# Introduzione alle FPGA mediante esempi

25 Ottobre 2014

**Sommario**

L'obiettivo è realizzare semplici programmi per introdurre e spiegare alcune problematiche tipiche delle FPGA. Il linguaggio di programmazione scelto per gli esempi è il Verilog HDL; l'ambiente di sviluppo è ALTERA Quartus II Web edition ver. 13.0.1 per Linux e il target è la scheda TerAsic DE0-Nano.

Gli argomenti trattati mirano a mostrare le problematiche relative al parallelismo, al clocking e alla simulazione RTL e Gate-Level dei circuiti.

La scelta delle FPGA ALTERA è dettata semplicemente dal fatto che i tool di sviluppo sono identici tra le versioni Linux e Windows oltre al fatto che la scheda di sviluppo fornita da TerAsic è probabilmente la meno costosa al momento in cui si scrive. In ultimo, non per importanza, vi è il fatto che ALTERA fornisce una vastissima quanto completa documentazione. A parte questo, non verranno fatte altre considerazioni di merito sulla qualità dei componenti; anzi si invita chi legge ad informarsi relativamente alle soluzioni dei competitor come XILINX, Lactice, ecc per capire quale casa offre le soluzioni migliori per i propri progetti.

Questo documento non è una guida ne completa ne esaustiva dei temi e dei linguaggi trattati, ma è solo un'introduzione all'argomento orientata agli studenti dell'ITIS E. Fermi di Mantova. Questo documento e il software descritto sono liberamente scaricabili dal sito dell'associazione LUGMan www.lugman.org.

## 1 Introduzione all'ambiente di sviluppo

Quartus II è l'IDE di sviluppo per dispositivi ALTERA scaricabile gratuitamente (previa registrazione) dal sito www.altera.com. La versione Web è quella completamente gratuita che mette a disposizione comunque un certo numero di IP[1] pronte all'uso che vanno dai PLL ai controller per le ram DDR.

Quartus II avviato si presenta come in figura 1.1. Di seguito verrà mostrato come creare un progetto per l'FPGA montata su DE0-Nano, ossia la Cyclon IV E EP4CE22F17C6.

Per creare un nuovo progetto accedere a File▷New Project Wizzard e apparirà un form di introduzione che si potrà evitare venga mostrato nuovamente; ad ogni modo cliccando su "Next" comparirà il form in figura 1.2; a questo punto occorre definire una directory dove mettere il progetto (Quartus II non la crea in automatico), definire un nome del progetto (es: prova); Quartus II automaticamente darà lo setesso nome anche alla "top-level design entry"[2], ossia l'ultimo campo. Si suggerisce di mettere un altro nome per esempio "main", in analogia con il linguaggio C, oppure aggiungere "_top" come suggeriscono i manuali ALTERA. Attenzione perchè questo campo darà il nome al file principale da cui cominciare a scrivere. Proseguire con Next dove sarà possibile inserire dei files al progetto. In questo caso non ce ne sono quindi si prosegue con Next.

Apparirà quindi il 3° punto (figura 1.3) che è quello più importante perchè permette di definire il device target sul quale occorrerà scaricare il progetto. Qui si possono fare due cose:

1. Proseguire senza indicare nulla: se si clicca su Next non c'è nessun problema, anzi in alcuni casi è anche meglio in quanto se si è in fase di sviluppo e non si sa quale FPGA può contenere il progetto, basta scegliere la famiglia nel primo menù a tendina e automaticamente Quartus II sceglierà una FPGA ad ogni compilazione del progetto.

2. Scegliere il dispositivo: se si conosce già il device di sviluppo è meglio selezionarlo in modo che il compilatore sappia già tutte le risorse a disposizione del progetto prima di compilare. Questo crea anche meno problemi in fase di assegnazione dei pin che deve essere sempre fatta per poter simulare il progetto.

Nel nostro esempio il sistema va settato come in figura 1.3:

[1] Intellectual Property

[2] La top-level entry è di fatto il primo modulo che il compilatore implementerà per poter costruire il circuito. In altre parole è "l'inizio del circito".

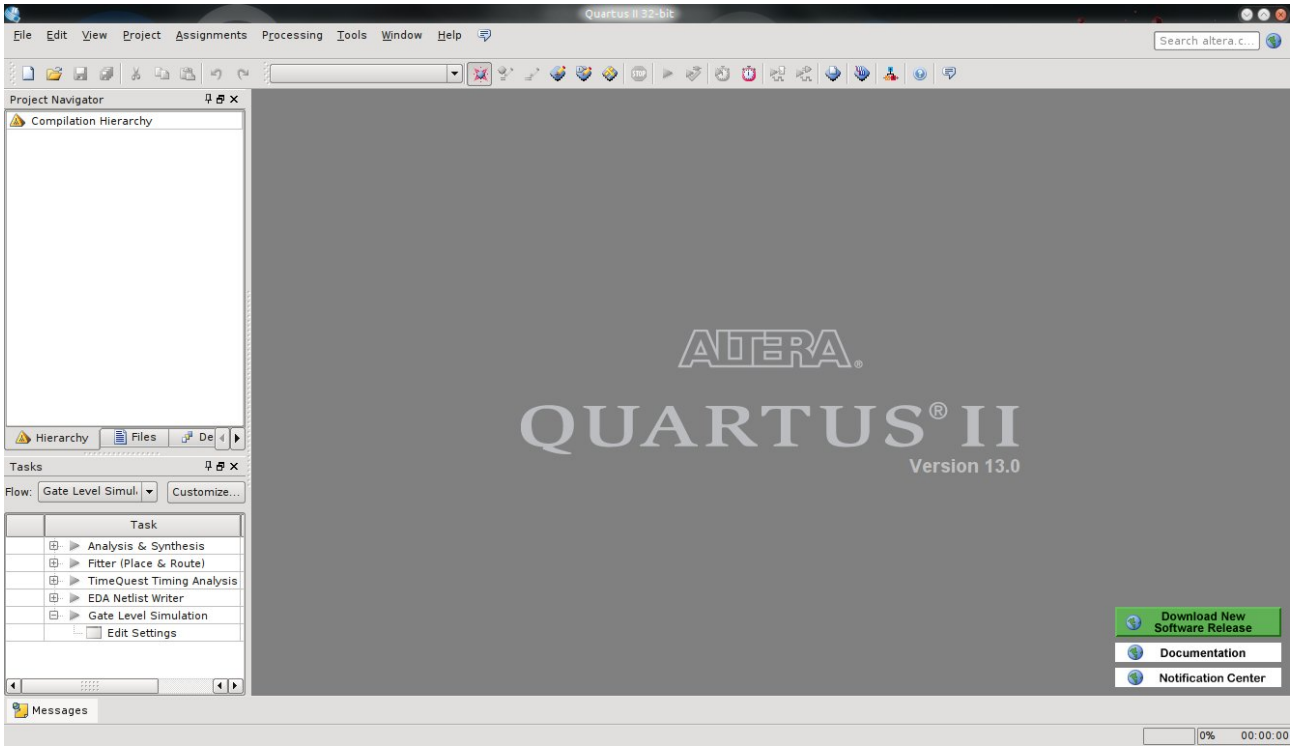



Figura 1.1: Quartus II 13.0 Web Edition 64 bit per Linux

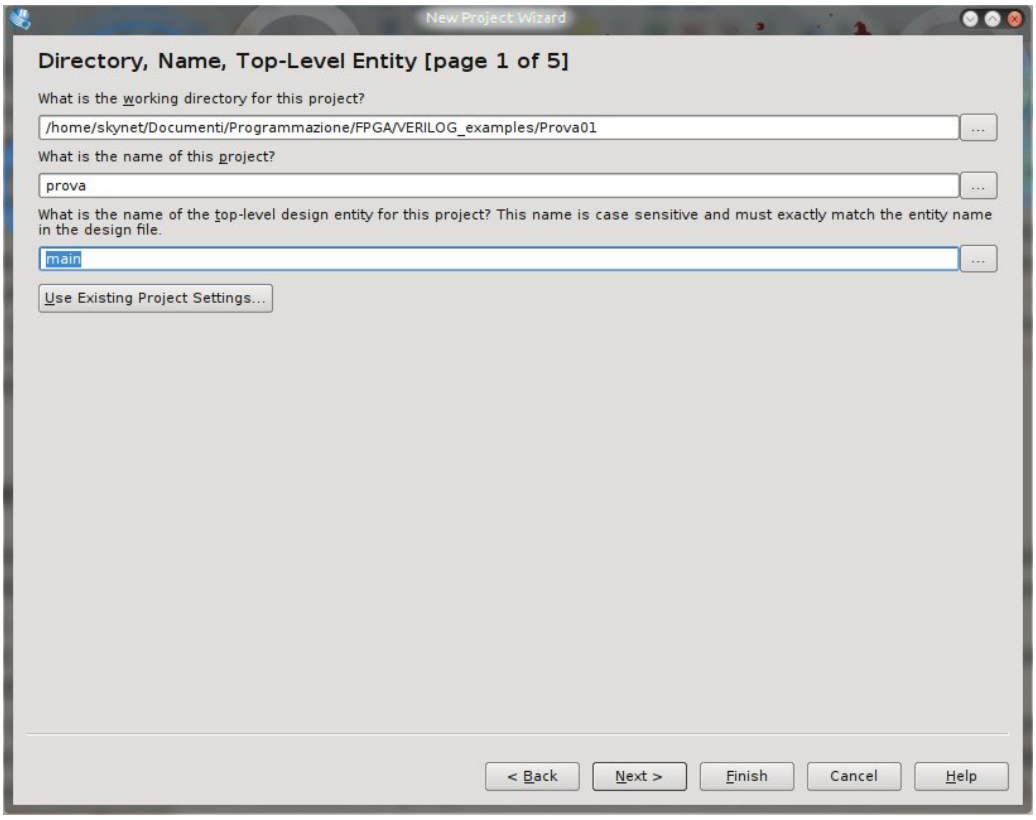



Figura 1.2: New Project Wizard

- Family: Cyclone IV E
- Selezionare l'FPGA EP4C22F17C6, aiutandosi eventualmente con il campo "Name filter"
- Avanzare con "Next"

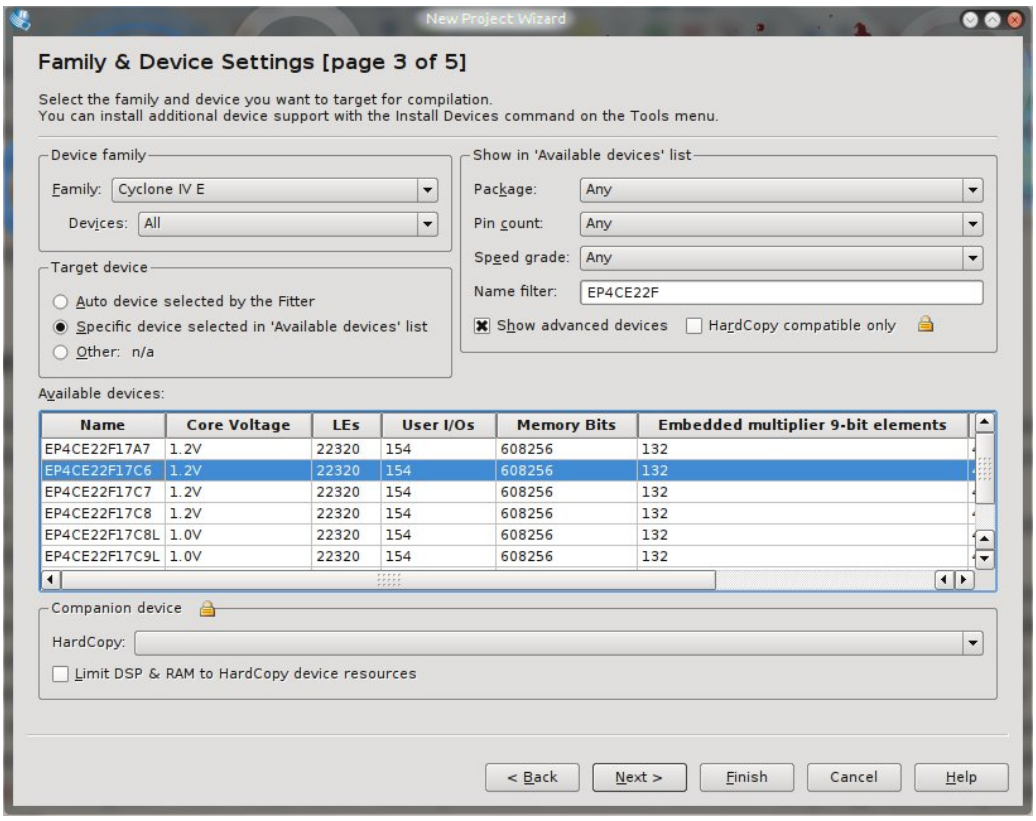



Figura 1.3: New Projec Wizard - Family&Device settings

A questo punto (passo 4 di 5) viene chiesto di indicare quali altri tool servono per la simulazione ed altre attività di sviluppo. Siccome verrà trattata solo la simulazione, quindi occorre sceglie il simulatore ModelSim-Altera, ossia la versione di ModelSim fornita da Altera. Esattamente come Quartus II è scaricabile gratuitamente. Siccome nel seguito verrà trattato il linguaggio Verilog HDL, occorrerà selezionarlo come linguaggio per la simulazione, come in figura 1.4.

Segue il punto 5/5 che non è altro che un riassunto; cliccando "Finish" il progetto è creato. A questo punto occorre definire il file principale. Si crea quindi un file cliccando su File ▷ New e comparirà il form in figura 1.5 da cui scegliere Verilog HDL.

In figura 1.6 è riportato un file *main.v*. Più avanti si entrerà nello specifico dei vari campi. Per ora si noti che la parte importante è la riga che indica il nome del modulo (*module main*) che deve essere appunto main. Quando il file verrà salvato, Quartus II proporrà automaticamente il nome "main.v". Questo progetto è già compilabile, ma si consiglia di settare nel riquadro *Task* in basso a sinistra il Flow "*Gate Level Simulation*" in modo da ottenere anche il codice per la simulazione.

## 1.1 Assegnazione dei pin

Il progetto appena creato non fa altro che accendere gli 8 led della scheda DE0-Nano trattandoli come una variabile ad 8 bit che si incrementa ad ogni colpo di clock.

Occore però istruire il compilatore per sapere a quali pin questi input e output sono connessi. Alla prima compilazione infatti, eseguibile con Processing ▷ Start Compilation, Quartus assegnerà di default dei pin durante il processo di "Fit" che si vede nel riquadro "Task" in basso a sinistra.

