISION  begin  case STATUS is  when Reset\_ST =>  NEXT\_STATUS<=ST\_0;  when ST\_0 =>  if number\_isready = '1' then  NEXT\_STATUS<=ST\_1;  else  NEXT\_STATUS<=ST\_0;  end if;  when ST\_1 =>  NEXT\_STATUS<=ST\_2;  when ST\_2 =>  if Divisor /= ZERO then  NEXT\_STATUS <= ST\_3;  else  NEXT\_STATUS <= ST\_9;  end if;  when ST\_3 =>  if Dividend < Divisor then  NEXT\_STATUS <= ST\_8;  else  NEXT\_STATUS <= ST\_4;  end if;  when ST\_4 =>  if Divisor\_Shift(SIZE-1) /= '1' then  NEXT\_STATUS <= ST\_3;  else  NEXT\_STATUS <= ST\_5;  end if;  when ST\_5 =>  NEXT\_STATUS<=ST\_6;  when ST\_6 =>  NEXT\_STATUS<=ST\_7;  when ST\_7 =>  if Divisor = Divisor\_Shift then  NEXT\_STATUS <= ST\_8;  else  NEXT\_STATUS <= ST\_5;  end if;  when ST\_8 =>  NEXT\_STATUS<=ST\_9;  when ST\_9 =>  NEXT\_STATUS<=ST\_0;  when others =>  NEXT\_STATUS<=STATUS;  end case;  end process; | process(clock, reset)  -- elaborate\_DIVISION\_FSM  begin  if reset='0' then  STATUS<=Reset\_ST;  elsif clock'event and clock='1' then  STATUS<=NEXT\_STATUS;  case NEXT\_STATUS is  when Reset\_ST =>  quotient\_port <= ZERO;  reminder\_port <= ZERO;  result\_isready <= '0' ;  when ST\_0 =>  quotient\_port <= ZERO;  reminder\_port <= ZERO;  result\_isready <= '0' ;  Divisor <= ZERO;  Divisor\_Shift <= ZERO;  Dividend <= ZERO;  Quotient <= ZERO;  Reminder <= ZERO;  when ST\_1 =>  Divisor <= divisor\_port;  Dividend <= dividend\_port;  when ST\_2 =>  Divisor\_Shift <= Divisor;  when ST\_3 =>  if Dividend<Divisor then  Quotient <= ZERO;  end if;  when ST\_4 =>  if Divisor\_Shift(SIZE-1) /= '1' then  Divisor\_Shift <= (Divisor\_Shift sll 1);  end if;  when ST\_5 =>  Quotient <= Quotient sll 1;  when ST\_6 =>  if Dividend >= Divisor\_Shift then  Dividend <= Dividend - Divisor\_Shift;  Quotient <= Quotient + UNO;  end if;  when ST\_7 =>  Divisor\_Shift <= (Divisor\_Shift srl 1);  when ST\_8 =>  Reminder <= Dividend;  when ST\_9 =>  quotient\_port <= Quotient;  reminder\_port <= Reminder;  result\_isready <= '1';  end case;  end if;  end process; |
|  |  |

## Simulazione

Per la simulazione ho utilizzato il file qui sotto riscritto:

add wave \*

add wave /divisore/\*

force /divisore/clock 1 20 ns, 0 40 ns -repeat 40

force /divisore/reset 1 0

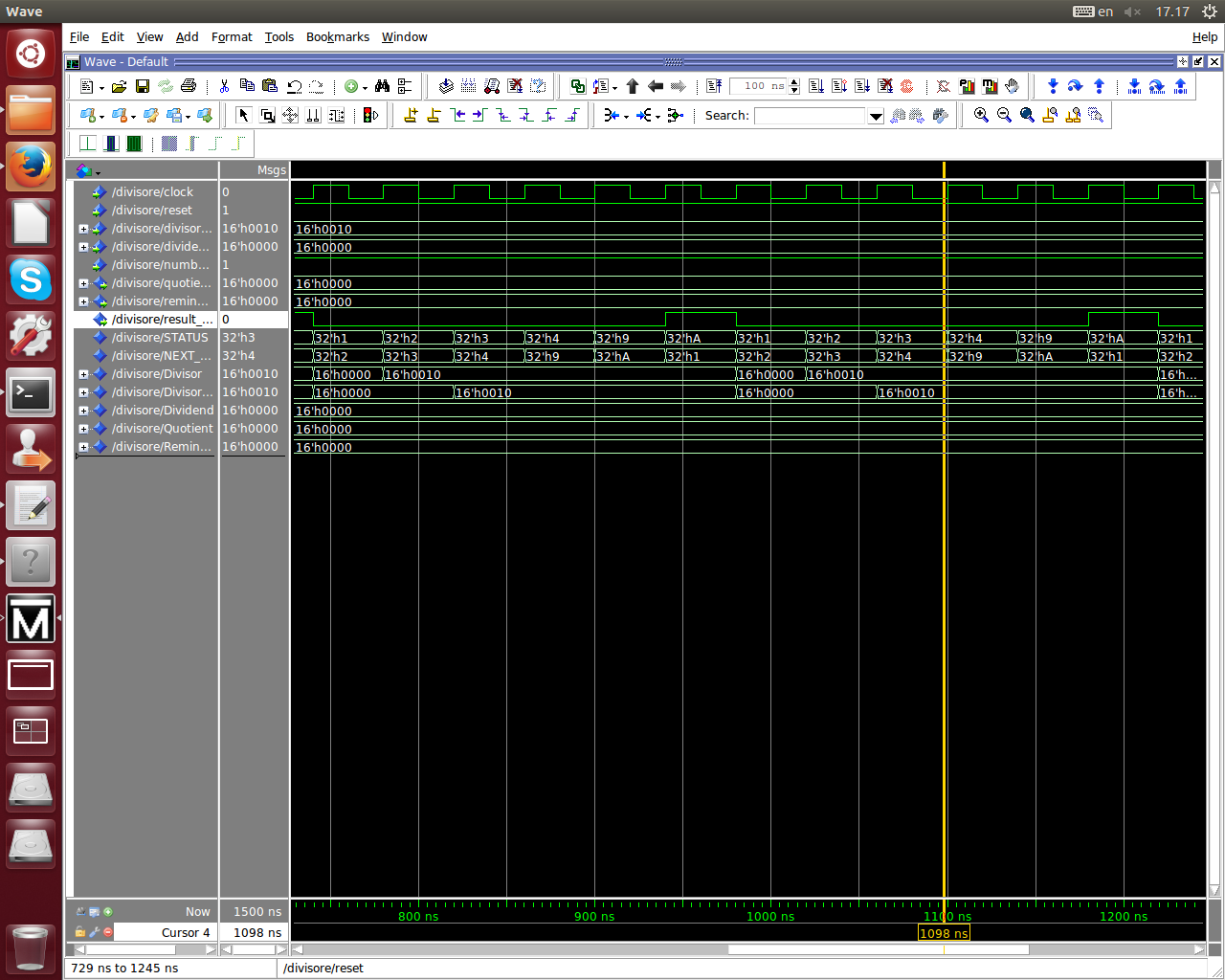
force /divisore/dividend\_port 0000000000100000 0

force /divisore/divisor\_port 0000000000000010 0

force /divisore/number\_isready 1 1

run 1500

quit

Ottengo quindi il seguente waveform:

Come si può notare la waveform i segnali di STATUS e NEXT\_STATUS cambiano seconda dei due processi scritti. Si può notare il risultato della siulazione è