Vi sono due metodi per assegnare i pin:

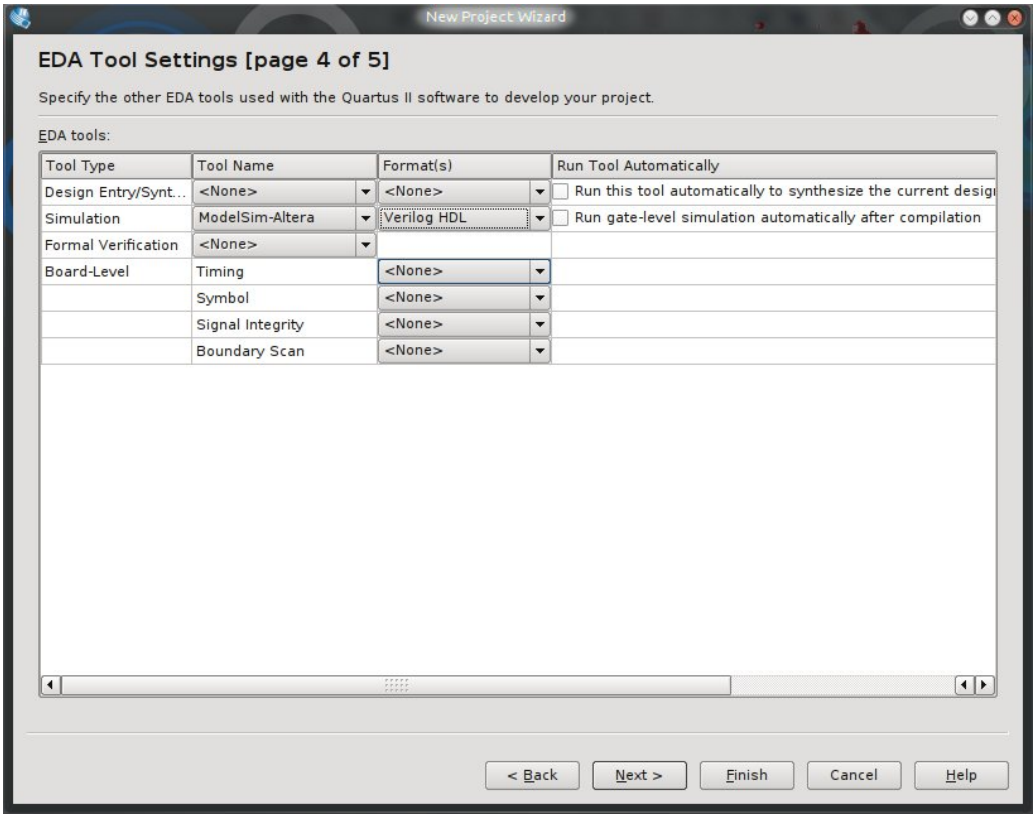



Figura 1.4: New Project Wizard - EDA Tool Settings

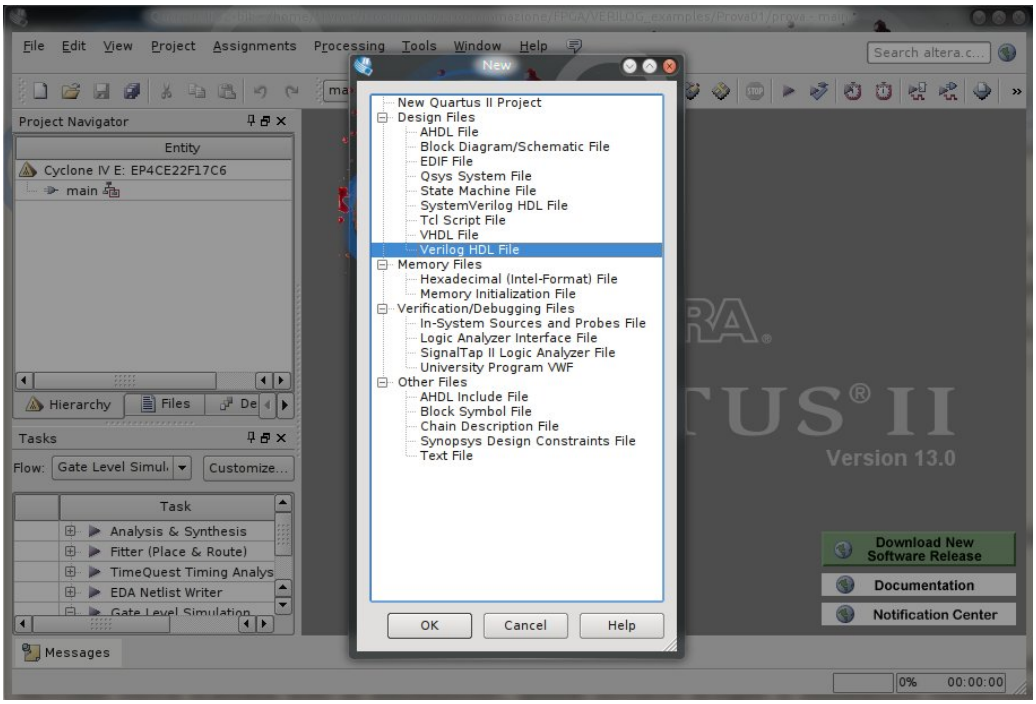



Figura 1.5: New File - scegliere Verilog HDL nella sezione *Design File*

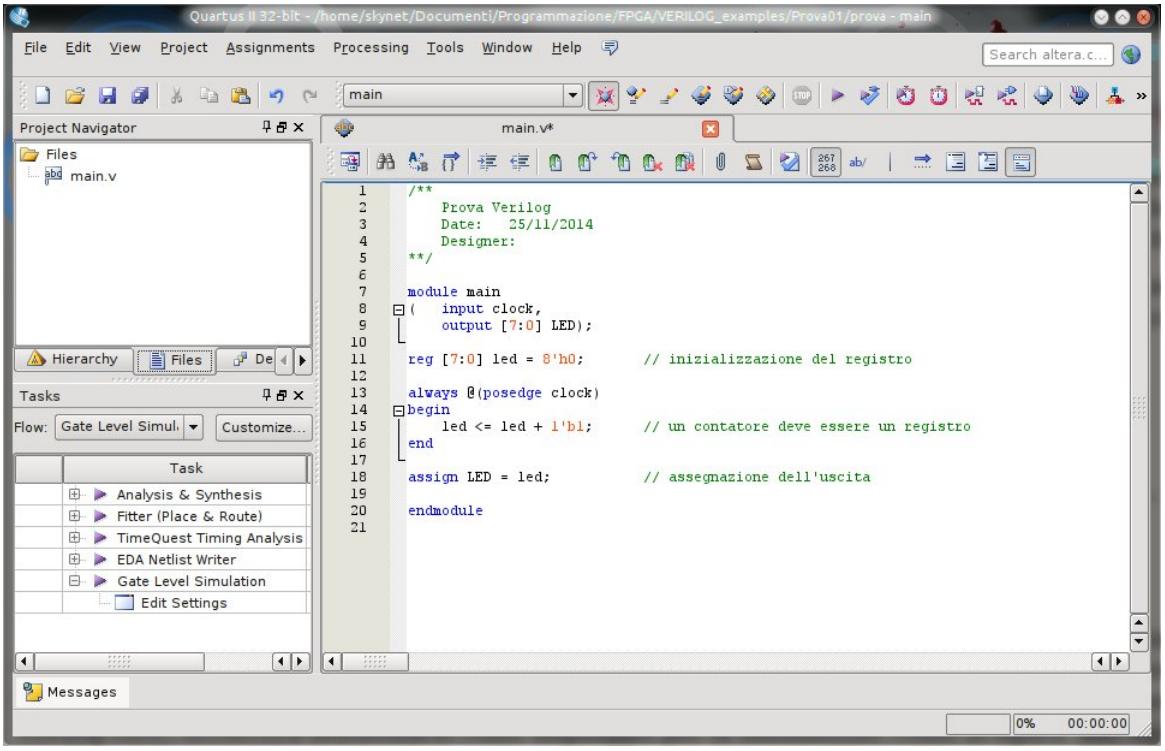



Figura 1.6: Esempio di progetto in Verilog - il modulo main è la top-level entry

**Pin Planer** si avvia con Assignments ▷ Pin Planner (figura 1.7) e permette di definire direttamente i pin già definiti come input e output nel modulo principale. Appena avviato il Pin Planner, si vede che vi sono già dei pin preassociati. Occorre ridefinirli leggendo dove sono associati i vari pin dal datasheet della scheda DE0-Nano come indicato in figura 1.7. I pin posizionati a mano compaiono come rossi sullo schema del chip.

**Pin Assignment** simile al Pin Planner, permette di settare qualsiasi pin con alcune varianti. Per esempio se non si sa ancora come e dove collegare i vari pin o se i pin di un modulo fossero più di quelli presenti sull'FPGA, è possibile per esempio definire i pin come virtuali. Ad ogni modo con il Pin Assignment occorre cercare con le opportune maschere. Il risultato è comunque uguale e si deve avere un risultato come quello riportato in 2.6.

Il progetto va quindi ricompilato una volta assegnati i pin.

## 1.2 RTL Viewer

Questo tool permette di visualizzare il circuito descritto dal linguaggio HDL scelto, ottimizzazioni comprese relative ai circuiti che non concorrono a generare output. Per visualizzare il circuito ocorre effettuare una compilazione (almeno la prima fase "*Analyss & Synthesis*") quindi invocare Tools ▷ Netlist Viewer ▷ RTL Viewer. Eseguendolo sull'esempio in figura 1.6 comparirà il circuito in figura 1.9.

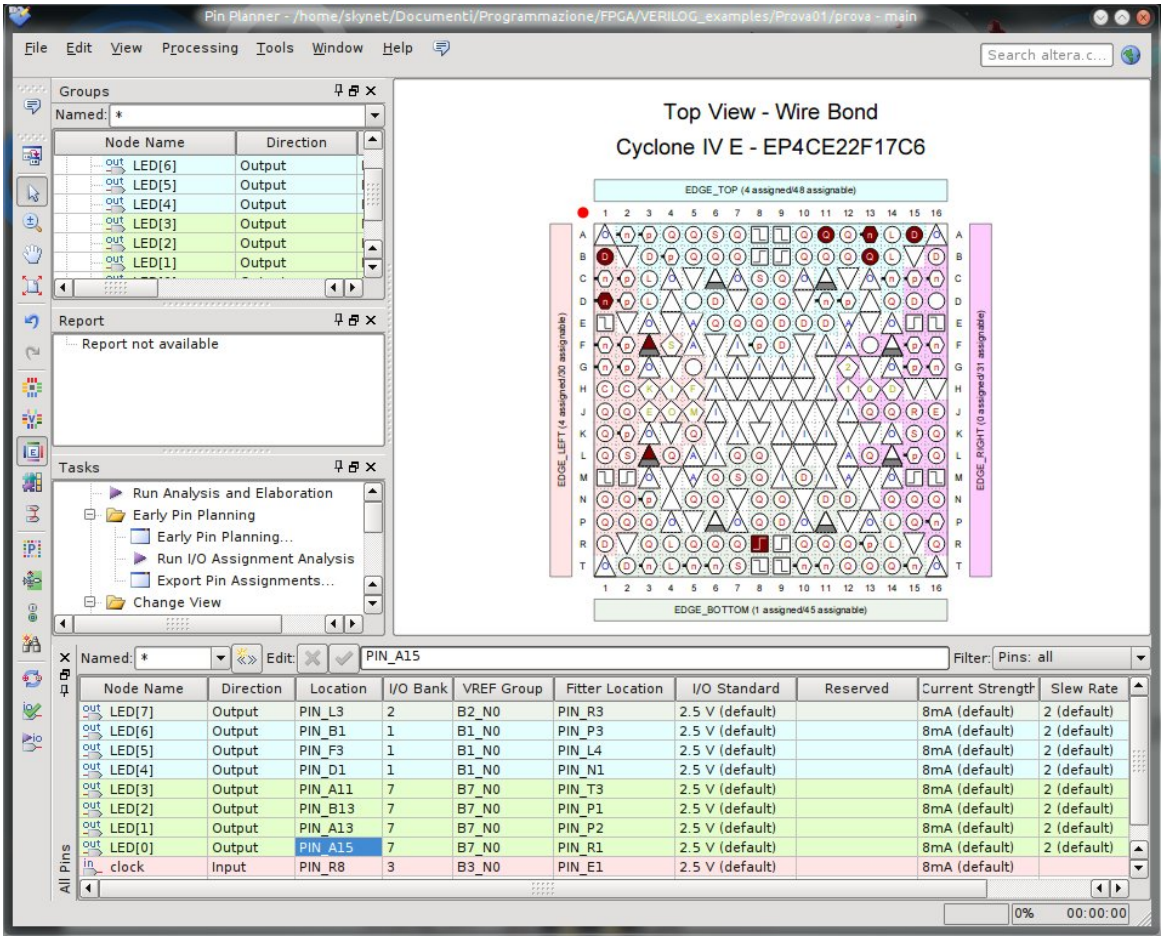



Figura 1.7: Pin Planner

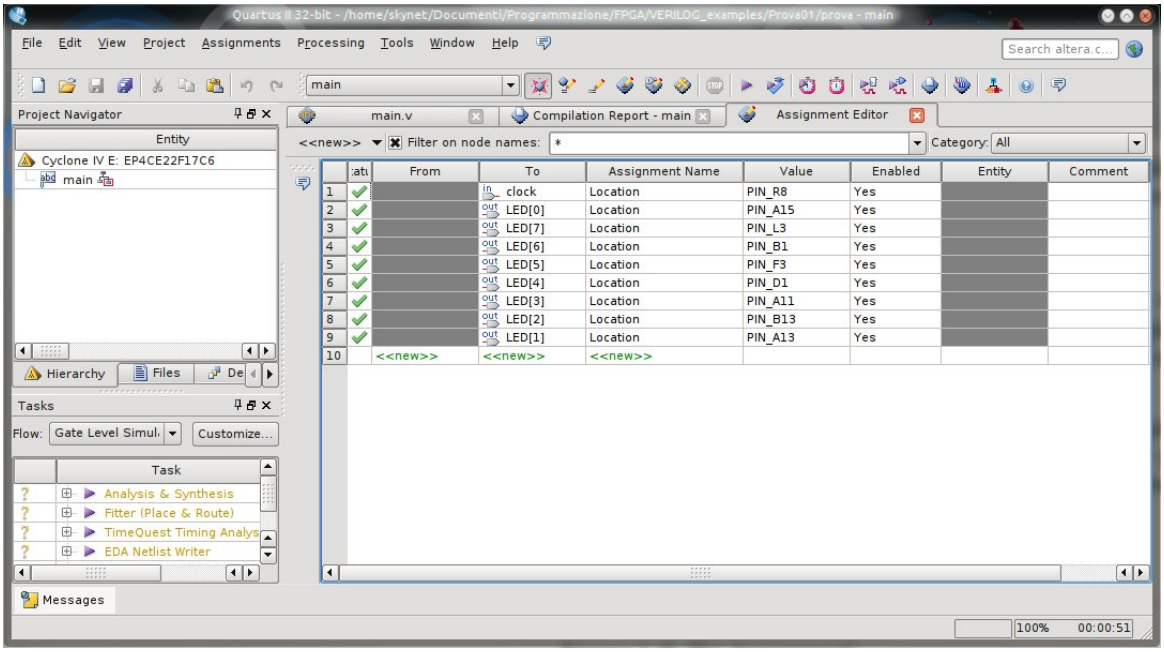



Figura 1.8: Assignment Editor - Assegnamento manuale dei pin

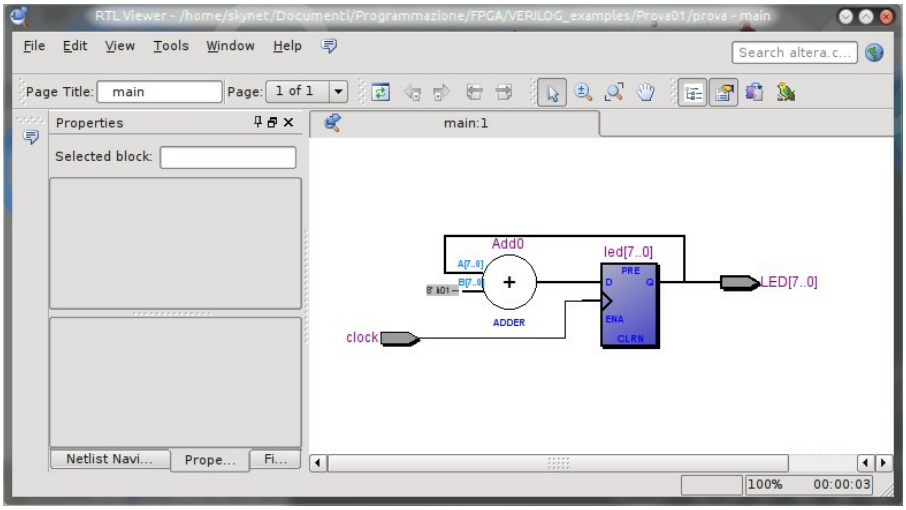



Figura 1.9: RTL Viewer

# 2 Esempi base in Verilog

Seguono ora alcuni esempi semplici mirati a capire come funzionano le basi del linguaggio Verilog. Siccome esistono molti libri e innumerevoli documenti in rete per quanto riguarda sintassi e formalismi avanzati, qui non verranno trattati nello specifico, mentre si vuole scrivere circuiti che mirino a capire determinate problematiche. Sintassi e formalismi vari verranno affrontati relativamente al singolo esempio.