Timing simulation in VHDL

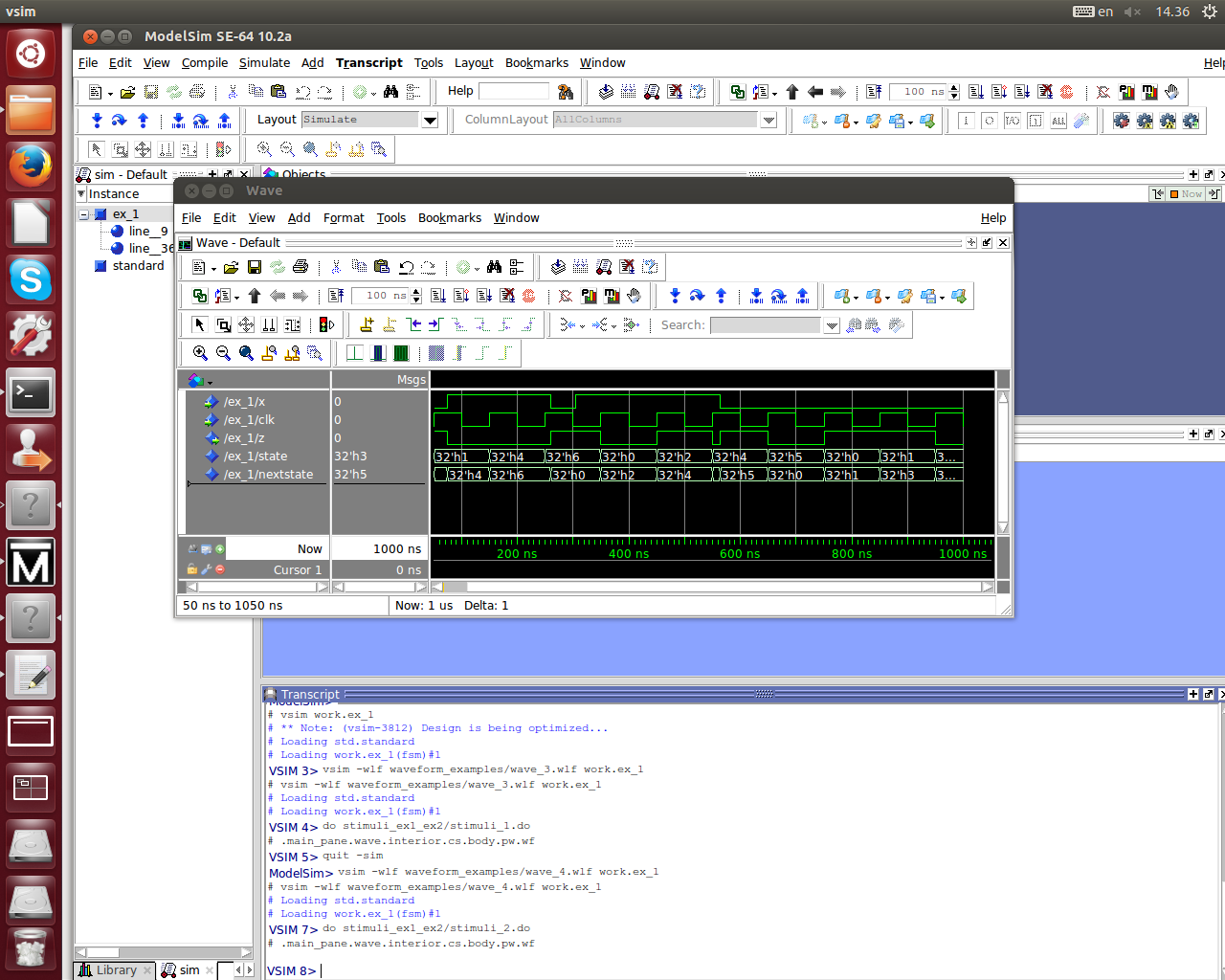
# Lezione 11

Questa lezione ha come obbiettivo l’analisi dei comportamento dei segnali in VHDL. Attraverso i seguenti esercizi:

## Ex\_1

### Ex\_1 stimuli\_1

### Ex\_1 stimuli\_2



### Ex\_1 stimuli\_3

### Confronti

#### Ex\_1 stimuli\_1 e Ex\_1 stimuli\_2

Come si può vedere ci sono molteplici differenze. In particolare si può notare come un ritardo nell’assegnamento del segnale provochi il medesimo ritardo nel cambiamento del valore z e di nextstate ma non influisce sullo stato in quanto il cambiamento è inferiore ad un ciclo di clock. Negli istanti tra 305 e 350 invece il cambiamento non si propaga su z e nextstate perchè essendo nello stato 6 non o una transazione per x=’1’ ma ho solo quella per x=’0’come si vede dal codice:

when 6 => if x=’0’ then z<=’1’; nextstate<=0; end if;

Alla fine invece un cambiamento permanente del segnale influisce immediatamente su nextstate e dal ciclo successivo anche su state.

#### Ex\_1 stimuli\_1 e Ex\_1 stimuli\_3

Come si può vedere ci sono molteplici differenze. In particolare si può notare come un anticipo nell’assegnamento del segnale x provochi cambiamenti del valore z, di nextstate e influisce anche sullo stato. Alla fine invece un cambiamento permanente del segnale influisce immediatamente su nextstate e dal ciclo successivo anche su state.

#### Ex\_1 stimuli\_2 e Ex\_1 stimuli\_3

Come si può vedere ci sono molteplici differenze. In particolare si può notare come un anticipo nell’assegnamento del segnale x provochi cambiamenti del valore z, di nextstate e influisce anche sullo stato. Alla fine invece in entrambi gli stimoli a x si assegna 0 a istanti diversi (563ns e 612ns) questo provoca cambiamenti sul tempo di assegnamento di z e nextstate ma non varia per lo state in quanto il cambiamento è inferiore ad un ciclo di clock e il cambiamento non si propaga.

## Ex\_2

### Ex\_2 stimuli\_1

### Ex\_2 stimuli\_2

### Ex\_2 stimuli\_3

### Confronti

Le differenze tra ex\_1 e ex\_2 consiste nel fatto che nell’ex\_2 al posto di usare il process sensibile al clock utilizzo una wait on clock:

1. process(clk) -- state Register

begin

if clk='1' then -- rising edge of clock

state <= nextstate;

end if;

end process;

1. wait on clk, x;

if (clk'event and clk = '1') then

state <= nextstate;

wait for 0 ns; -- wait for state to be updated

#### Ex\_1 stimuli\_1 e Ex\_2 stimuli\_1

Come si può vedere dalle simulazioni il comportamento per ex\_1 e ex\_2 è uguale.

#### Ex\_1 stimuli\_2 e Ex\_2 stimuli\_2

Come si può vedere dalle simulazioni il comportamento per ex\_1 e ex\_2 è uguale.

#### Ex\_1 stimuli\_3 e Ex\_2 stimuli\_3

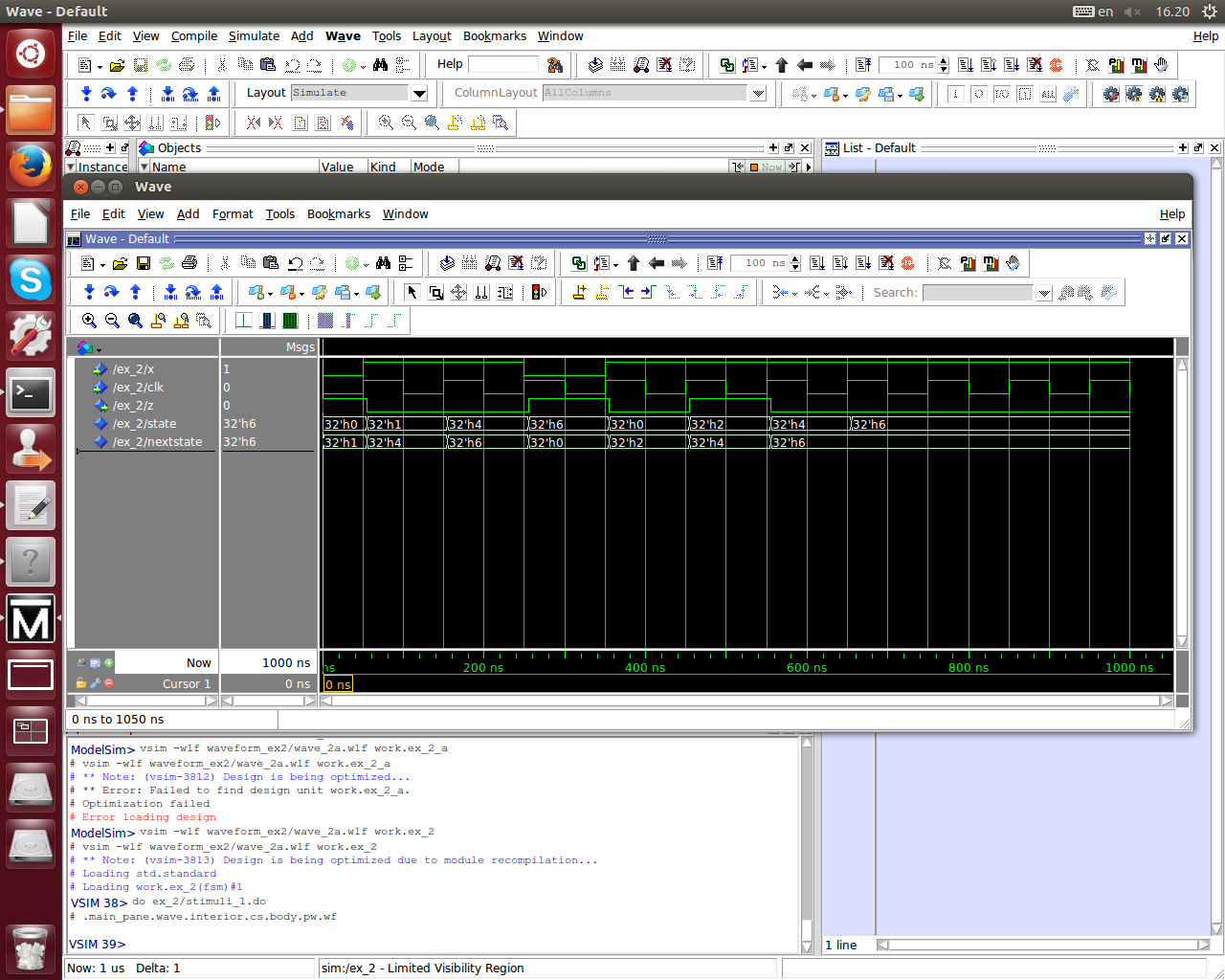
Come si può vedere dalle simulazioni il comportamento per ex\_1 e ex\_2 è uguale.