Si da per scontata la conoscenza delle porte logiche di base (and, or, not, ecc) e di come esse possono essere istanziate in Verilog.

Di seguito è riportato un esempio che riporta alcune delle più comuni istruzioni:

```
module prova #(parameter n = 8)
(    input   clk,              // clock
     input   key0,             // input generico
     output  [n-1:0] LED,          // bus in uscita di 8 bit
     output  wor out);         // complex OR: 1 bit

reg [n-1:0] led = 8'h0;       // buffer da 8 bit inizializzato a 0
wire x;                       // variabile "senza memoria"

assign LED = led;             // solo i "wire" possono andare in assignment

assign out = led[1];          // out = led1 or led2 or (led7 xor led0)
assign out = led[2];          // out = led1 or led2 or (led7 xor led0)
assign out = x;               // out = led1 or led2 or (led7 xor led0)

always @(posedge clk)
begin
     if(key0)
          led = 8'b1100_0011; // led = 195 (unsigned) - Bloking assignment
     else
          led <= led + 3'd9;  // non-Blocking assignment
end

assign x = led[7] ^ led[0]; // XOR di due bit

endmodule
```

**module** descrive il circuito in termini di input-output.

**parameter** definisce un parametro e ne assegna il valore di default. I parametri servono per configurare le istanze dei moduli.

**input** variabile di ingresso. E' trattato come un tipo wire.

**output** variabile di uscita. E' trattato come un tipo wire.

**reg** variabile di tipo registro che mantiene lo stato (valore) e viene assegnato solo nei blocchi `always`.

**wire** variabili che permettono le interconnessioni tra moduli e porte logiche. Vengono assegnati tramite gli `assign` statement.

**wor** wire-or: è di fatto un "filo" in uscita da una porta OR; tutti i valori a cui è assegnata questa variabile rappresentano gli ingressi della porta OR. Esistono ovviamente anche le variabili `wand`.

**assign** statement per l'assegnazione delle variabili wire. sono istruzioni che avvengono "istantaneamente" indipendentemente dal clock, ossia il valore della variabile assegnata dipende dal valore istantaneo di ciò che segue il segno di =.

**always** blocco che gestisce gli eventi definiti dalla lista di sensitività che lo segue. Permette di realizzare codice sequenziale che viene assegnato appunto su evento e quindi le variabili assegnate in questi blocchi devono essere tipo `reg`; `begin - end` sono i due costruttori usati nella logica sequenziale.

**@(posedge clk)** è un evento scatenante. Ciò che segue viene eseguito sul fronte positivo della variabile di clock. Questa è la lista di senssitività.

**8'b11_00_11** metodo di definizione dei numeri e delle costanti: `x'yzzz_zzz` dove $x$ è la base numerica, $y$ è una lettera che indica la rappresentazione del numero, $z$ sono le cifre del numero; nell'esempio in esame il numero è costituito da 8 bit pari a 00110011. "_" è solo un separatore per rendere più chiaro il numero. Altre basi sono "$d$" per decimale, "$h$" per esadecimale; per le variabili tipo bit i valori assegnabili sono 0, 1, x e z, dove x indica "valore indeterminato" e z "alta impedenza".

= assegnazione per gli statement `assign`. Se usato nei blocchi `always`, l'istruzione è bloccante, ossia l'istruzione successiva non viene eseguita fino a che questa non è terminata.

<= assegnazione non-blocking nei blocchi `always`, ossia questo assegnamento viene eseguito in parallelo all'istruzione successiva.

## 2.1 FLIP-FLOP e LATCH

I flip-flop e i latch sono le strutture principali per la creazione di circuiti, soprattutto se verranno realizzati contatori o funzioni matematiche di qualche tipo. Seguono quindi alcuni esempi di come realizzare queste strutture. La comprensione di questo è fondamentale perchè permette di capire come scrivere strutture più complesse più avanti.

### 2.1.1 Flip-Flop con reset sincrono

L'algoritmo 1 rappresenta un flip flop con reset asincrono. Il comando di `reset` non viene incluso nella lista di sensitività. Questo implica che il segnale di `reset` verrà elaborato solo sul fronte di clock.

Algoritmo 1 Flip-Flop con reset sincrono

```
module main
(   input   clk,        // clock
    input   in,         // input
    input   reset,      // reset
    output  Q);         // uscita

reg q = 1'b0;

// Flip-Flop con reset sincrono
// ----------------------------
always @(posedge clk)   // <-- lista di sensitività
begin
    if(reset)
        q <= 1'b0;
    else
        q <= in;
end

assign Q = q;

endmodule
```

Compilando il questo codice si vede che il compilatore indica che è stato usato un solo elemento logico (figura 2.1), ossia la variabile `q` che è appunto un registro. La variabile `Q` è invece un semplice *wire*, ossia un filo che permette di portare in uscita il valore del registro interno al modulo. Anche le variabili di input sono considerate di tipo *wire*.

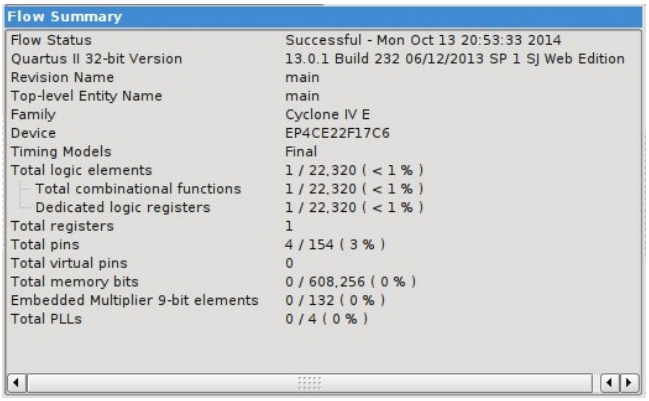



Figura 2.1: Flip-Flop con reset sincrono - Compilation Report

In figura è riportato invece il circuito ottenuto tramite *RTL Viewer* dove si vede che di fatto è stata creata una rete esterna al flip-flop costituita da un multiplexer che sceglie tra il valore '0' e il valore '`in`' da passare al flip-flop.

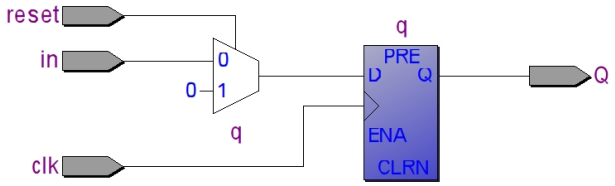



Figura 2.2: Flip-Flop con reset asincrono - RTL Viewer

### 2.1.2 Flip-Flop con reset asincrono

Nel caso in cui il comando di `reset` sia interno alla lista di sensitività (algoritmo 2), il comando di `reset` diventa asincrono perchè va ad agire sull'uscita senza aspettare la variazione del clock. Il circuito RTL è quello in figura

2.3. Questo circuito utilizza ancora un elemento logico, mentre la logica combinatoria, ossia il mux usato nel reset sincrono, è scomparsa.

Algoritmo 2 Flip-Flop con reset sincrono

```
module main
(   input   clk,        // clock
    input   in,         // input
    input   reset,      // reset
    output  Q);         // uscita

reg q = 1'b0;

always @(posedge clk, posedge reset)
begin
    if(reset)
        q <= 1'b0;
    else
        q <= in;
end

assign Q = q;

endmodule
```

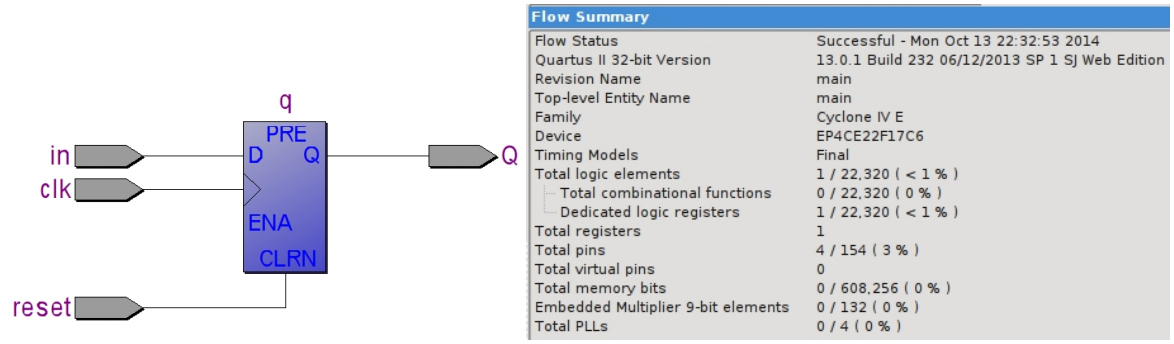



Figura 2.3: Flip-Flop con reset sincrono - RTL Viewer

### 2.1.3 LATCH

I latch mantengono l'uscita al valore dell'ingresso quando, tipicamente, il segnale di ingresso va alto. Negli algoritmi è mostrato un modo equivalente per ottenere lo stesso risultato. Anche se sembra che il secondo metodo non usi un registro, in realtà una unità di memoria verrà impliciramente generata, come si potrà osservare del *Compilation Report* o dai circuiti RTL in figura 2.4.

Algoritmo 3 LATCH

```
module main
(   input   clk,        // clock
    input   in,         // input
    input   reset,      // reset
    output  Q);         // uscita

reg q = 1'b0;

always @(posedge clk, posedge reset)
begin
    if(reset)
        q <= 1'b0;
    else
        q <= in;
end

assign Q = q;

endmodule
```

Algoritmo 4 LATCH

```
module main
(   input   clk,        // clock
    input   in,         // input
    input   reset,      // reset
    output  Q);         // uscita

assign Q = clk ? in : Q;

endmodule
```

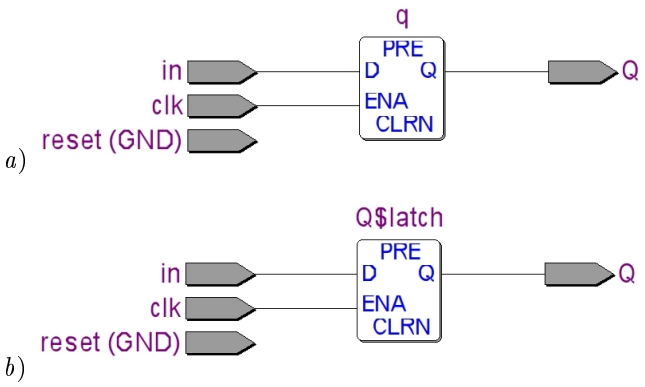



Figura 2.4: LETCH generato dagli goritmi 3 (a) e 4 (b)

### 2.1.4 LATCH con reset

Volendo aggiungere il reset ad un latch in modo simile a come era definito nel flip-flop, si genererà semplicemente una più o meno complessa logica combinatoria, come riportato in figura 2.5.

Algoritmo 5 LATCH con reset

```
module main
(   input   clk,        // clock
    input   in,         // input
    input   reset,      // reset
    output  Q);         // uscita

reg q = 1'b0;

always @(*)
begin
    if(clk)
        q <= in;
    else
    if(reset)
        q <= 1'b0;
end

assign Q = q;

endmodule
```

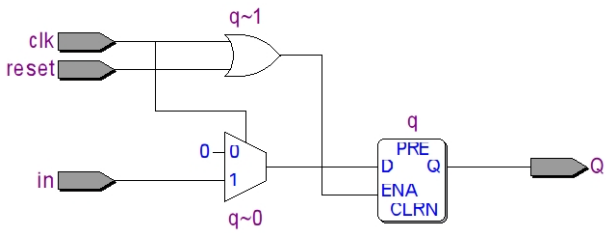



Figura 2.5: LATCH con reset - RTL Viewer

## 2.2 Shift Register

Uno shift register è una catena di celle di memoria da un bit ciascuna che a fronte di un evento di clock spostano il valore di una cella alla cella successiva.

Per realizzare uno shift-register in Verilog ci sono come sempre molti modi. Tipicamente si utilizzano i comandi di shift simili a quelli del C come segue:

```
y = x << 1'b1;      // shift logico a sinistra
y = x >> 1'b1;      // shift logico a destra
y = x <<< 1'b1;     // shift aritmetico a sinistra
y = x >>> 1'b1;     // shift aritmetico a sinistra
```

Oppure sostituendo i vari bit come segue, che è un sistema forse un po' più versatile:

```
y = {y[n-1:0], x};  // shift a destra con sostituzione del primo bit con il valore di x
y = {x,y[n:1]}; // shift a sinistra con sostituzione del'ultimo bit con il valore di x
```

L'esempio riportato nell'algoritmo 6 permette di settare il valore del primo led degli 8 sulla DE0-Nano premendo il tarso KEY1, mentre premendo il tasto KEY0 si ottiene l'avanzamento (shift) dei bit.

**Algoritmo 6** Shift Register

```
module main
(   input       Key0,   // shift dei bit
    input       Key1,   // set a 1 del primo bit
    output      [7:0]LED);

reg [7:0] led = 8'h0;
wire key0, key1;

always @(posedge key0 or posedge key1)  // lista di sensitività
begin
    if(key1)    // con "posedge", l'if deve essere vero con key=1; viceversa con "negedge"
        led[0] <= 1'b1;
    else
        led <= led << 1'b1;
end

assign key0 = ~Key0;    // equivalente in questo caso a !Key0
assign key1 = ~Key1;
assign LED = led;

endmodule
```

**Nota:** Key0 e Key1 sulla DE0-Nano sono cablate a logica inversa. Per questo, se si vuole usare il fronte positivo per intercettare gli eventi, occorre negarli come in questo esempio. In più se nella lista di sensitività si usa "`posedge`", allora l'argomento dell'`if` deve essere vero con `key1 = 1`, mentre con negedge occorrerebbe negare l'espressione altriemnti si avrebbe un errore di compilazione.

Occorre quindi settare i pin. Si può utilizzare sia il *Pin Planner* che l'*Assignment Editor*; quest'ultimo è forse un po' più versatile perchè permette di settare anche i pin come virtuali. Prima di avviare il programma, occorre compilare prima il software e quindi, avviare l'*Assignment Editor* da Assigments ▷ Assignment Editor. Il programma partirà senza campi preimpostati. Occorre inserire manualmente i pin solo nella colonna "*To*", quindi nella colonna "*Assignment Name*" occorre selezionare "*Location (Accept wildcards/groups)*" come in figura 2.6. I vari pin invece vanno indicati a mano esattamente come in figura. Il Pin-out della scheda è disponibile in rete. Terasic e ALTERA forniscono anche il file *.qsf* con già l'assegnazione per tutti i pin.

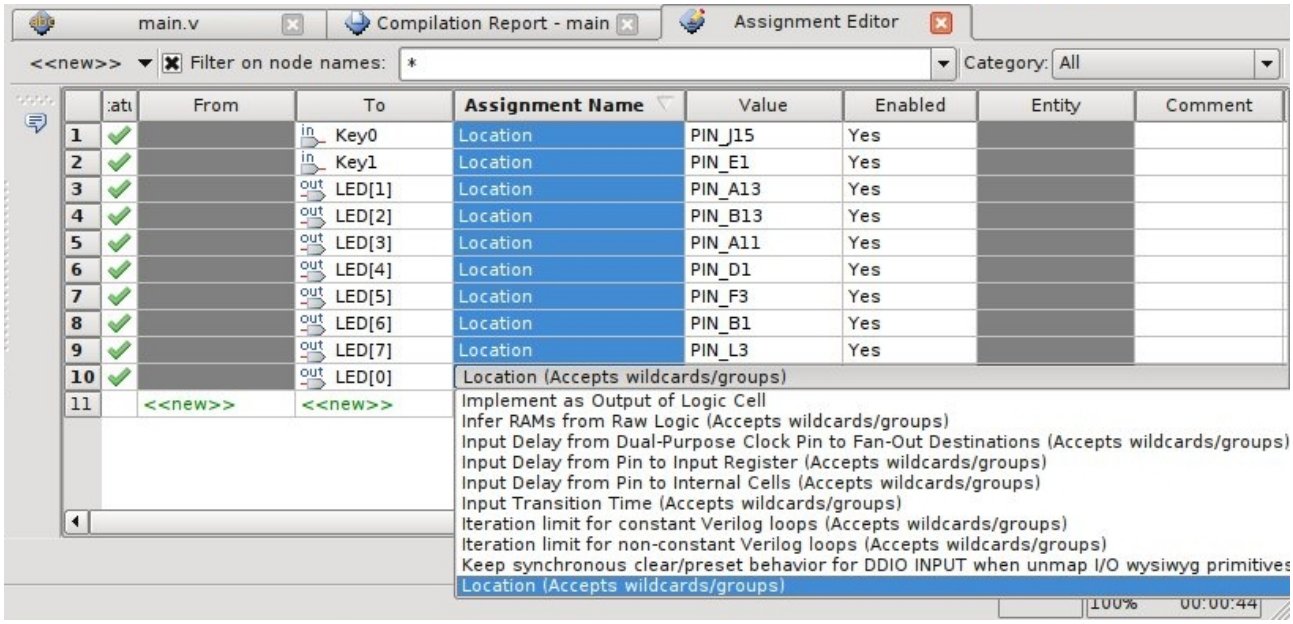



Figura 2.6: Assignment Editor

A questo punto occorre ricompilare e scaricare sulla scheda il software che andrà a scriversi in ram. Per farlo occorre lanciare il programmatore con Tools ▷ Programmer e, se non ha caricato il file, occorre aggiungerlo manualmente. Il file è nella directory *output_files/* ed è il file *main.sof*.

**Nota implementativa:** una volta programmata la scheda, basta agire sulle Key0 e Key1 per testare il funzionamento. Se i LED avanzano velocemente è a causa dei rimbalzi multipli dei pulsanti che andrebbero filtrati in qualche modo, per esempio con un filtro temporale che restituisca il valore dopo un certo tempo.

**Nota Tri-state pin:** è bene sempre definire che tutti i pin non assegnati diventino input in alta impedenza. Per farlo occorre invocare Assigments ▷ Device ▷ Device and Pin Option e scegliere "*As input tri-state*" come in figura 2.7. Se i led non fossero tutti assegnati e non si facesse questo procedimento, si vedrebbero i led legermente accesi perchè sarebbe setato un pull-up. Questo procedimento si può fare già all'inizio in fase di creazione del progetto.

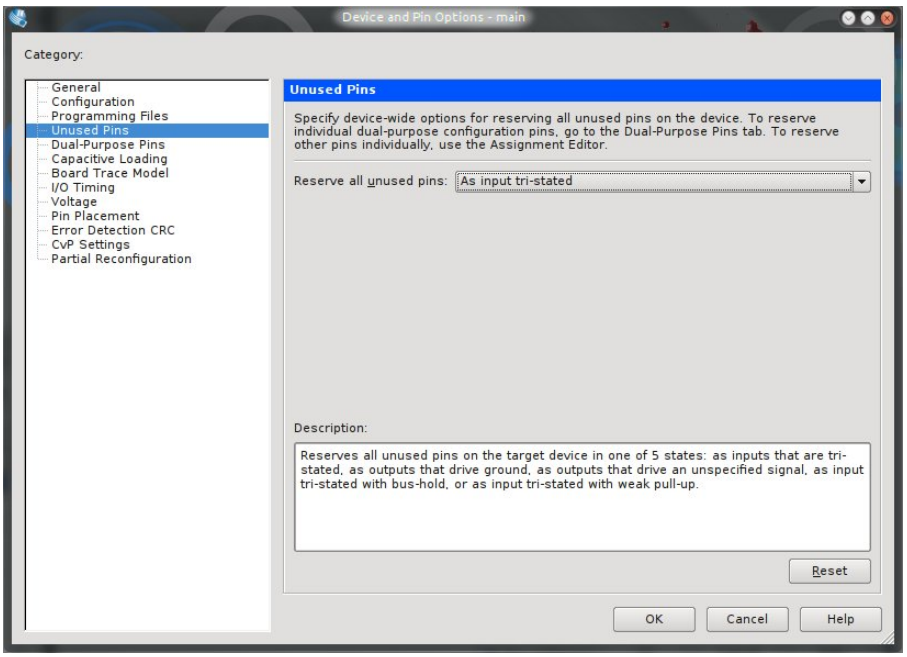



Figura 2.7: Device and Pin Options - Unused Pin

### 2.2.1 Istruzioni Parallele e Sequenziali - Blocking & non-Blocking statements

I circuiti programmabili hanno tutti la peculiarità che pressoché tutto il codice è parallelo. Per verificare il parallelismo delle istruzioni si considerino i due algoritmi seguenti. Entrambi accendono un led e lo spostano fino all'ultima posizione a intervalli di circa 1s.

Scaricando sulla DE0-Nano l'algoritmo numero 7, si vedrà che il primo led (led0) si accenderà e scorrerà fino alla posizione 8 (led7 acceso). A questo punto tutti i led si spengono per poi accendere nuovamente il led0 dopo circa un secondo e ricominciare il processo. Questo è dovuto al fatto che nel blocco always gli assegnamenti sono non bloccanti. Si noti che se l'assegnazione `led <= led << 1'b1` venisse posta dopo l'`if`, il risultato sarebbe identico.

**Algoritmo 7** Shift register - non blocking

```
module main(    input wire clk,
                output wire [7:0] Q);
parameter n = 24;

reg [7:0] led;
reg [n:0] count;

initial
begin
    led <= 0;
    count <= 0;
end

always @(posedge clk)
begin
    count <= count + 1'b1;
end

always @(posedge count[n])
begin
    led <= led << 1'b1;      // non-blocking
    if(led == 0)
        led <= 1'b1;         // questa istruzione non può essere blocking
end

assign Q = led;

endmodule
```

**Nota:** se la prima assegnazione è non-blocking (ossia `<=`) la seconda all'interno dell'`if` non può essere blocking.

Scaricando l'algoritmo 8 invece si ha che i led non si spegneranno mai tutti. Questo perchè nell'istante in cui si ha l'ultimo shift, la variabile `led` diventa zero e successivamente viene eseguito l'`if` che ripristina il primo bit a 1.

**Algoritmo 8** Shift register - blocking

```
module main(    input wire clk,
                output wire [7:0] Q);
parameter n = 24;

reg [7:0] led;
reg [n:0] count;

initial
begin
    led <= 0;
    count <= 0;
end

always @(posedge clk)
begin
    count <= count + 1'b1;
end

always @(posedge count[n])
begin
    led = led = 1'b1;    // blocking
    if(led == 0)
        led = 1'b1;      // questa istruzione può essere blocking
end

assign Q = led;

endmodule
```

## 2.3 Contatori e Moltiplicatori

Contatori e moltiplicatori si realizzano con delle normali operazioni. In particolare i contatori vengono realizzati sfruttando gli elementi logici dell'FPGA, mentre i moltiplicatori, quando possibile, vengono implementati usando i moltiplicatori hardware già presenti in alcune FPGA come la Cyclone IV in esame.

L'algoritmo 9 riporta una mac normalizzata. Si noti che nel compilation report (figura 2.8) viene indicato l'utilizzo di 8 embedded multiplayer. Per sfruttare le risorse dell'FPGA e non i moltiplicatori embedded, si può usare il moltiplicatore dato con le Megafunction di ALTERA (Tools▷MegaWizard Plugin Manager e scegliere LPM_MULT).

**Algoritmo 9** MAC normalizzata

```
module mac( input   clk,
            input [31:0] A, B, C,
            output [31:0] Q);

reg [63:0] q;

always @(posedge clk)
begin
    q <= A * B + {C,{32{1'b0}}};
end

assign Q = q[63:32];

endmodule
```

| | |
|---|---|
| Flow Status | Successful - Wed Oct 15 23:03:03 2014 |
| Quartus II 32-bit Version | 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition |
| Revision Name | main |
| Top-level Entity Name | main |
| Family | Cyclone IV E |
| Device | EP4CE22F17C6 |
| Timing Models | Final |
| Total logic elements | 112 / 22,320 ( < 1 % ) |
| Total combinational functions | 112 / 22,320 ( < 1 % ) |
| Dedicated logic registers | 32 / 22,320 ( < 1 % ) |
| Total registers | 32 |
| Total pins | 129 / 154 ( 84 % ) |
| Total virtual pins | 0 |
| Total memory bits | 0 / 608,256 ( 0 % ) |
| Embedded Multiplier 9-bit elements | 8 / 132 ( 6 % ) |
| Total PLLs | 0 / 4 ( 0 % ) |

Figura 2.8: MAC - Compilation Report

# 3 Simulazione

La simulazione è il primo strumento di debug che permette di valutare sia se il codice scritto funziona dal punto di vista logico che funzionale. In particolare si hanno due tipi di simulazione:

**RTL** *Register-transfer level simulation*: verifica il funzionamento del circuito a livello logico e quindi ideale. Permette di simulare ogni singolo modulo qualunque sia il linguaggio con cui è stato scritto, senza dover simulare tutto il progetto.

**Gate-Level** o anche *Functional simulation*: è la simulazione funzionale, ossia una simulazione realistica del circuito scritto tenendo conto dei ritardi dovuti all'hardware. Questo tipo di simulazione è più complesso, soprattutto per simulare singole parti di circuito; inoltre è molto importante definire il linguaggio con cui è descritto il circuito. Se il programma è scritto sfruttando molti linguaggi, occorre convertire tutto il progetto nel linguaggio scelto per la simulazione.

L'algoritmo 10 ha come scopo la simulazione del circuito per valutare innanzitutto le differenze tra i due tipi di simulazione. L'obiettivo è portare in uscita su due pin un clock e il valore di un registro il cui valore è condizionato proprio dal clock. Il clock in ingresso è da 50MHz e viene poi ridotto a 10MHz tramite un PLL inserito con una MegaFunction. Per inserire il PLL occorre invocare il Megafunction Wizard con Tools▷MegaWizard Plug-In Manager. Comparirà quindi un form a cui occorre indicare di creare una nuova megafunction cliccando su "*Create a new custom megafuncion variation*". Cliccando su "Next" comparirà il form da cui scegliere la MegaFunction e occorre settare i vari campi come in figura3.1:

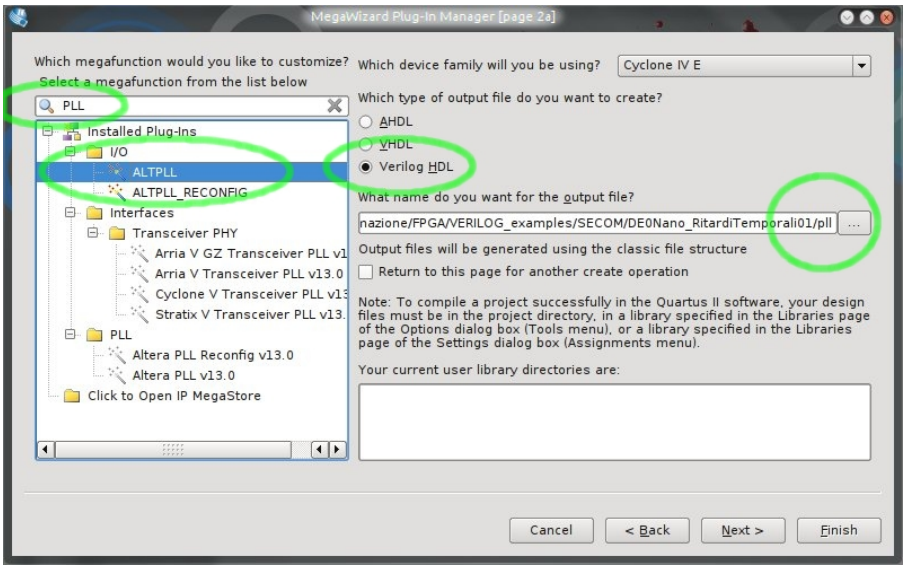



Figura 3.1: Megafunction: variation declaration

- definire un nome della variante, in questo caso *pll*
- scegliere il linguaggio in cui la variante verrà definita, nel nostro caso *Verilog HDL*
- indicare nel filtro PLL in modo da trovare subito la funzione desiderata
- scegliere ALTPLL, ossia l'ALTERA Phase-Locked-Loop megafunction

Cliccando su Next comparirà il configuratore della megafunction (figura 3.2). Nella prima pagina occorrerà indicare che il clock di ingresso è 50 MHz e che lo speed grade[3] della FPGA è 6[4]. Avanzando nelle schede, eliminare tutte le selezioni fino ad ottenere un solo segnale di ingresso (`inclk0`) e uno di uscita (`c0`).

[3]Lo speed grade è uno dei parametri più importanti che definisce principalmente la velocità del componente e definisce i limiti massimi del PLL

[4]Speed Grade = 6 si deduce dal nome dell'FPGA: EP4CE22F17C**6**N. Questo valore indica che la massima frequenza ammissibile in uscita dal PLL è di 315MHz.

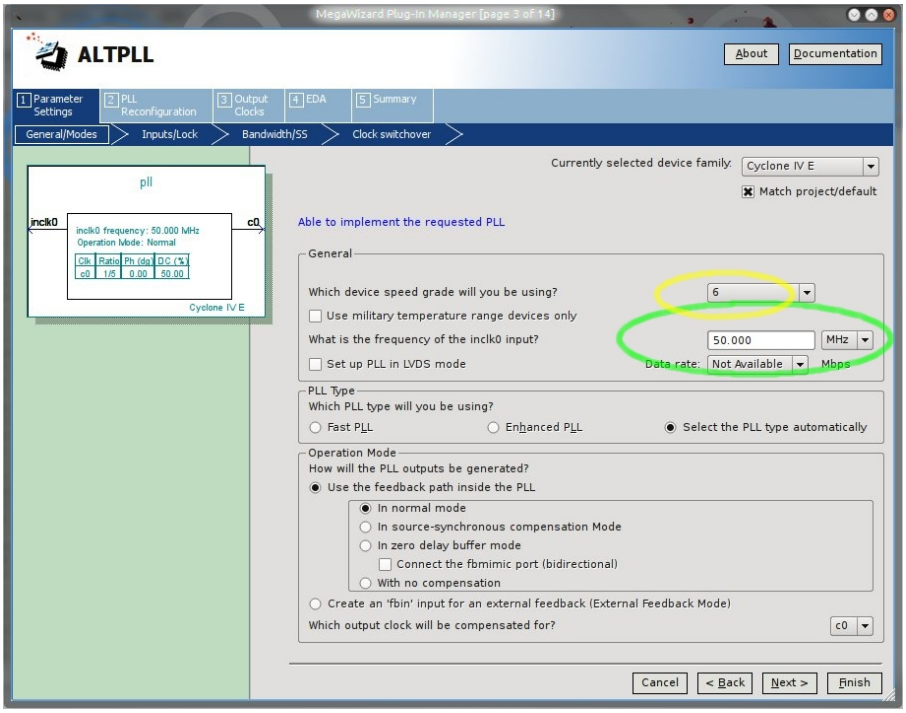



Figura 3.2: Megafunction configuration

**Algoritmo 10** Esempio per la simulazione

```
module main
(    input        clk,        // 50 MHz
     output   clkMirror,
     output   Q);

wire clk10;          // 10 MHz
reg q = 1'b0;
reg [7:0] count = 8'h0;

pll pll_inst
(    .inclk0 ( clk ),
     .c0 ( clk10 ));

assign clkMirror = clk10;
assign Q = q;

always @(posedge clk10)
begin
     count <= count + 1'b1;
     q <= count[7];
end

endmodule
```

Si noti che per effettuare le simulazioni sarebbe meglio settare anche i pin, o quantomeno eseguire la prima fase di compilazione (sintesi).