## Ex\_3

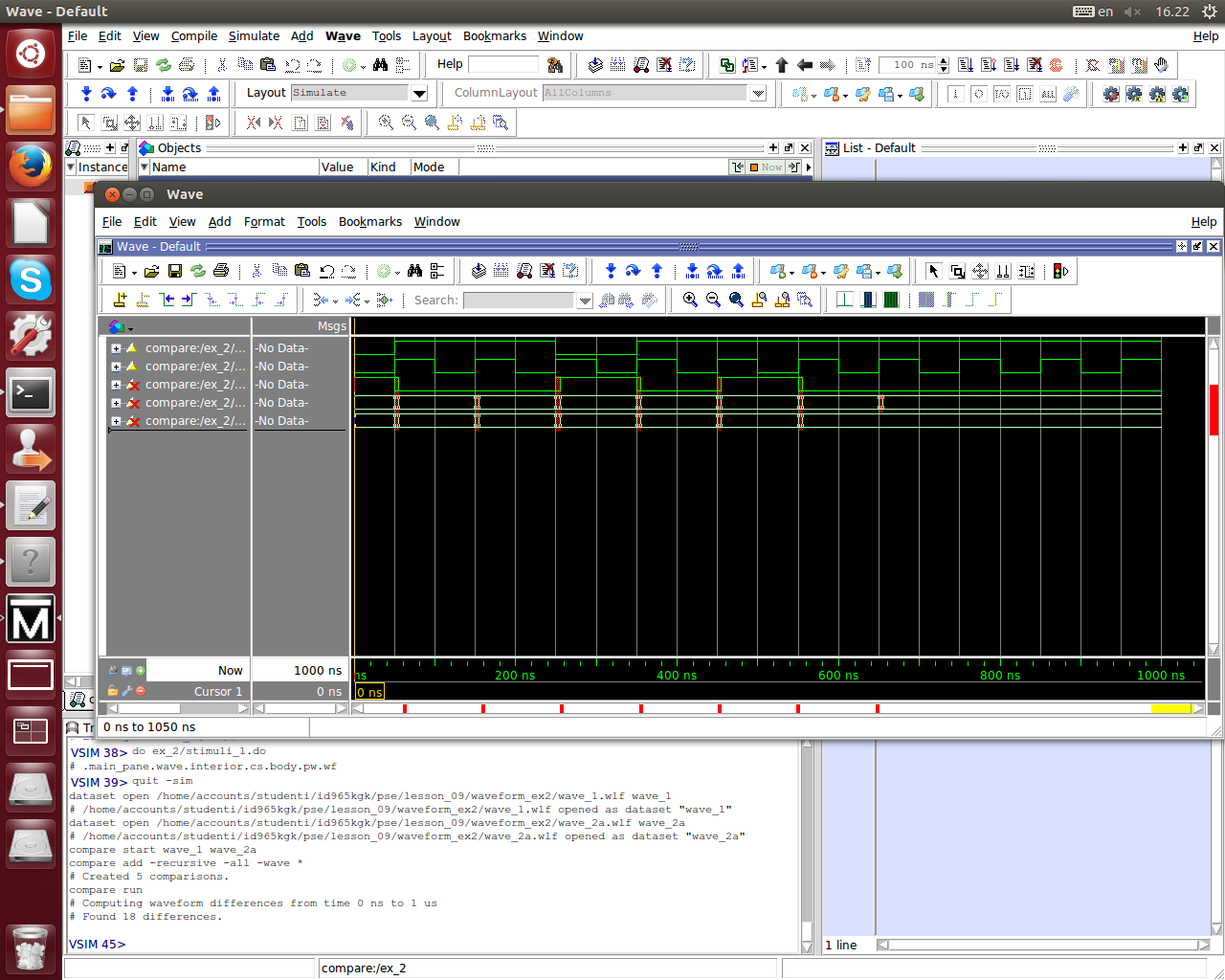
Nell’esercizio 3 devo considerare le varie versioni dell’esercizio 2:

### Versione a)

Questo è il risultato della versione a) simulato con il stimuli\_1.



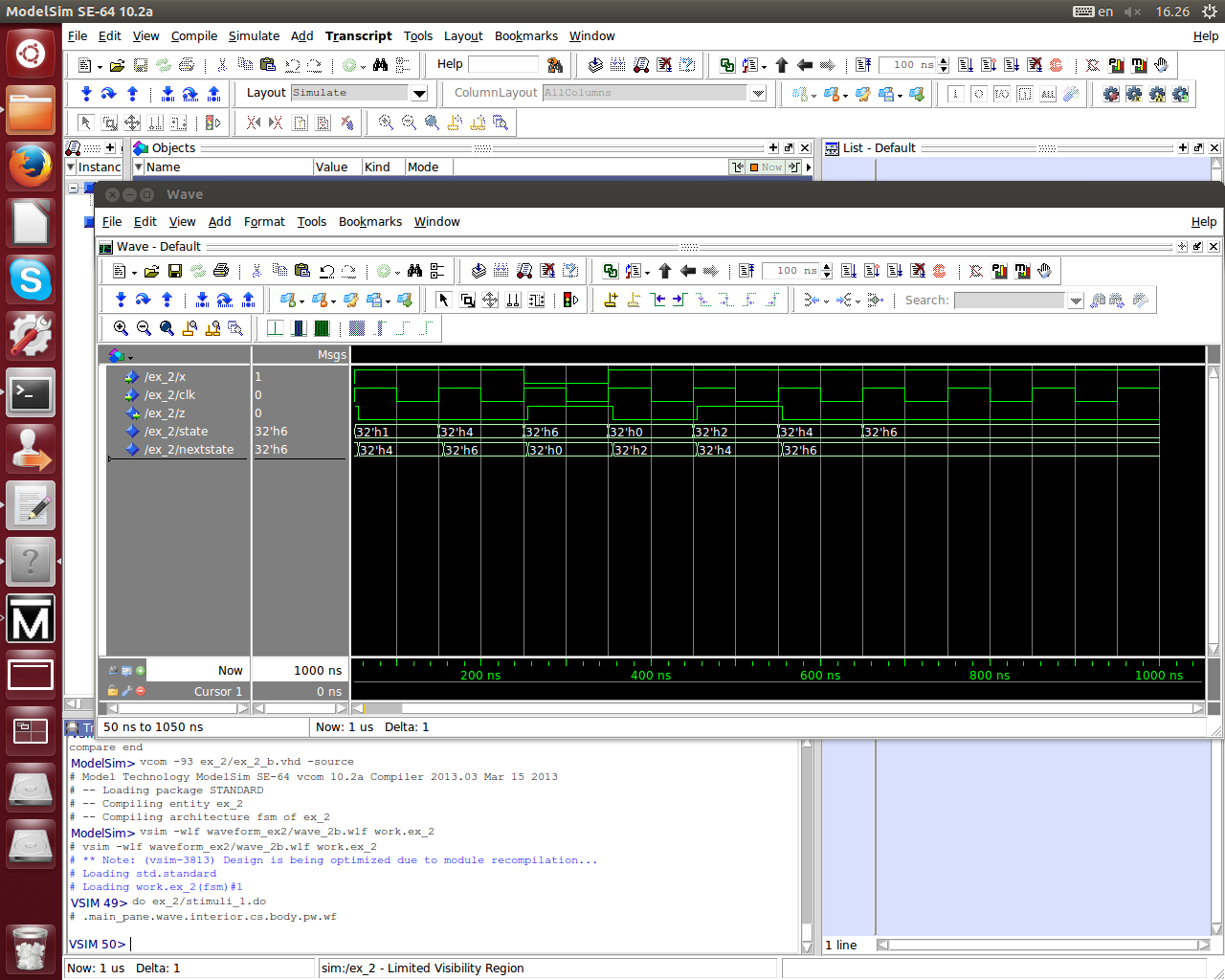
#### Differenza con ex\_2

Questa è la differenza tra l’esercizio\_2 simulato con lo stimuli\_1 e la versione a) dell’esercizio\_2 con lo stesso simuli:

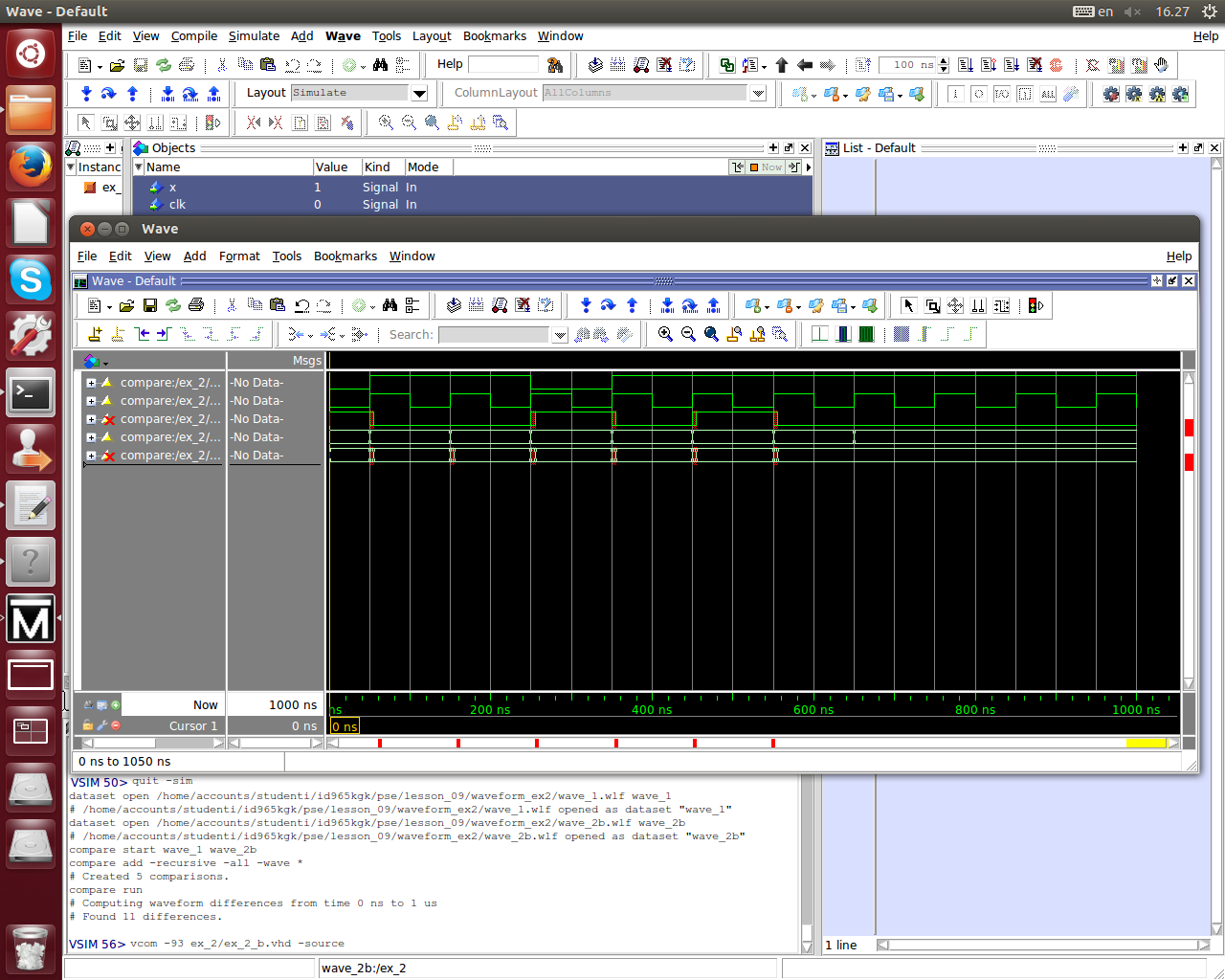
Come si può notare la variazione è solo nei segnali nextstate, z che vengono ritardati 5ns. Questo ritardo è dovuto al wait per aggiornare lo stato che da 0 ns in questa versione viene assegnato a 5ns. L’altra modifica fatta in questa versione non ha influenza sui tempi di x e neststate come si può vedere nella versione b) ma ha influenza sui nanosecondi in cui viene aggiornato state che subiscono 5 ns di ritardo.

### Versione b)

Questo è il risultato della versione b) simulato con il stimuli\_1, come si può notare è identica quella della versione a) tranne che per il segnale state.

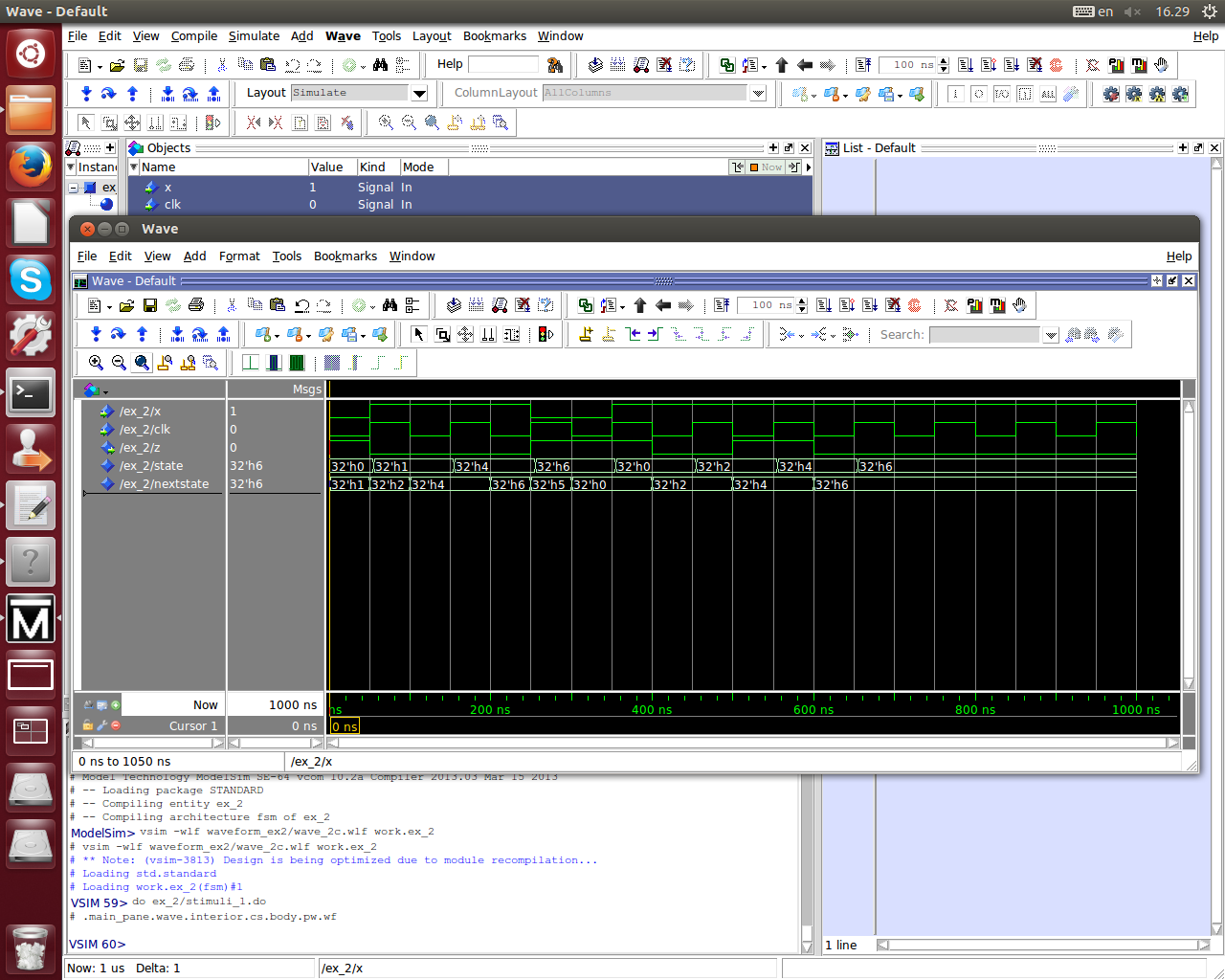


#### Differenza con ex\_2

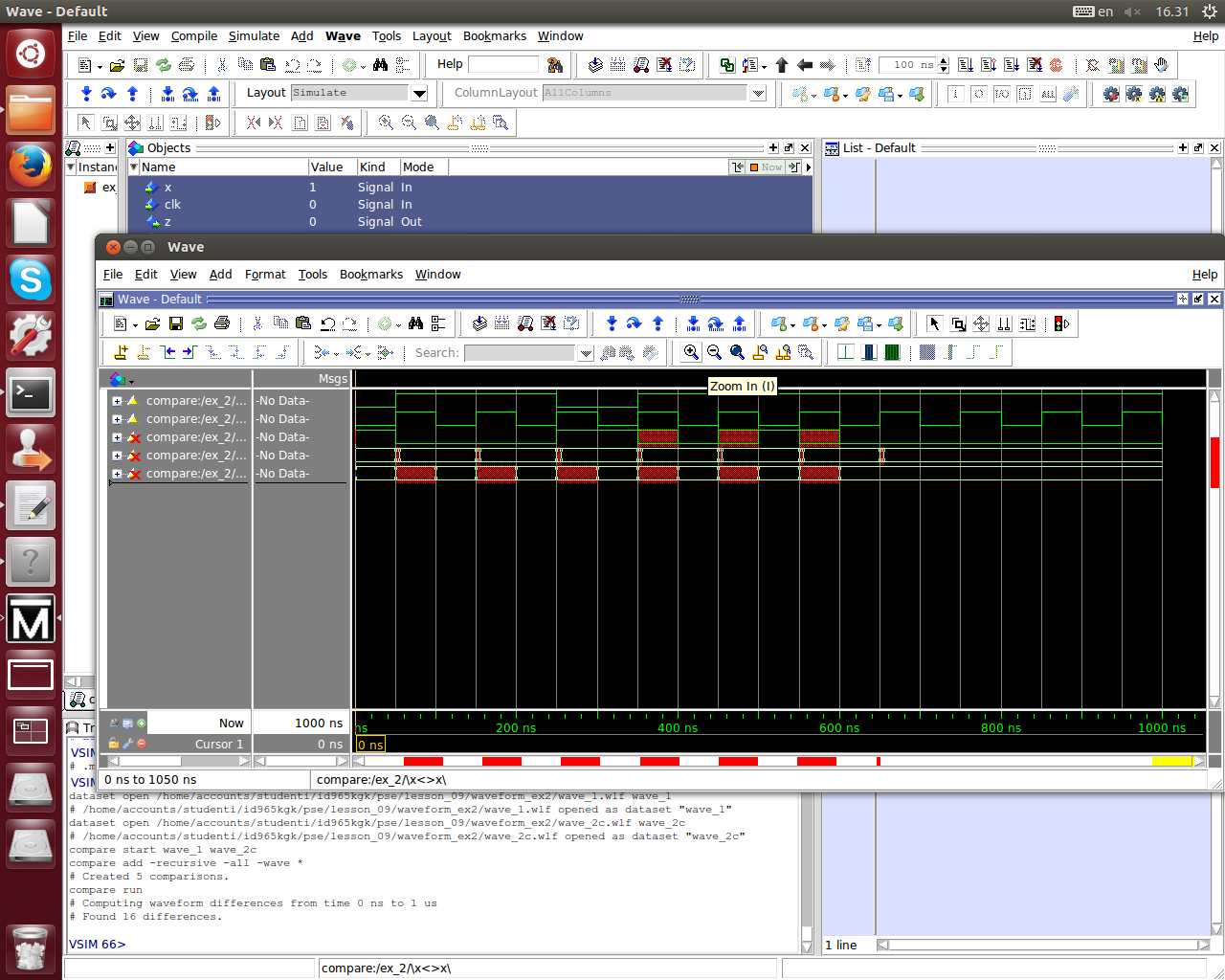
Questa è la differenza tra l’esercizio\_2 simulato con lo stimuli\_1 e la versione b) dell’esercizio\_2 con lo stesso simuli

Come si può notare la variazione è solo nei segnali z e nextstate che vengono ritardati 5ns. Questo ritardo è dovuto al wait per aggiornare lo stato che da 0 ns in questa versione viene assegnato a 5ns. Il segnale state non viene ritardato in quanto l’assegnamento a state è pressoché immediato.