## 3.1 RTL Simulation

Per prima cosa, questo tipo di simulazione non necessita tipicamente di nessuna configurazione particolare. Cliccando due volte su uno dei moduli è possibile simularli anche singolarmente. Se la simulazione non funzionasse, occorre in Quartus II ricreare i simboli del file che contiene il modulo. Non è questo il caso, ma a titolo informativo occorre cliccare file (*main.v*) con il tasto destro e cliccare su "*Create Symbol Files for Current File*".

A questo punto si può rilanciare la simulazione cliccando su Tools ▷ Run Simulation Tool ▷ RTL Simulation e verrà avviato il *ModelSim* (figura 3.3), ma tutto ciò non basterà: infatti se si clicca due volte sul modulo *main* per simularlo, *ModelSim* non funzionerà a causa del fatto che il programma contiene le Megafunction ALTERA.

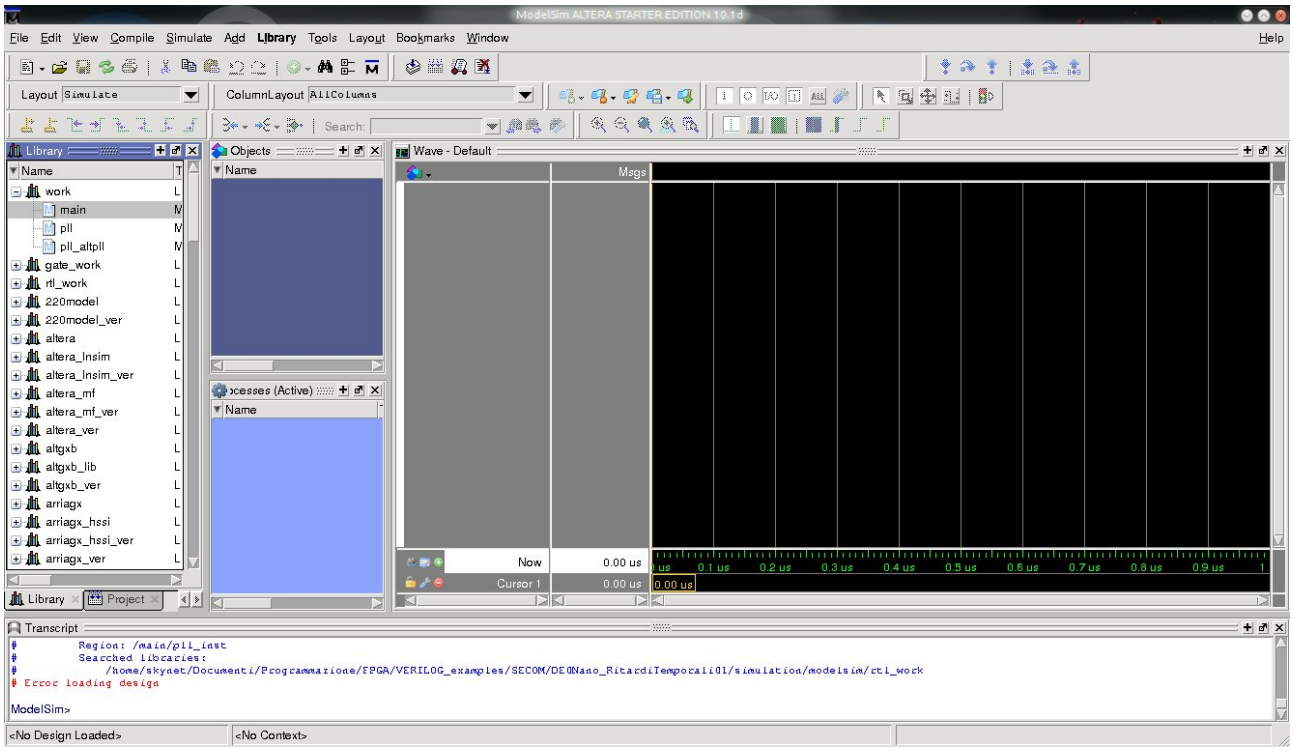



Figura 3.3: Model Sim - RTL Simulation

Perché si possano simulare questi progetti, occorre parametrizzare la simulazione come segue:

- avviare la simulazione dal menù di *ModelSim*: Simulate ▷ Start Simulation
- selezionare dal form appena avviato (figura 3.4-a) Work ▷ main
- selezionare quindi le librerie manualmente cliccando sull'etichetta *Libraries* (figura 3.4-b). Aggiungere le seguenti librerie relative al linguaggio Verilog[5]: *altera_ver*, *altera_mf_ver* (per le megafunction), *lpm_ver* e soprattutto la *cycloneive_ver*.
- Quindi eseguire la simulazione cliccando su "OK"

A questo punto si può procedere con la simulazione. Nei riquadri centrali (*Objects*) compariranno tutte le variabili simulabili. trascinarle nel form *Wave-Default* e configurare il clock di ingresso cliccando col tasto destro su `/main/clk` (in *Wave-Default*), scegliere "*Clock*" e settare nel campo "*Period*" del form che comparirà, 20ns e quindi "OK". A questo punto settare come tempo di simulazione un tempo adeguato, per esempio 100us ed eseguire la simulazione cliccando sul tasto a fianco. Apparirà l'oscillogramma come in figura 3.5. Cliccando sulle forme d'onda, con i comandi "c", "o" e "i" si potrà zoommare fino a vedere che tutti i fronti sono perfettamente allineati al clock a 50MHz senza alcun ritardo. Quando l'8° bit del contatore è alto, anche l'uscita `Q` è alta.

## 3.2 Gate-Level Simulation

Per eseguire la simulazione funzionale occorre compilare il progetto eseguendo l'"*EDA Netlist Writer*" che, se è selezionato nel riquadro "*Tasks*" la compilazione "*Gate Level Simulation*", tutto avverrà in automatico.

A causa del fatto che il progetto è stato scritto in Verilog e che è stato indicato a ModelSim che il linguaggio di simulazione è appunto il Verilog HDL, occorre sempre definire le librerie per la simulazione come fatto in

[5] In realtà non servono tutte queste librerie per questa simulazione, ma per altre più complesse sì

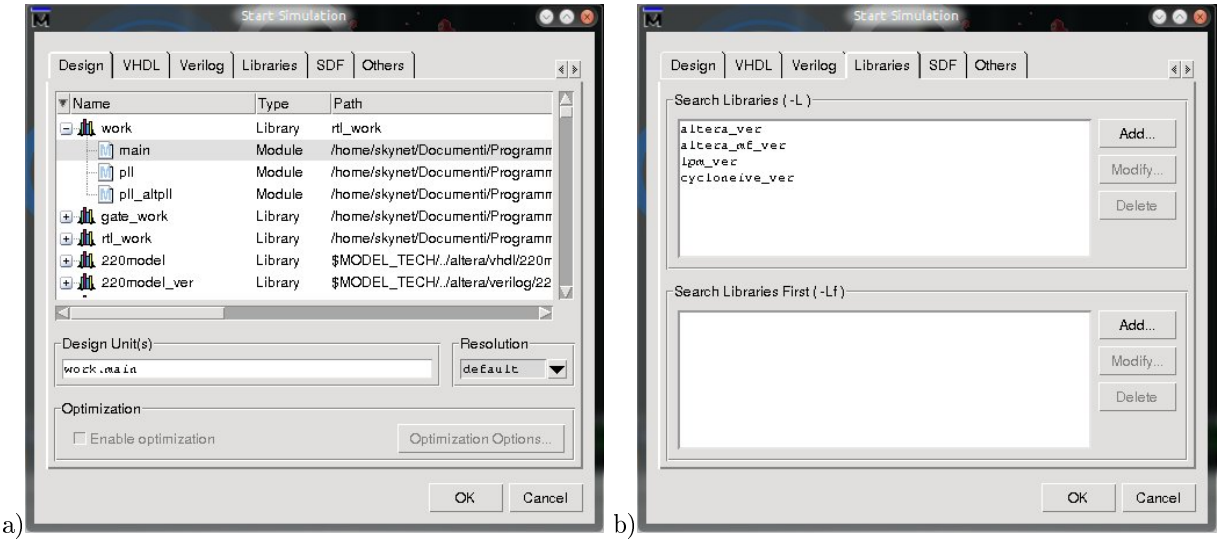



Figura 3.4: Start Simulation

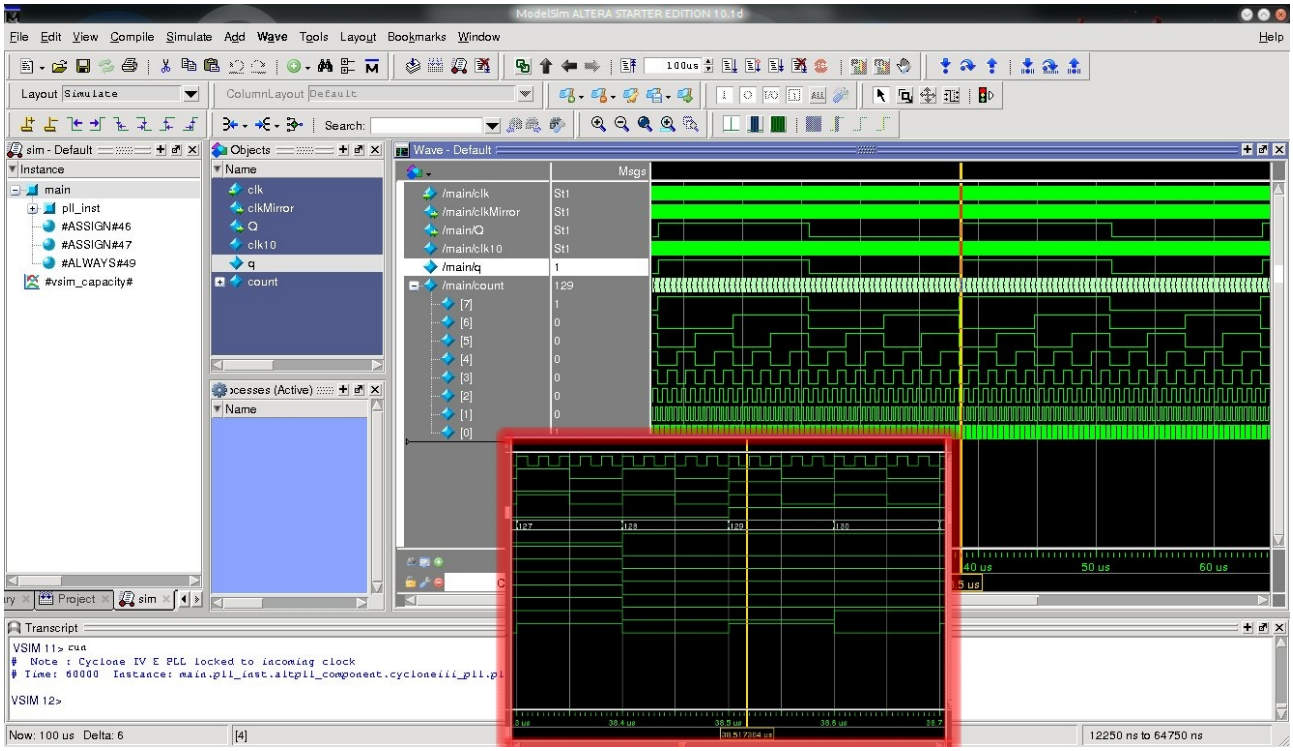



Figura 3.5: Esecuzione della simulazione RTL

precedenza al paragrafo 3.1. Avviare quindi ModelSim da Quartus II tramite Tools ▷ Run Simulation Tool ▷ Gate Level Simulation e quindi procedere come segue:

- avviare la simulazione dal menù di *ModelSim*: Simulate ▷ Start Simulation
- selezionare dal form appena avviato (figura 3.4-a) Work ▷ main
- selezionare quindi le librerie manualmente cliccando sull'etichetta Libraries (figura 3.4-b). Aggiungere le seguenti librerie relative al linguaggio Verilog: *altera_ver*, *altera_mf_ver* (per le megafunction), *lpm_ver* e soprattutto la *cycloneive_ver*.
- Quindi eseguire la simulazione cliccando su "OK"

A differenza del caso RTL si vede che nel riquadro "*Objects*" compaiono moltissime altre variabili. E' quindi bene inserire solo le variabili che servono: sicuramente gli input e output (distinguibili da una freccia verde sui rombi) ed eventuali variabili interne tra cui il clock a 10MHz identificato dalla variabile "`sim:/main/\pll_inst|altpll_component|auto_generated|wire_pll1_clk[0]~clkctrl_outclk`", ossia l'uscita del PLL. Anche in questo caso si aggiunge il clock a 50MHz settando il periodo a 20ns e quindi si simulano 100us. Il risultato è quello in figura 3.6.

In questa simulazione si vede che i vari fronti non sono tutti sincronizzati, ma ci sono dei ritardi variabili in funzione dei vari segnali. Tra il clock riportato sul pin in uscita a l'uscita `Q`, ci sono quasi 0.815ps.

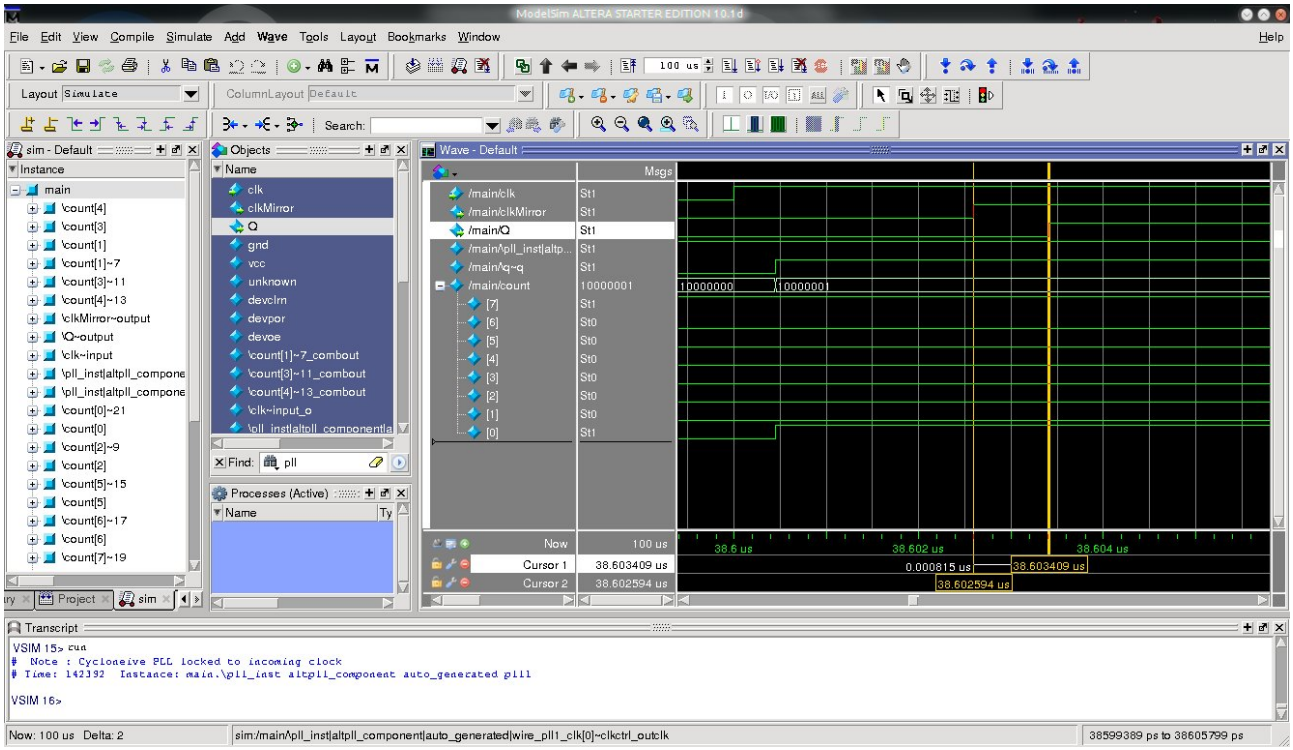



Figura 3.6: Esecuzione della simulazione Gate-Level

**Nota:** per automatizzare le simulazioni future è possibile salvare lo stato attuale cliccando prima nel riquadro Wave, e quindi dal menù File ▷ Save Format salvare il file *wave.do* (si può anche cambiare nome). Questo file non è altro che uno script e quindi editandolo aggiungendo come prima linea "`vsim -L altera_ver -L altera_mf_ver -L lpm_ver -L cycloneive_ver -do main_run_msim_gate_verilog.do -l msim_transcript -i work.main`", ogni volta che si aprirà questo file partirà questa simulazione, ma senza settaggi e forzamenti delle variabili. Aggiungendo alla fine del file gli appositi comandi, sarà possibile anche configurare le variabili. Tutti i comandi che si danno graficamente verranno riportati nella console "*Transcript*" in basso nel *ModelSim*; basta copiarli nel file *.do*.

## 3.3 Verifica dei ritardi in hardware

I pin scelti e settati tramite *Assignent Editor* (figura 3.7) sono i seguenti:

`clk` PIN_R8, clock da 50MHz

`clkMirror` PIN_J14, mirror del clock a 10MHz: pin 40 del connettore GPIO01 (GPIO_133 sugli schemi della scheda)

`q` PIN_L13, 8° pin del contatore: pin 36 del connettore GPIO01 (GPIO_129 sugli schemi della scheda)

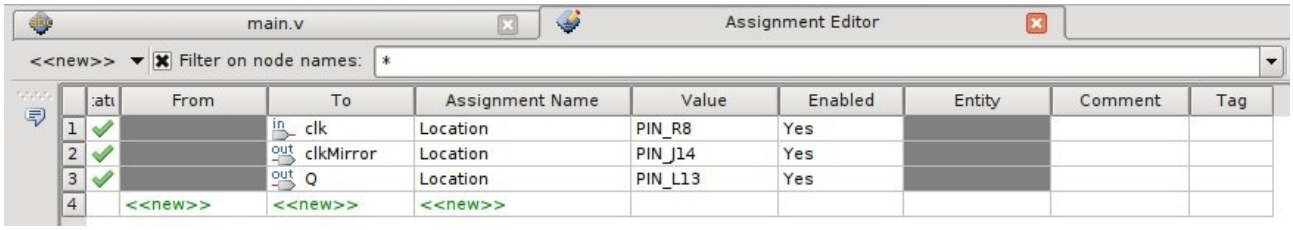



Figura 3.7: Assegnamento Pin per verifica comportamento hardware

In questo esempio è stato introdotto un PLL per portare da 50 a 10MHz il clock in quanto la lettura di 50MHz su è abbastanza disturbata. Non è chiaro se sia dovuto al pin o alla banda passante della sonda dell'oscilloscopio. A questo punto collegare al pin 12 del connettore GPIO01 la massa dell'oscilloscopio e le sonde sul pin 40 e sul 36. In figura 3.8-a verifica che la frequenza del clock di uscita siano 10MHz, mentre in figura 3.8-b si è cercato di verificare il ritardo di simulazione (figura 3.6) che è stato misurato come un po' meno di 1 pico secondo; è difficile misurare tempi di questa entità, ad ogni modo la figura 3.8-b misura un'ordine di grandezza simile e l'intervallo è misurato ad occhio è stato preso forse un po' più largo. Ad ogni modo l'obiettivo

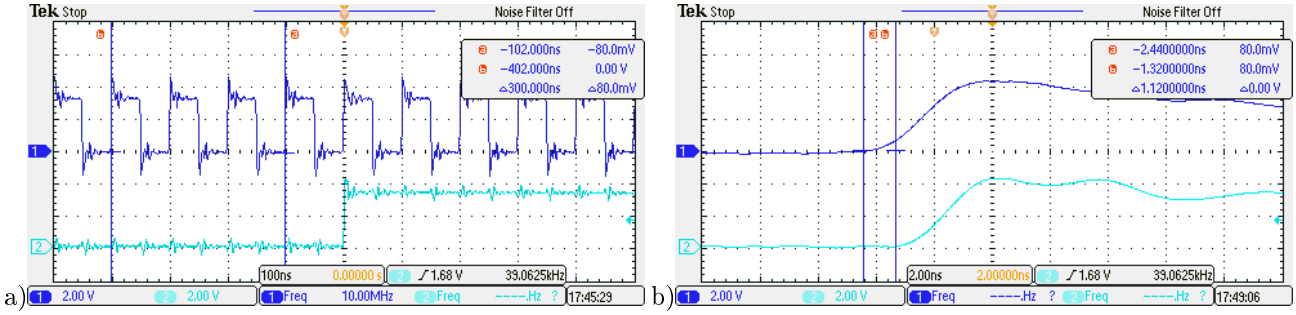



Figura 3.8: Verifica frequenza a 10MHz (a) e misure dei ritardi dell'algoritmo 10 (b)

è dimostrare il ritardo hardware che è sempre da considerare quando si sviluppa un circuito. Tale ritardo non è sempre costante: infatti il ritardo appena misurato è estremamente contenuto, ma questo dipende dalle logiche implementate, dal tipo di FPGA, ecc. Di norma è bene considerare che le operazioni abbiano un ritardo pari o prossimo al valore dello speed grade dell'FPGA (6 ns in questo caso).

# 4 Altri esempi

A fronte del fatto che i tempi vano opportunamente verificati in funzione delle operazioni che si fanno, occorre tenere presente che ogni operazione ha pesi differenti. Seguono quindi degli esempi che mostrano in simulazione i ritardi di alcune operazioni. Questi tempi sono attendibili.

## 4.1 Moltiplicazione con una pipe

La moltiplicazione che segue viene eseguita sul fronte di clock in modo da poter verificare quanto trascorre dal fronte di clock alla stabilizzazione dei valori:

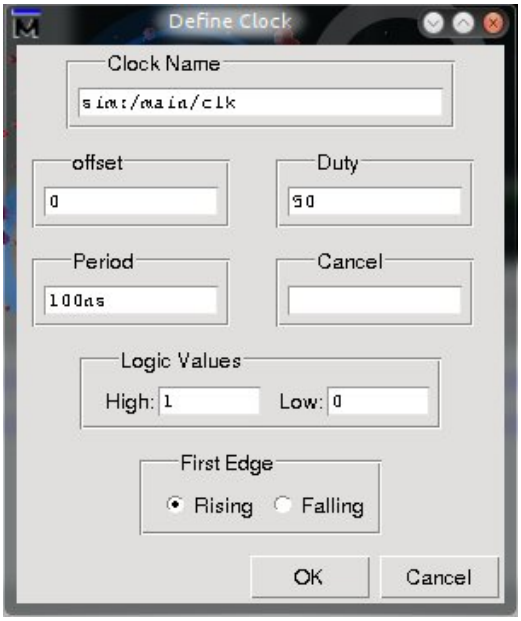
```
module main #(parameter n = 16) // metodo migliore di definizione dei parametri per simulazioni RTL
(    input   clk,                // 50 MHz
     input   [n-1:0]A, B,        // A e B sono variabili a 16 bit
     output  [n-1:0]Q);

reg [(n*2)-1:0] q;               // valore a 32 bit: risultato della moltiplicazione

always @(posedge clk)
begin
     q <= A * B;
end
assign Q = q[(n*2)-1:n];         // Moltiplicazione normalizzata: uso solo i 32bit alti

endmodule
```

Compilando e avviando la simulazione *Gate-Level*, eseguire la simulazione ricordandosi di aggiungere le librerie come spiegato a pagina 21. Aggiungere quindi tutti gli input, gli output e la variabile Q. Settare quindi i valori dei vari input a piacere. Essendo una simulazione il clock può essere fissato a meno di 50MHz, per esempio a 100ns[6]; A e B possono essere settate a piacere, per esempio 2000 e 4000 rispettivamente. Siccome "tutto è un bit" anche i numeri sono definiti in bit. Per evitare di dover inserire tutti i bit e convertire i numeri, è possibile definire la base numerica per esempio con 10#2000 che indica base 10 e valore 2000 come mostrato in figura 4.1. La visualizzazione dei valori è poi conveniente metterla in decimale.

a) 

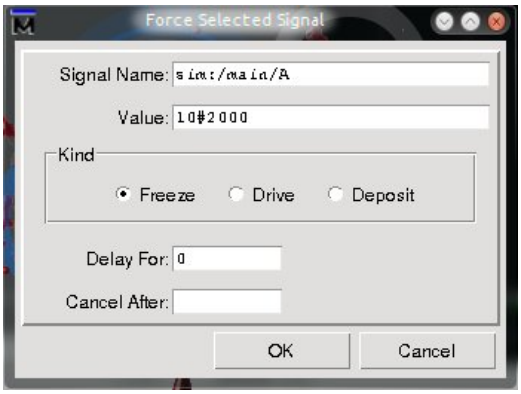

b) 


Figura 4.1: Set clock (a) e dei valori di `A` e `B` (b)

Eseguendo quindi una simulazione e andando a cambiare dopo 400us il valore di `A`, si vede al successivo fronte di clock quando il valore si è stabilizzato (figura 4.2-a). La prima cosa che si nota è che inizialmente `Q` ha un valore pari a "*x*"; questo perchè non è stato assegnato nessun valore iniziale. `q` invece ha gli ultimi valori a "*z*"; questo perchè quei valori tendenzialmente non servono alla moltiplicazione perchè comunque vengono presi soltanto i bit più significativi e quindi vengono posti in "alta impedenza" (*z* appunto).

Proseguendo si vede che in 4.2-b il ritardo dal fronte di clock positivo e il risultato stabile della moltiplicazione sono 3.53ns. Non a caso si è detto "risultato stabile": se infatti si ingrandisce l'intorno del cambio di valore del risultato si osserva che il valore in realtà è cambiato due volte andando, la prima volta, anche a un valore più elevato del risultato finale. Questo spiega perchè è sempre bene clockare sia l'ingresso di una funzione, sia clockarne l'uscita in modo da renderla disponibile solo dopo un adeguato tempo oltre il quale si è sicuri che il valore sia stabile onde evitare problemi transitori. Quest'ultimo passo in questo esempio non viene mostrato, ma esiste sul sito ALTERA un ottimo esempio.

[6]Si noti che ridurre la frequenza dei segnali, soprattutto quelli di clock, può accelerare in alcuni casi le simulazioni.

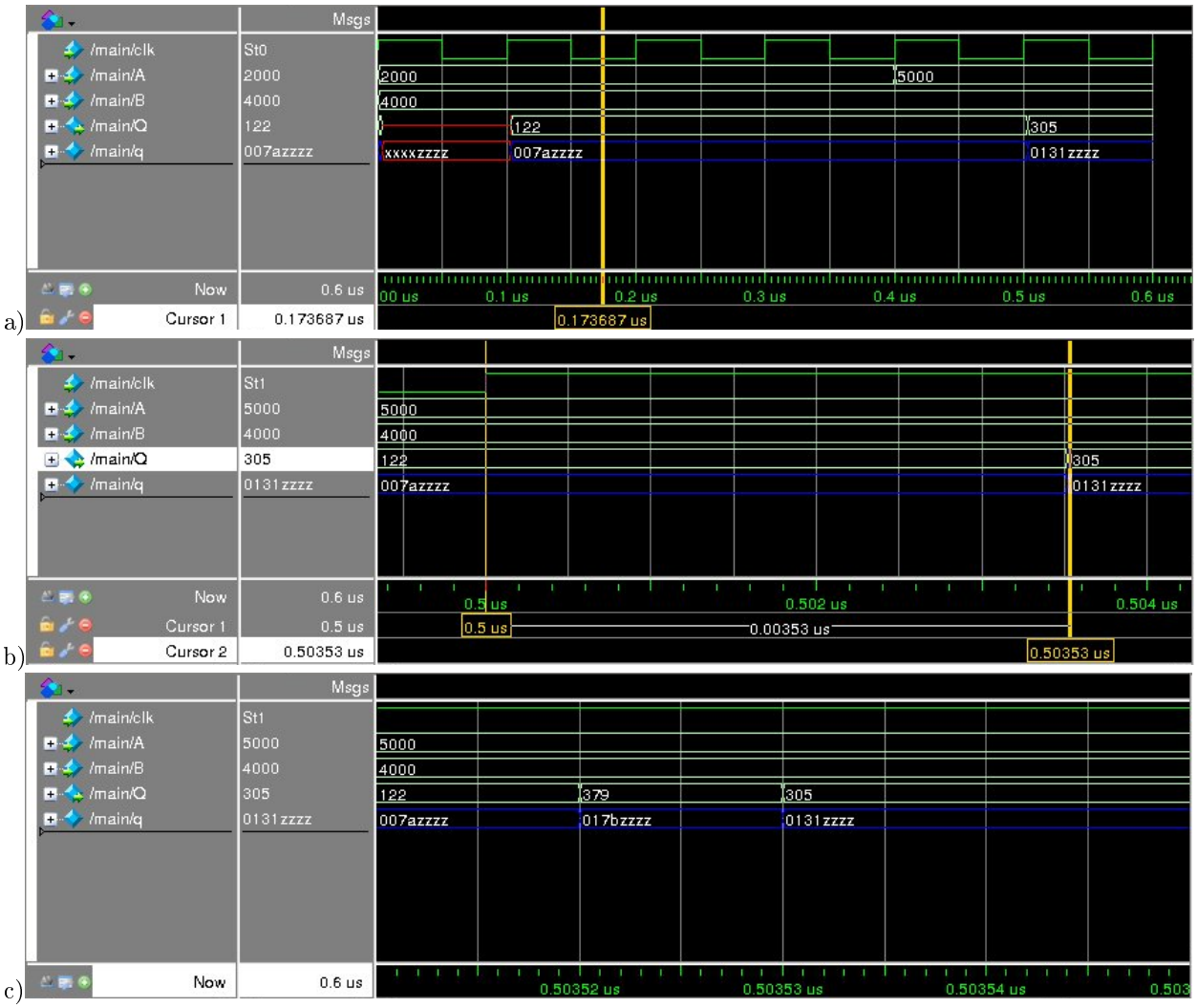



Figura 4.2: MUL - (a) Simulazione, (b) misura ritardo del calcolo, (c) variazione intermedia del calcolo

**Nota:** La moltiplicazione normalizzata qui implementata è realizzata con variabili intere unsigned a 16 bit. Una completa moltiplicazione normalizzata che tenesse conto del segno dei due valori a 16 bit, dovrebbe essere diversa perchè il bit di segno dei due numeri moltiplicati andrebbe a finire in un unico bit di segno. In questo caso, quindi, la vera moltiplicazione sarebbe tra due numeri positivi a 15bit seguita da un'opportuna gestione del segno.

## 4.2 PWM

La *Pulse Width Modulation* è una tecnica che permette di controllare un'uscita digitale in modo che la tensione media di questa uscita in un dato intervallo di tempo (detto periodo di commutazione) risulti pari al valore di tensione per il duty-cycle $\delta$:

$$\delta = \frac{T_{on}}{T}$$

dove $T_{on}$ è il il tempo in cui l'uscita permane allo stato alto (1 logico, 3.3V fisico), mentre $T$ è il tempo totale ($T_{on} + T_{off}$); per quanto detto quindi, se la tensione in uscita su un pin allo stato logico "1" della FPGA è a $V$=3.3V, allora la tensione media in un periodo di switching sarà:

$$v = V \cdot \delta = V \cdot \frac{T_{on}}{T}$$

Questa tecnica viene usta per il controllo di inverter, chopper, ecc. Infatti se l'uscita dell'FPGA va a comandare l'accensione e lo spegnimento di un componente di potenza (es: IGBT), è possibile modulare una tensione di uscita che, mediamente, abbia una forma d'onda desiderata (per esempio una sinusoide). Applicazioni tipiche sono il controllo di motori sincroni, asincroni, sincroni a magneti permanenti, generatori statici, ecc.