### Versione c)

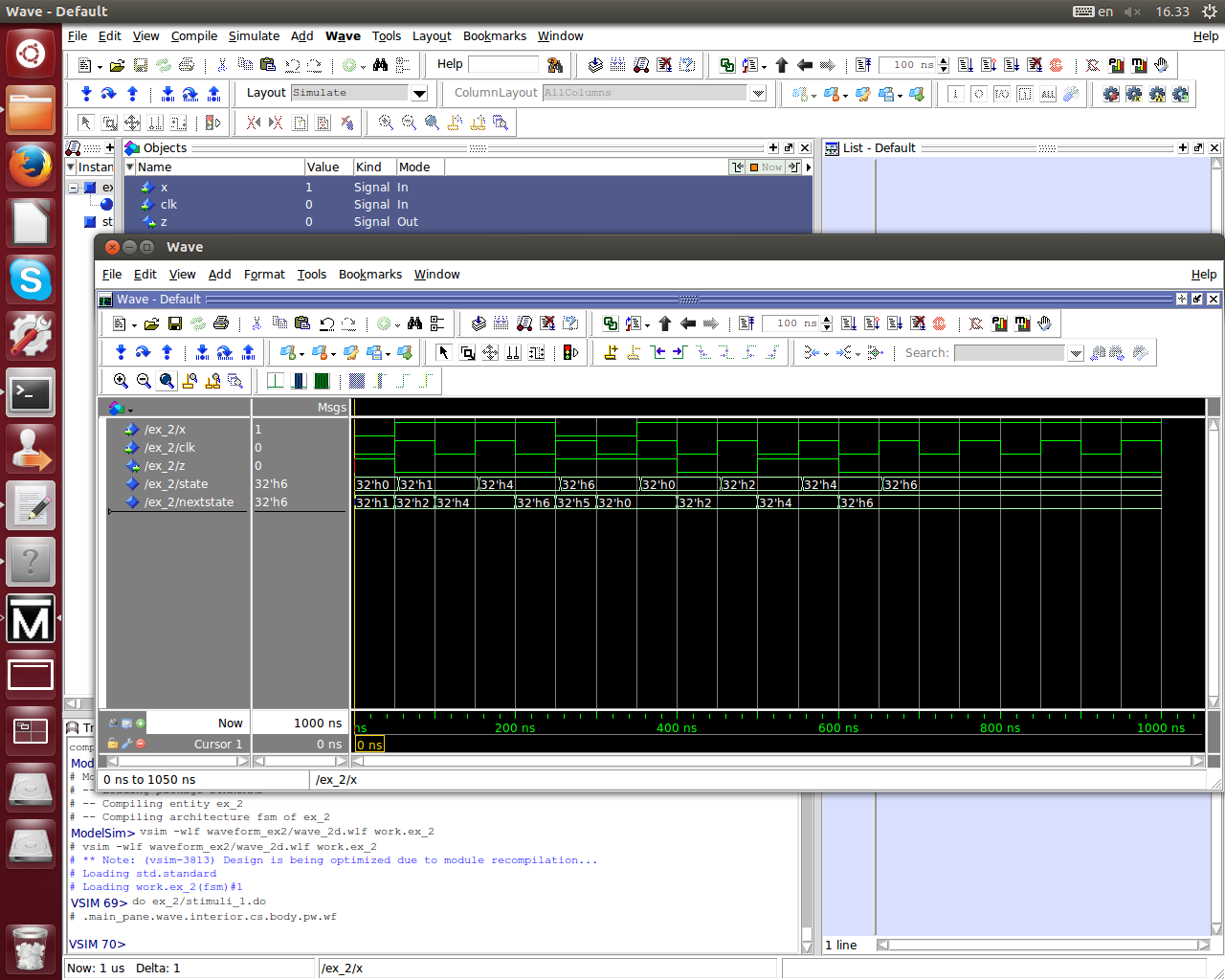
Questo è il risultato della versione c) simulato con il stimuli\_1:

#### Differenza con ex\_2

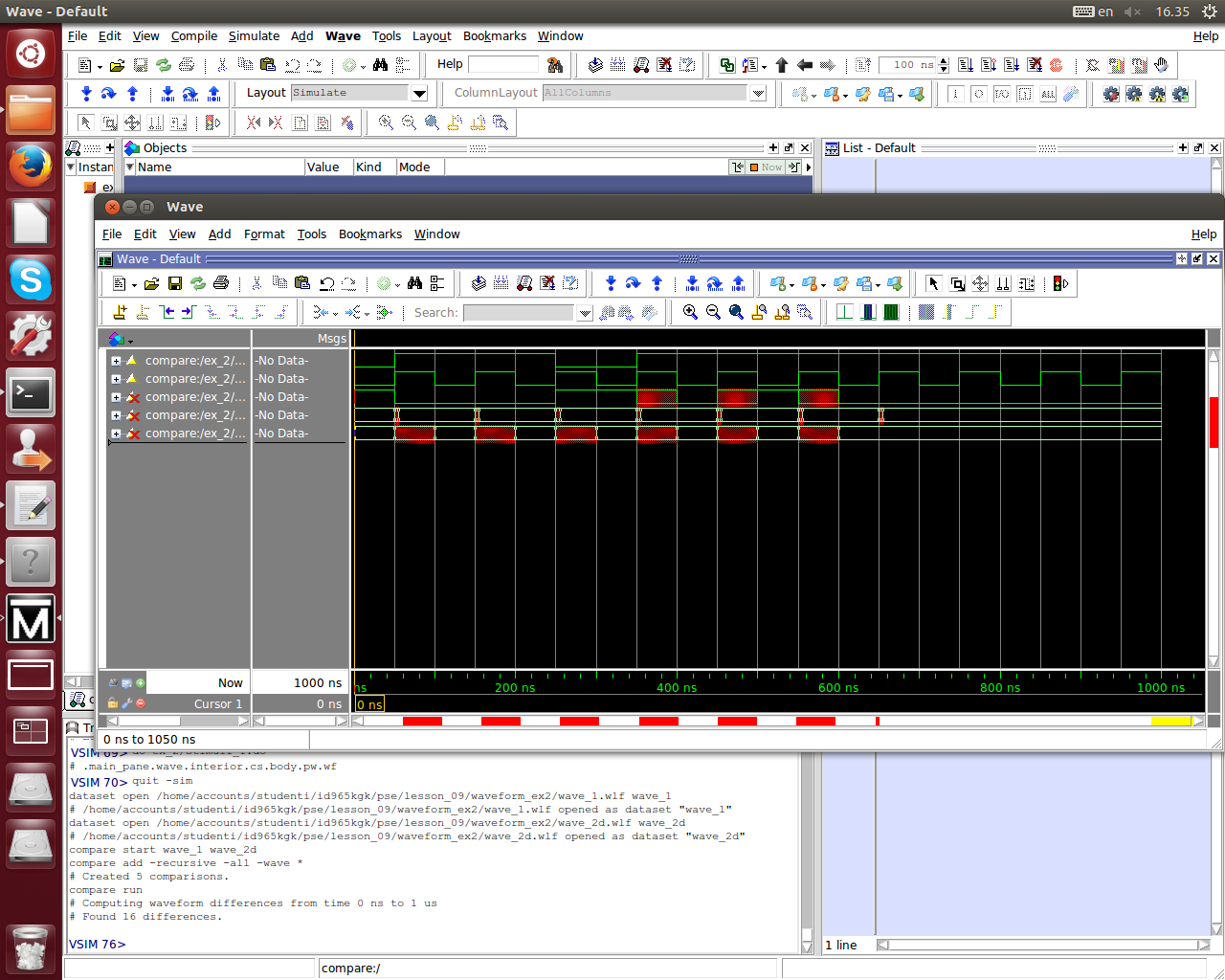
Questa è la differenza tra l’esercizio\_2 simulato con lo stimuli\_1 e la versione c) dell’esercizio\_2 con lo stesso simuli

Come si può notare la variazione è sui segnali z, state e nextstate. A causa di un ritardo di aggiornamento del segnale state di 5 ns il segnale nextstate ha una variazione di 50 ns lo stesso vale per z.

### Versione d)

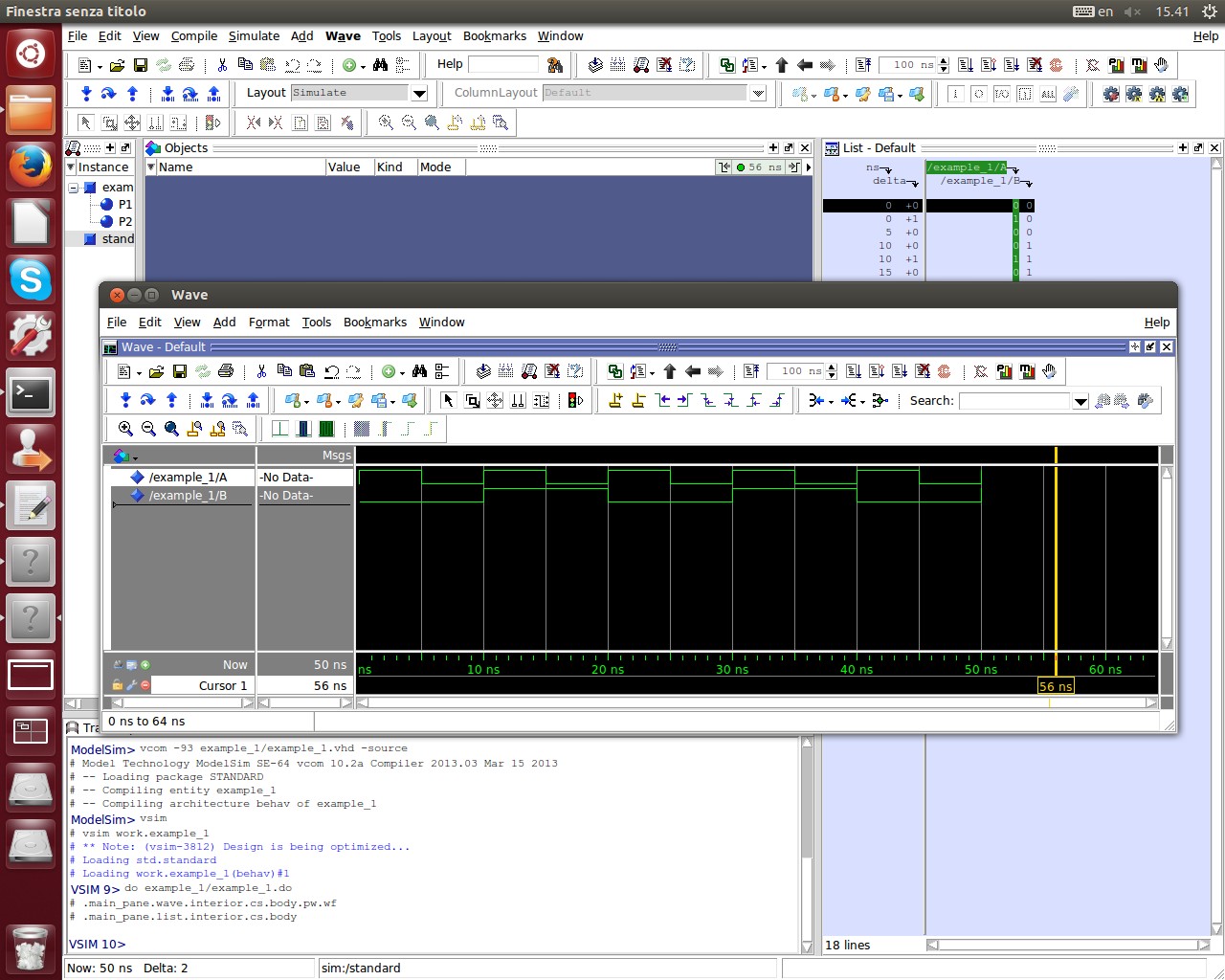
Questo è il risultato della versione d) simulato con il stimuli\_1:

#### Differenza con ex\_2

Questa è la differenza tra l’esercizio\_2 simulato con lo stimuli\_1 e la versione c) dell’esercizio\_2 con lo stesso simuli.

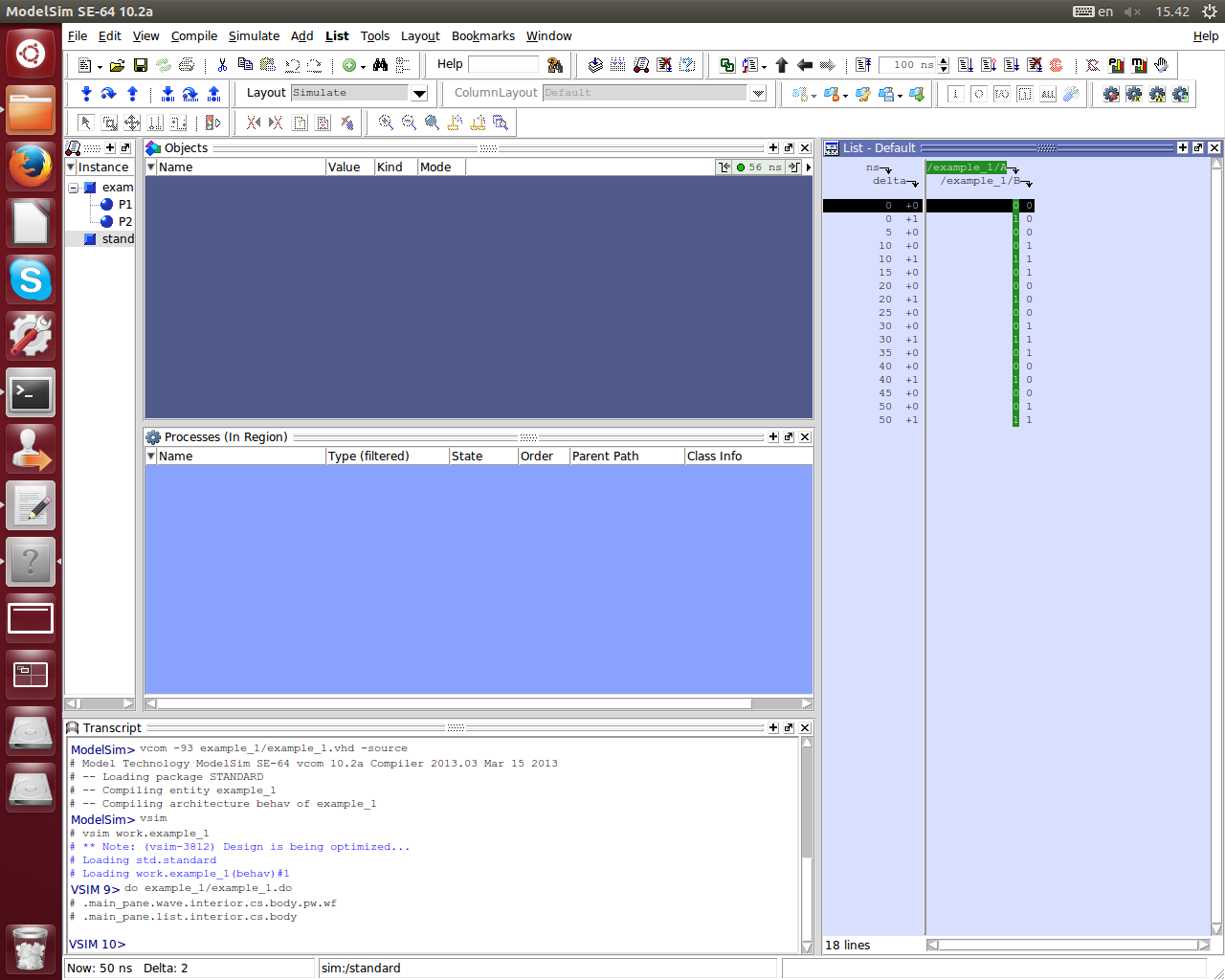
Come si può notare la variazione è sui segnali z, state e nextstate. A causa di un ritardo di aggiornamento del segnale state di 5 ns il segnale nextstate ha una variazione di 50 ns lo stesso vale per z. Ed è identica alla simulazione della versione d).