La PWM è una funzione che è quasi sempre fornita da opportune periferiche dei DSP o microprocessori industriali per inverter relativamente semplici. Realizzare questa funzione su FPGA tipicamente ha il vantaggio di poter realizzare molti canali PWM, poter avere una elevata risoluzione in tensione, avere una elevata diagnostica, poter controllare varie topologie anche molto complesse, ecc.

### 4.2.1 Pulsazione di un LED

Per ottenere una PWM occorre comparare un segnale portante (*carrier*) con un segnale modulante che rappresenta ad ogni tempo ciclo il duty-cycle. L'esempio che segue genererà una carrier a 14 bit e la confronterà con un duty-cycle a 12. Per semplicità il duty-cycle è aggiornato a fronte dell'azzeramento della carrer; siccome la carrier è un contatore che incrementa un bit alla volta, il suo valore tornerà 0 quanto l'ultimo suo bit andrà a 0.

```
module SimplePWM
(    input       clk,            // 50MHz - PIN_R8
     output      LED);           // LED0 - PIN_A15

parameter c = 14;
parameter d = 12;

reg [d:0] duty_cycle = 1'd0;
reg [c:0] carrier = 1'd0;           // portante: dente di sega

// Creazione della portante
always @(posedge clk)
begin
     carrier <= carrier + 1'b1;
end

// Creazione del duty-cycle
always @(negedge carrier[c])
begin
     duty_cycle <= duty_cycle + 1'b1;
end

assign LED = duty_cycle > carrier[c:c-d];

endmodule
```

L'esecuzione di questo codice porta ad illuminare gradualmente il LED0. Per come è concepito il programma si ha che:

- la frequenza di periodo di switching dipenda dalla carrier e dal clock e sarà pari a: $T = \frac{2^{c+1}}{50 \cdot 10^6} = 655.36 \mu s$
- siccome la triangolare si azzera ogni $T$, la frequenza di switching è semplicemente $f_{SW} = \frac{1}{T} = 1525.88 Hz$
- il led tende ad illuminarsi partendo dalla condizione di spendo, raggiunge il massimo dell'illuminazione e si spegne nuovamente in modo istantaneo.

Per evitare che il led si spenga in modo istantaneo si può o agire sulla carrier o sul duty-cycle: uno dei due, raggiunto il suo massimo, deve ridursi gradualmente. E' chiaro che chi deve ridurre il suo valore deve essere la carrier perchè il duty-cycle nella pratica è dato da una sorgente esterna (tipicamente un regolatore).

L'esempio appena visto serve solo per capire il principio di funzionamento della PWM. Nel paragrafo successivo si realizzerà un esempio un po' più realistico che si scontra con i problemi tipici della progettazione di una PWM, come:

- cosa occorre fare per realizzare una frequenza di switching desiderata? (es: 1kHz)
- è meglio mantenere stabile il numero di bit e variare i clock o viceversa?
- è quindi conveniente lavorare in potenze di 2 o no?

### 4.2.2 PWM

L'esempio che segue può sembrare complesso, ma in realtà ciò che conta fare delle scelte per risolvere delle specifiche di progetto, che potrebbero essere queste:

- Ottenere una frequenza di 2kHz
- La frequenza potrebbe essere settata in modo differente a valori diversi dal precedente (600Hz, 1kHz, 4kHz, ecc)
- La risoluzione in tensione deve essere almeno di 1/1000 alle varie frequenze

Dall'esempio visto al paragrafo 4.2.1 sembra che lavorare in potenza di 2 sia la cosa più pratica. Questo è effettivamente vero, ma è chiaro che non si può creare una qualsivoglia frequenza con una potenza di 2 a meno da non ritoccare il clock; il clock della DE0-Nano infatti è un 50MHz e quindi ottenere un clock da un PLL che sia esattamente una potenza di 2 potrebbe non essere così semplice. Una modifica hardware per mettere un clock da 32768kHz non è ovviamente praticabile.

Quindi ipotizzando che la soluzione a "potenza di 2" sia la migliore, allora l'alternativa è variare lo step che non potrà più essere di un solo bit, ma dovrà essere normalizzato e quindi "spalmato su più bit" per tenera alta la precisione e ridurre conseguentemente l'errore di integrazione. Questo vuol dire che occorre fare alcune scelte arbitrarie. A titolo di esempio si considerino le seguenti:

- lo step di incremento della carrier viene definito su una base di 16 bit.
- in un certo senso lo step rappresenta i decimali. La carrier allora dovrà essere di altri 16 bit per un totale di 32 bit.
- per semplicità anche il duty-cycle dovrà essere un valore a 32 bit.
- la carrier deve essere *signed*. Questo forse complicherà un po' le cose dal punto di vista implementativo, ma dal punto di vista logico dovrebbe essere un po' più chiaro.
- chiaramente anche il duty-cycle sarà un valore *signed*.
- la carrier sarà una triangolare pura, quindi quando raggiungerà il valore massimo, ricomincerà a scendere fino a raggiungere il valore minimo.

**Nota:** se la portante (carrier) è una triangolare, allora la frequenza di switching ($f_{SW}$) è il tempo che intercorre tra due vertici. Nei sistemi di controllo, se possibile, è bene che il douty-cycle vari ad ogni vertice della triangolare, ossia due volte in un intervallo di switching. D'ora in avanti quindi si intende il tempo di campionamento (o di sample) come la metà del tempo di switching $T_s = \frac{1}{2f_{sw}}$ e chiaramente la frequenza di sample sarà $f_s = 2f_{SW}$.

Da queste prime specifiche si capisce che la carrier è un numero che deve essere compreso tra $-2^{31} \div 2^{31}$. Quindi quando il duty-cycle è pari a 0 il valore reale in uscita sarà del 50%.

**Definizione dello step** fissato il clock a 50MHz, la carrier deve raggiunge il valore $2^{31}$ dopo $500\mu s$ e tornare a 0 dei successivi 500 $\mu s$ per avere una frequenza di 1kHz. Quindi lo step dovrà essere di:

$$\frac{2^{31}}{500 \cdot 10^{-6} \cdot 50 \cdot 10^{6}} \backsimeq 171799$$

ovviamente questo è un valore approssimato che non porta ad avere perfettamente 1kHz, ma l'errore è comunque molto contenuto e quindi trascurabile.

**Definizione del modulo PWM** Il modulo PWM non è particolarmente importante. Viene riportato solo per completezza. Di fato è un semplice PWM che permette, grazie al segnale `syncEnable`, di effettuare una sola commutazione tra due vertici della triangolare rispettando così la frequenza di switching richiesta anche a fronte di una variazione del duty-cycle più rapida.

```
module PWM
(   input clk,                          // clock
    input syncEnable,                   // Abilitazione per la commutazione
    input signed [31:0] mod,            // modulante o duty-cycle da applicare
    input signed [31:0] carrier,        // portante
    output P                            // comando P secco
);

reg CommEnable = 0;                     // Commutation Allowed
reg C = 1'b0;                           // comandi della PWM
reg Cm = 1'b0;                          // mirror comando PWM

always @(posedge syncEnable or posedge clk)
begin
    if(syncEnable)
    begin
        Cm = C;                 // memorizzo lo stato della PWM
        CommEnable = 1'b1;      // riabilito la commutazione
    end
    else
    if(Cm ^ C)
        CommEnable = 1'b0;      // disabilito la commutazione
end

// Generazione della commutazione pura senza tempo morto
always @(posedge clk)
begin
    if(CommEnable)
        C = ($signed(mod) > $signed(carrier));
end

assign P = C;

endmodule
```

**Costruzione della carrier (triangolare)** se la carrier non fosse pensata per essere realizzata come potenza di 2, sarebbe relativamente complicato gestirla perchè occorrerebbe tenere conto della pendenza positiva e negativa per sommare o sottrarre lo step scelto. Inoltre occorrerebbe gestire l'overflow oltre il valore massimo e minimo della triangolare.

La carrier pensata in potenza di 2 invece è comoda perchè la si può costruire con semplici operazioni logiche e anche i vari bit hanno un significato. In particolare si ha che:

- il bit 31 del contatore che genererà la carrier (`cnt`) rappresenterà anche la pendenza della carrier stessa
- `XOR` (`^`) del contatore con il suo bit più significativo permette di invertire il suo andamento creando la triangolare. Tale triangolare è però un numero che varia tra 0 e $2^{31} - 1$.
- Per centrare il valore occorre quindi sottrarre la metà, ossia $2^{30} - 1$.
- Per poi tornare a lavorare a 32 bit occorre poi shiftare a sinistra in modo aritmetico di 1 bit. Tutto questo porta all'istruzione seguente: `carrier = (((cnt ^ {32{cnt[31]}}) - {30{1'b1}}) <<< 1'd1);`

**Definizione del duty-cycle per 8 LED** Giusto per complicare un po' l'esempio, il codice seguente riporta anche come far variare su tutti gli 8 LED la PWM. Di fatto viene fatta una inizializzazione di 8 contatori precaricati con 1/8 di periodo ciascuno.

Le cose più importanti da notare sono:

1. `sync`: questo segnale non è altro che un impulso che viene generato ad ogni vertice della portante. Tipicamente questo è un segnale di sincronismo che viene o generato dalla FPGA oppure dal DSP e serve per tenere sincronizzati questi due dispositivi. In questo caso viene generato solo dalla FPGA e passato al modulo `pwm`.
2. Il ciclo `for` utilizzato genera effettivamente del codice. Infatti è impossibile modificare l'indice del ciclo, ma si può solo utilizzarlo.
3. Sono state invocate 8 istanze del modulo `pwm` a mano. Se fossero state molte di più sarebbe stato un problema.

```
module main(
    input       clk,            // clock da 25Mz, quindi a 50MHz viaggerà a 2kHz la fSW
    output  [7:0] LED);         // comando poritivo

reg signed  [31:0] dutyTmp [7:0];
reg signed  [31:0] duty [7:0];
reg [31:0]  cnt = 0;            // contatore di base da cui generare la carrier
reg slope = 1'b1;
reg signed [31:0] carrier;      // Va gestito per forza come reg e non come wire

wire sync;              // Equivalente di un cmando di sync da un uC/DSP. Ora lo genero io.

integer i;

initial
begin
    for (i=0; i<8; i = i+1)
    begin
        dutyTmp[i] <= 32'd536870912 * i;
        duty[i] <= 0;
    end
end

always @(posedge clk)
begin   : Counter
    // lo metto qui con l'"=" per essere certo di passare un ciclo.
    slope = cnt[31];

    // step: 171799 = 1kHz @ clk=25M; cambiando questo si cambia la frequenza di switching
    cnt = cnt + 32'd171799;

    // NOTA: si perde comunque il bit meno significativo!
    carrier = (((cnt ^ {32{cnt[31]}}) - {30{1'b1}}) <<< 1'd1);
end

assign sync = slope ^ cnt[31];

// Generazione del DutyCyle
always @(posedge clk)
begin
    for (i=0; i<8; i = i+1)
```

```
    begin
        dutyTmp[i] <= dutyTmp[i] + 32'd54;
        duty[i] = (((dutyTmp[i] ^ {32{dutyTmp[i][31]}}) - {30{1'b1}}) <<< 1'd1);
    end
end

PWM pwm1
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[0])),
    .carrier($signed(carrier)),
    .P(LED[0]));

PWM pwm2
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[1])),
    .carrier($signed(carrier)),
    .P(LED[1]));

PWM pwm3
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[2])),
    .carrier($signed(carrier)),
    .P(LED[2]));

PWM pwm4
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[3])),
    .carrier($signed(carrier)),
    .P(LED[3]));

PWM pwm5
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[4])),
    .carrier($signed(carrier)),
    .P(LED[4]));

PWM pwm6
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[5])),
    .carrier($signed(carrier)),
    .P(LED[5]));

PWM pwm7
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[6])),
    .carrier($signed(carrier)),
    .P(LED[6]));

PWM pwm8
(   .clk(clk),
    .syncEnable(sync),
    .mod($signed(duty[7])),
    .carrier($signed(carrier)),
    .P(LED[7]));

endmodule
```

**Generate statement** Come detto al paragrafo precedente, realizzare 8 istanze del modulo PWM a mano. Nell'esempio sono riportati dei cicli `for` che operano direttamente sulle variabili come indici. E' possibile fare la stessa cosa con i blocchi `generate` che permettono di istanziare moduli, assign statement, always statement, ecc

in modo statico. In altre parole il codice viene "esploso" e le varie istanze sono i nomi indicate dopo il `begin`; per questa ragione nei blocchi `generate` il nome dell'istanza dopo il `begin` è obbligatorio.

Altra cosa fondamentale è che le variabili utilizzate dai blocchi `generate` devono essere variabili di tipo `genvar`; queste variabili vengono cancellate dopo la generazione del codice e non possono essere utilizzate al di fuori di `generate`.

Il codice che segue è esattamente quello riportato a pagina 25, ma estremamente più compatto.

```
module main(
    input       clk,
    output  [7:0] LED
);

reg signed  [31:0] dutyTmp [7:0];
reg signed  [31:0] duty [7:0];
reg [31:0]  cnt = 0;
reg slope = 1'b1;
reg signed [31:0] carrier;

wire sync;

integer i;

initial
begin
    for (i=0; i<8; i = i+1)
    begin
        dutyTmp[i] <= 32'd536870912 * i;
        duty[i] <= 0;
    end
end

always @(posedge clk)
begin   : Counter
    slope = cnt[31];
    cnt = cnt + 32'd171799;
    carrier = (((cnt ^ {32{cnt[31]}}) - {30{1'b1}}) <<< 1'd1);
end

assign sync = slope ^ cnt[31];

// Generazione del DutyCyle
always @(posedge clk)
begin
    for (i=0; i<8; i = i+1)
    begin
        dutyTmp[i] <= dutyTmp[i] + 32'd54;
        duty[i] = (((dutyTmp[i] ^ {32{dutyTmp[i][31]}}) - {30{1'b1}}) <<< 1'd1);
    end
end

// Generazione di tutte le 8 PWM
genvar j;
generate
for (j = 0; j < 8; j = j + 1) begin : pwmInstance
    PWM pwmX
    (   .clk(clk),
        .syncEnable(sync),
        .mod($signed(duty[j])),
        .carrier($signed(carrier)),
        .P(LED[j]));
end endgenerate

endmodule
```

Compilando il modulo in questo modo verranno quindi generate 8 istanza di PWM chiamate `pwmInstance`$[i]$ come si vede dalla figura 4.3 che riporta la gerarchia dei moduli chiamati.

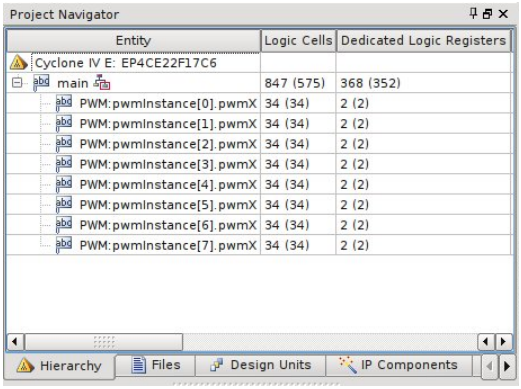



Figura 4.3: Istanze del modulo PWM

## 4.3 Memoria Interna

L'FPGA come detto può avere all'interno alcune unità precablate grazie alle quali è possibile avere alcune funzionalità senza dover ricorrere all'utilizzo di risorse logiche del dispositivo. Esattamente come per i moltiplicatori visti in precedenza (4.1) anche per la memoria è possibile utilizzare o le risorse logiche o risorse interne. Dai compilation report visti fino ad ora (figure 2.1, 2.8), si può notare che questa FPGA possiede 608256 bit di memoria utilizzabile come RAM o ROM.