## Ex\_4



### Signal drivers

|  |  |  |
| --- | --- | --- |
|  |  | ‘0’ |
|  |  | ‘0’ |



@0-init

|  |  |  |
| --- | --- | --- |
| ‘0’@5 | ‘1’@∆ | ‘0’ |
|  |  | ‘0’ |

@0

|  |  |  |
| --- | --- | --- |
|  | ‘0’@5 | ‘1’ |
|  | ‘1’@10 | ‘0’ |

@∆

|  |  |  |
| --- | --- | --- |
|  |  | ‘0’ |
|  | ‘1’@10 | ‘0’ |

@5

|  |  |  |
| --- | --- | --- |
| ‘0’@15 | ‘1’@10+∆ | ‘0’ |
|  |  | ‘1’ |

@10

|  |  |  |
| --- | --- | --- |
|  | ‘0’@15 | ‘1’ |
|  | ‘0’@20 | ‘1’ |

@10+∆

|  |  |  |
| --- | --- | --- |
|  |  | ‘0’ |
|  | ‘0’@20 | ‘1’ |

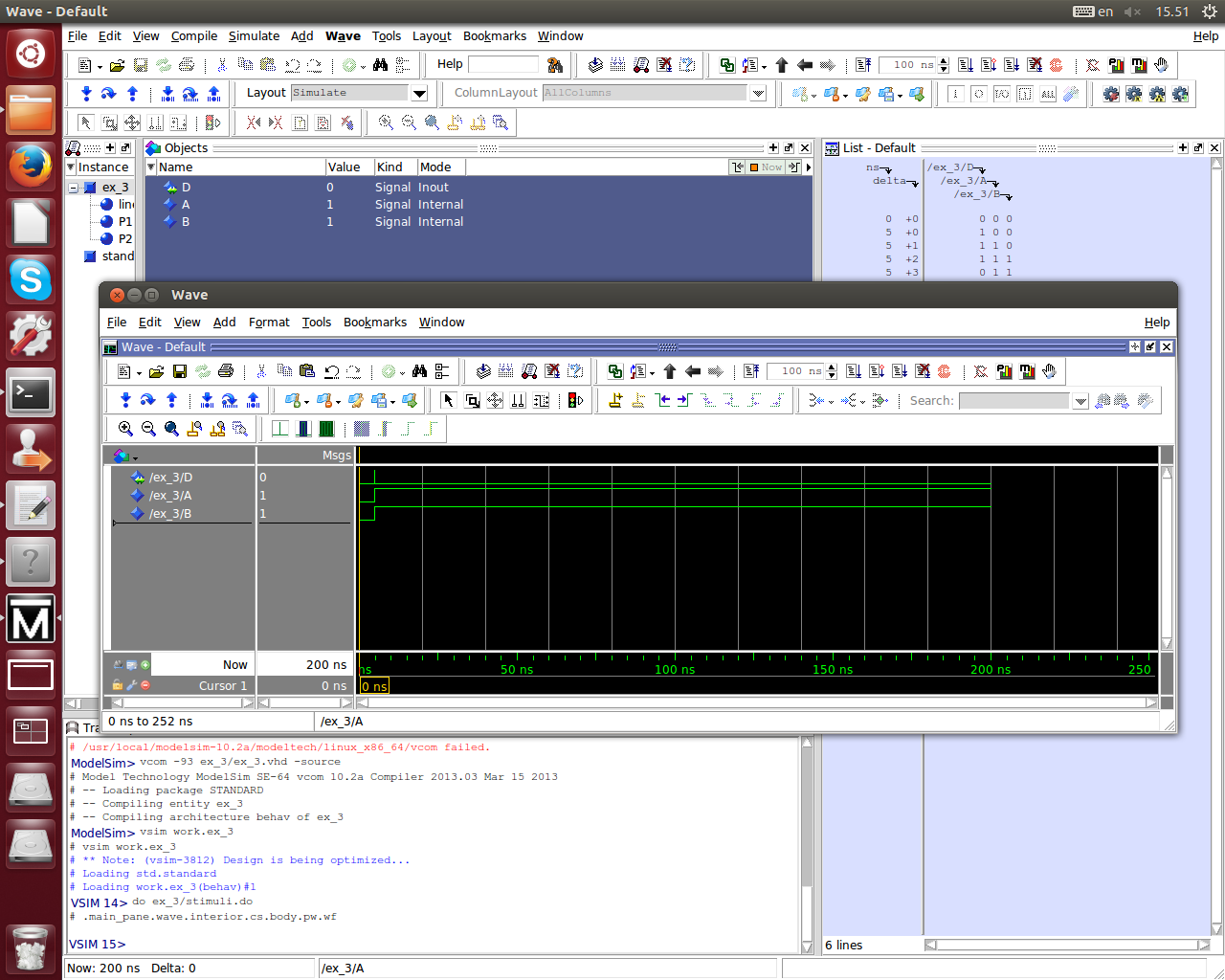
@15

|  |  |  |
| --- | --- | --- |
| ‘0’@25 | ‘1’@20+∆ | ‘0’ |
|  |  | ‘0’ |

@20

Come si può notare il comportamento è ciclico.

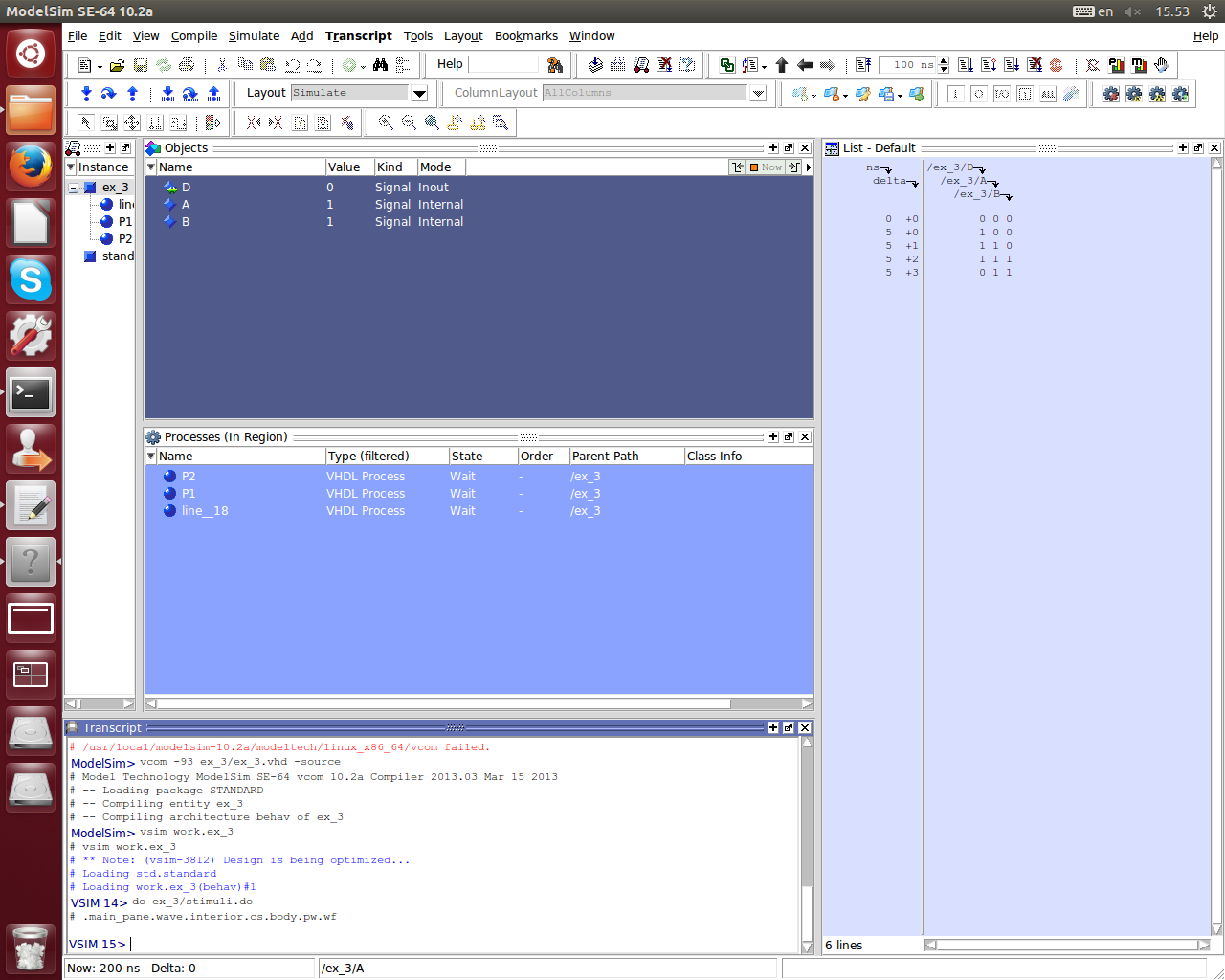
## Ex\_5



### Signal drivers

Considero i segnali A, B,C,D,E,F in questo ordine.

|  |  |  |
| --- | --- | --- |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |



@0-init

|  |  |  |
| --- | --- | --- |
|  | ‘0’@0+∆ | ‘0’ |
|  | ‘0’@0+∆ | ‘0’ |
|  | ‘0’@0+∆ | ‘0’ |
|  | ‘1’@5 | ‘0’ |
|  | ‘0’@0+∆ | ‘0’ |
|  | ‘0’@0+∆ | ‘0’ |

@0

|  |  |  |
| --- | --- | --- |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  | ‘1’@5 | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |

@0+∆

|  |  |  |
| --- | --- | --- |
|  | ‘1’@5+∆ | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘1’ |
|  |  | ‘0’ |
|  |  | ‘0’ |

@5

|  |  |  |
| --- | --- | --- |
|  |  | ‘1’ |
|  | ‘1’@5+2∆ | ‘0’ |
|  | ‘1’@0+2∆ | ‘0’ |
|  |  | ‘1’ |
|  |  | ‘0’ |
|  |  | ‘0’ |

@5+∆

@5+∆

|  |  |  |
| --- | --- | --- |
|  | ‘1’@5+3∆ | ‘1’ |
|  |  | ‘1’ |
|  |  | ‘1’ |
|  | ‘0’@5+3∆ | ‘1’ |
|  | ‘1’@5+3∆ | ‘0’ |
|  | ‘0’@5+3∆ | ‘0’ |

@5+2∆

|  |  |  |
| --- | --- | --- |
|  | ‘1’@5+4∆ | ‘1’ |
|  |  | ‘1’ |
|  |  | ‘1’ |
|  |  | ‘0’ |
|  |  | ‘1’ |
|  |  | ‘0’ |

@5+3∆

|  |  |  |
| --- | --- | --- |
|  |  | ‘1’ |
|  |  | ‘1’ |
|  |  | ‘1’ |
|  |  | ‘0’ |
|  |  | ‘1’ |
|  |  | ‘0’ |

@5+4∆

Il segnale da qui non cambia più valore.

## Ex\_6

Considero i segnali A,B,C,D,E in questo ordine.

|  |  |  |
| --- | --- | --- |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |

@0

|  |  |  |
| --- | --- | --- |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘0’ |
|  | ‘1’@10 | ‘0’ |

@0

|  |  |  |
| --- | --- | --- |
|  | ‘1’@15 | ‘0’ |
|  | ‘1’@10+∆ | ‘0’ |
|  | ‘0’@20 | ‘0’ |
|  |  | ‘0’ |
|  |  | ‘1’ |

@10

|  |  |  |
| --- | --- | --- |
|  | ‘5’@25 | ‘0’ |
|  | ‘8’@10+2∆ | ‘1’ |
|  | ‘0’@20 | ‘0’ |
|  | ‘1’@13 | ‘1’ |
|  |  | ‘1’ |

@10+∆

Essendo un assegnamento inerziale cancella assegnamenti precedenti. Tolgo quindi ‘1’@15.

|  |  |  |
| --- | --- | --- |
|  | ‘5’@25 | ‘0’ |
|  |  | ‘8’ |
|  | ‘0’@20 | ‘0’ |
|  | ‘1’@13 | ‘1’ |
|  |  | ‘1’ |

@10+2∆

|  |  |  |
| --- | --- | --- |
|  | ‘5’@25 | ‘0’ |
|  |  | ‘8’ |
|  | ‘0’@20 | ‘0’ |
|  |  | ‘1’ |
|  |  | ‘1’ |

@13

|  |  |  |
| --- | --- | --- |
|  | ‘5’@25 | ‘1’ |
|  |  | ‘8’ |
|  |  | ‘1’ |
|  |  | ‘1’ |
|  |  | ‘1’ |

@20

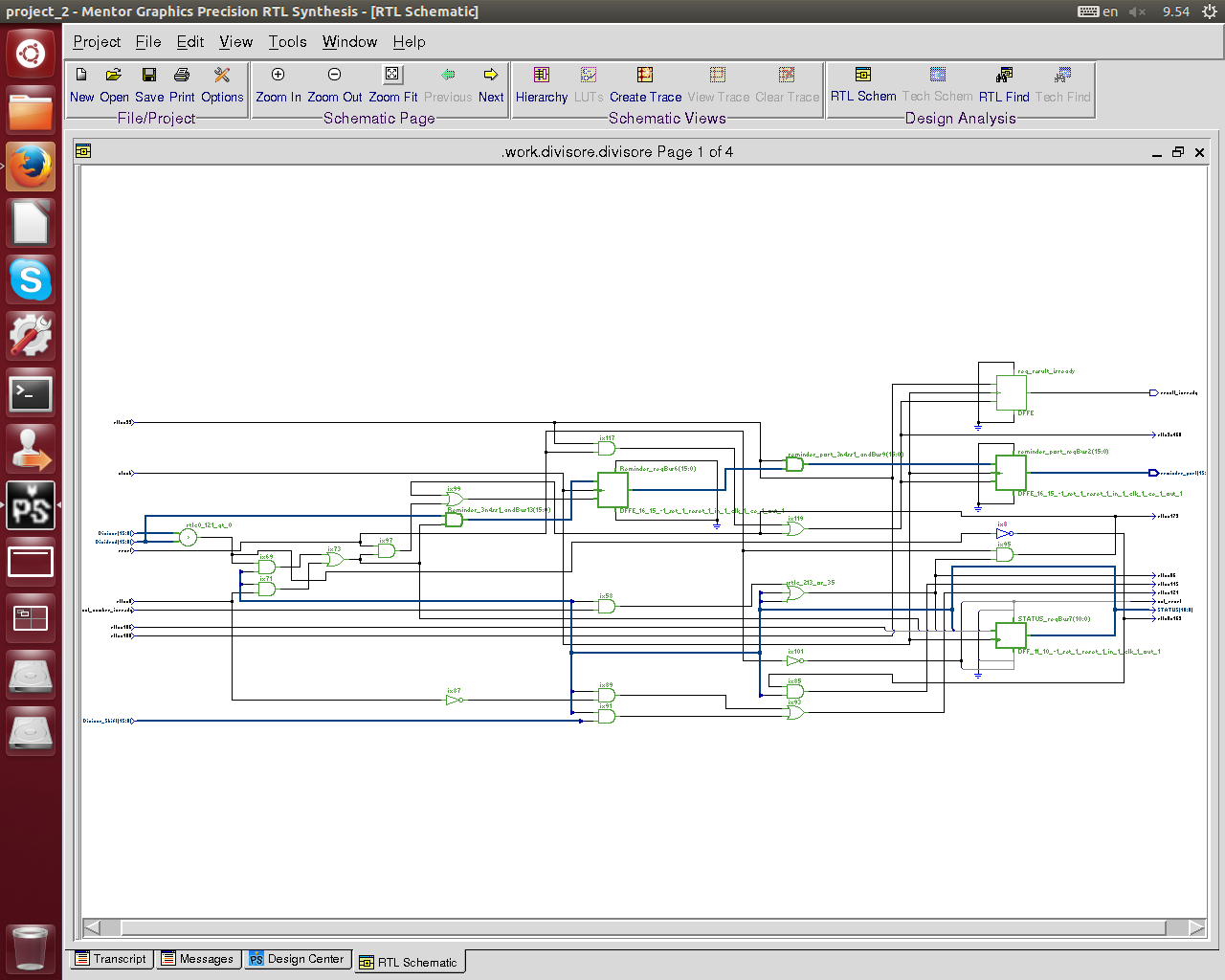
|  |  |  |
| --- | --- | --- |
|  |  | ‘5’ |
| @25 |  | ‘8’ |
|  |  | ‘1’ |
|  |  | ‘1’ |
|  |  | ‘1’ |

Sintesi automatica del codice VHDL a livello RTL

# Lezione 12

L’obiettivo di questa lezione è sintetizzare il modulo Divider per la tecnologa FPGA con PrecisionRTL.

## Compilazione e sintesi

Utilizzando il tool di sintesi ho sintetizzato il mio circuito Divider seguendo il procedimento indicato. Ho creato un nuovo progetto al quale ho aggiunto i file divider.vhdl. Ho sintetizzato il io design per spartan6 FPGA come richiesto. Compilando ho ottenuto il seguente RTL Schema:

a questo punto il mio design è pronto per la fase di sintesi. Dopo la sintesi posso valutare la qualità del risultato.

## Area

Il report dell’area è il seguente:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Device Utilization for 6SLX16CSG324

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Resource Used Avail Utilization

---------------------------------------------------------------

IOS 67 232 28.88%

Global Buffers 1 16 6.25%

LUTs 139 9112 1.53%

CLB Slices 35 2278 1.54%

Dffs or Latches 108 18224 0.59%

Block RAMs 0 32 0.00%

RAMB8BWER 0 - -

RAMB16BWER 0 - -

DSP48A1s 2 32 6.25%

---------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Library: work Cell: divisore View: divisore

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Cell Library References Total Area

BUFGP xis6 1 x

DSP48A1 xis6 2 x 1 2 DSP48A1s

FDC xis6 10 x 1 10 Dffs or Latches

FDE xis6 97 x 1 97 Dffs or Latches

FDP xis6 1 x 1 1 Dffs or Latches

GND xis6 1 x

IBUF xis6 34 x

INV xis6 1 x

LUT2 xis6 19 x 1 19 LUTs

LUT3 xis6 16 x 1 16 LUTs

LUT4 xis6 8 x 1 8 LUTs

LUT5 xis6 20 x 1 20 LUTs

LUT6 xis6 71 x 1 71 LUTs

LUT6\_L xis6 5 x 1 5 LUTs

MUXCY xis6 2 x 1 2 MUX CARRYs

MUXCY\_L xis6 20 x 1 20 MUX CARRYs

OBUF xis6 33 x

VCC xis6 1 x

XORCY xis6 16 x

Number of ports : 68

Number of nets : 408

Number of instances : 358

Number of references to this view : 0

Total accumulated area :

Number of DSP48A1s : 2

Number of Dffs or Latches : 108

Number of LUTs : 139

Number of MUX CARRYs : 22

Number of gates : 141

Number of accumulated instances : 358

## FSM

Il risultato per l’EFSM è il seguente:

FSM EXTRACTION ANALYSIS

=======================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Module : work.divisore(divisore)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Number of FSMs Extracted: 1

1.

--------------------------------------------------

Number of States : 11

Primary State Variable : STATUS[3:0]

Other State Variable(s) : 1) NEXT\_STATUS[3:0]

Async set/reset state(s) : 0000

Re-encoding Scheme : ONEHOT

FSM: Info, state encoding table:

FSM: Index Literal Encoding

FSM: 0 0000 00000000001

FSM: 1 0001 00000000010

FSM: 2 0010 00000000100

FSM: 3 0011 00000001000

FSM: 4 0100 00000010000

FSM: 5 0101 00000100000

FSM: 6 0110 00001000000

FSM: 7 0111 00010000000

FSM: 8 1000 00100000000

FSM: 9 1001 01000000000

FSM: 10 1010 10000000000