Tipicamente questa memoria viene usata per fare da cache nei processori come il NIOS II [7], ma può essere indirizzata anche come semplice memoria interna, per fare una dual-port RAM, ecc. La frequenza di accesso a questa memoria è pari alla frequenza massima consentita dal PLL, ossia 315MHz; ciò implica che lo speed grade del dispositivo impatta anche sulle performance di questa unità. Per maggiori informazioni consultare [9].

**Specifica** L'esempio che segue prevede di realizzare una PWM che realizzi una una forma d'onda sinusoidale. La sinusoide sarà mappata in memoria, ma non completamente. Per risparmiare infatti, il seno verrà mappato solo per un quarto, sufficiente per realizzarlo per qualsiasi valore dell'angolo. I passi da eseguire sono i seguenti:

- Creare un PLL (Megafunction) per ottenere due frequenze: 25MHz e 12.5 MHz: questo perchè questo esempio era stato sviluppato su una Cylone III C8 e per non avere problemi si è preferito mantenerlo identico come timing. Inoltre frequenze più basse hanno alcuni vantaggi sulla risoluzione di alcuni calcoli. Per inserire la Megafunction seguire i passi descritti al capitolo 3 e abilitare sia l'uscita `c0` che l'uscita `c1`.
- Creare la Megafunction per la RAM. Siccome verrà solo letta si può scegliere anche la ROM e precaricarla con un file .mif il quale non è altro che una tabella che può essere scritta in *calc*, *Matlab* o con cosa si preferisce e si possono copiare i dati con un semplice copia-incolla.
- Realizzare quindi una PWM come fatto negli esempi recedenti
- Aggiungere il DeadTime per avere una PWM più realistica: un inverter reale infatti deve avere un certo "tempo morto" tra l'accensione di un IGBT e l'altro per permettere ai diodi di clamp di ripristinare la barriera di potenziale. In questo esempio, in quanto ideale, si potrebbe anche evitare.
- Creare un angolo per accedere alla RAM. Per le scelte fatte l'angolo è un numero a 32 bit unsigned. Analogamente per quanto fatto per creare la triangolare, anche qui si ricorre all'XOR, ma non sull'ultimo bit, bensì sul penultimo. Questo creerà una triangolare di frequenza doppia rispetto alla variazione dell'angolo come in figura. Questo perchè il valore ottenuto ci permetterà di avere un valore dell'angolo variabile tra 0-90° dove è stato definito il seno. Il valore del seno dovrà poi tenere conto del segno in funzione del vero valore dell'angolo a 32 bit.

**Implementazione** Creazione del file MIF:

- Selezionare un nuovo file: File▷New e selezionare Memory Initialization File

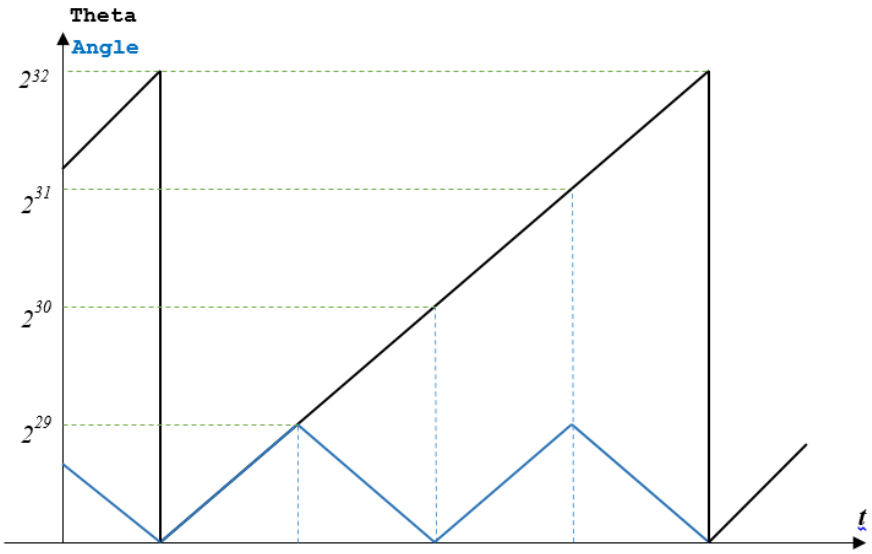



Figura 4.4: Andamento di `Theta` e `Angle`

- Alla richiesta di quante word: inserire $2^{14}$, ossia 16384; questo è il valore del bus indirizzi che è un bus a 14 bit
- Impostare word size a 15 perchè si vuole un valore del seno a 16bit con segno, quindi il seno varia tra 0 e $2^{15} - 1$ nella semi-onda positiva, mentre in quella negativa andrà gestito il segno e quindi si avrà un bit in più; è l'equivalente del bus dati.
- Caricare il file creando con un foglio elettronico o un qualunque altro programma, un seno variabile tra 0 e 90° moltiplicato per $2^{15}$. Copiare il risultato nel file .mif e salvarlo. Dovrebbe apparire qualche cosa come in figura 4.5.

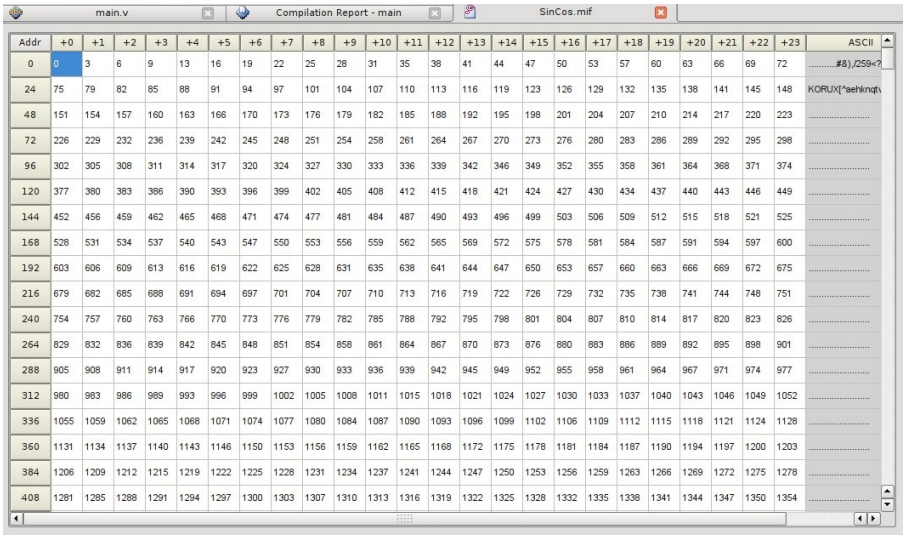



Figura 4.5: File MIF

Aggiungere la Megafunction:

- Avviare il Magafunction Wizard: Tools ▷ MegaWizard Plugin Manager.
- Selezionare ROM-1 port (aiutarsi con i filti scrivendo ROM).

- Configurare la rom come in figura 4.6.
- Aggiungere il file .mif (figura 4.7) che deve essere stato creato in precedenza e formattato adeguatamente.
- Procedere e completare la configurazione.

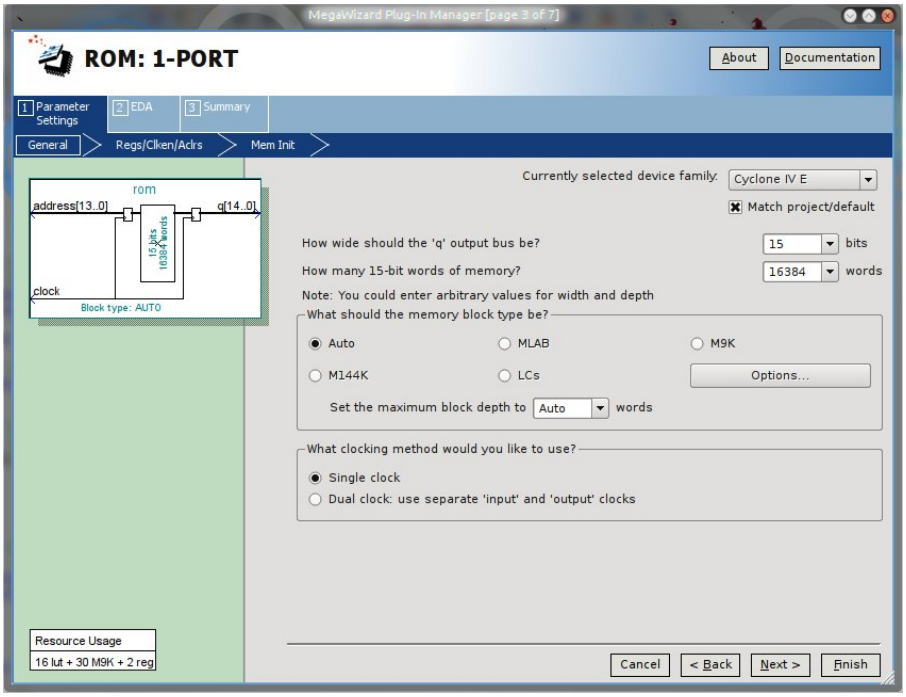



Figura 4.6: ROM 1-Port - Address & Data configuration

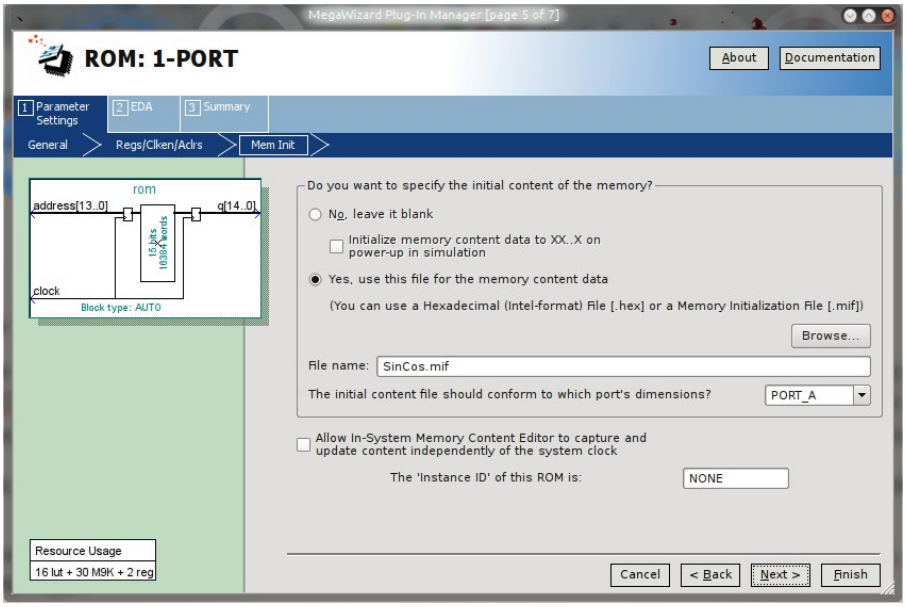



Figura 4.7: ROM 1-Port - Preconfigurazione con file .mif

Il codice prevede quindi la creazione di un bus a 14 bit, ossia `angle`. Questo angolo è derivato da `theta` il quale è il vero angolo che varierà tra 0 e $2\pi$. Ovviamente la rappresentazione in virgola fissa implica che, scelta la base a 32 bit, l'angolo vari tra 0 e $2^{32} - 1$.

**Nota:** gli angoli sono le grandezze tipiche da mantenere a valori interi a potenze di 2; questo perchè angoli e numeri interi sono entrambe funzioni cicliche e quindi non occorre gestirne l'overflow come invece occorrerebbe fare con rappresentazioni floating point.

La creazione della variabile `angle` è, come detto, realizzata basandosi sul penultimo bit di theta prendendo solo 14 bit. Questo porta alla seguente istruzione: `angle = theta[29:16] ^ {14{theta[30]}}`

Il codice è quindi il seguente:

```
module main
(    input   clk1,            // clock 50MHz
     output  [7:0] LED,       // led
     output  P,               // PWM +: PIN_P11 (GPIO_110) pin 10 GPIO1
     output  N);              // PWM -: PIN_T11 (GPIO_17) pin 15 GPIO1

// Definizione dei clock
wire     clk2;                    // clock a 25MHz         50/2
wire     clk4;                    // clock a 12.5MHz  50/4

wire     sync;                    // segnale di sync per evitare le commutazioni multiple

wire signed [14:0] sin; // seno a 15 bit in uscita dalla RAM
wire [13:0] angle;      // angolo ridottto a 0-pi/2 (è un'altra triangolare di fatto)
reg signed [15:0] duty; // Duty-Cycle
reg [31:0] theta = 0;   // angolo effettivo 0-2pi da cui otterremo "angle"

wire     p;                       // PWM positiva senza dead time
reg slope;
reg [31:0] carrier; // triangolare per realizzare la PWM
reg [31:0] cnt;          // contatore da cui ottenere la carrier


// Creazione dei due clock aggiuntivi
pll pll0
(    .inclk0 ( clk1 ),
     .c0 ( clk2 ),
     .c1 ( clk4 ));


// Generazione dell'angolo: più il clock è lento, maggiore è la risoluzione numerica
// dello step, ma minore è la risoluzione dell'angolo.
always @(posedge clk2)
begin    : AngleGeneration
     theta <= theta + 30'd8590; //step 8590 per avere 50Hz con un clock di 25MHz
end

// L'angolo è normalizzato a 14 bit ed è una triangolare
assign angle = theta[29:16] ^ {14{theta[30]}};


// Creazione del seno ridotto: è di fatto 1/4 di seno.
// Si entra con Angle e si esce con 1/4 di seno sempre positivo.
rom rom0
(    .address ( angle ),
     .clock ( clk2 ),
     .q ( sin ) );


// Generazione del Duty-Cycle a 16bit con segno
// -----------------------------------------
always @(negedge clk4)  // il clock deve essere più lento di quello che genera theta
begin
     if(theta[31])
          duty <= -{1'b0, sin[14:0]};
     else
          duty <= +{1'b0, sin[14:0]};
end


// Generazione della Carrier con clock a 50MHz per ottenere 2kHz di switching
// ---------------------------------------------------------------------------
```

```
always @(posedge clk1)
begin   : Counter
    slope = cnt[31];                    // con "=" si è certi di passare un ciclo.
    cnt = cnt + 32'd171799;             // step: 171799 = 2kHz @ clk=50M;
    //cnt = cnt + 32'd858993;           // step: 858993 = 10kHz @ clk=50M;
    carrier = (((cnt ^ {32{cnt[31]}}) - {30{1'b1}}) <<< 1'd1);
end

assign sync = slope ^ cnt[31];


// Generazione di una gamba PWM
// ----------------------------
PWM pwm0
(   .clk(clk1),
    .syncEnable(sync),
    .mod({duty, {16{1'b0}}}),   // modulante a 32 bit signed
    .carrier(carrier),
    .P(p) );

// Applicazione del tempo morto: in questo esempio non servirebbe,
// ma così si realizza un caso più realistico.
DeadTime
(   .clk(clk1),
    .dt(10'd250),       // 250@50MHz = 5us di delay, 50 = 1us
    .p(p), .n(!p),
    .P(P), .N(N) );

// Assegnazione dei led
assign LED = {{6{1'b0}}, P, N};

endmodule
```

**Test in Hardware** Per effettuare il test di questo programma occorre configurare i pin P e N come indicato nel software e in figura 4.8. Questo perchè così sarà possibile collegare un filtro RC e verificare che effettivamente si sta creando una sinusoide a 50Hz come richiestp.

Il filtro RC va connesso tra il pin 10 del connettore GPIO1 e la massa; il condensatore va a massa ovviamente. Il filtro utilizzato prevede una resistenza da $1k\Omega$ e una capacità di $4.7\mu F$ che porta ad avere una banda passante di 212Hz circa. La funzione di trasferimento nel dominio della frequenza è:

$$G(s) = \frac{1}{s \cdot R \cdot C + 1}$$

Questa funzione tornerà utile in seguito per verificare il comportamento del filtro e avere delle controprove relativamente alle frequenze di uscita.

Il primo test è una verifica visiva: in figura 4.9 si vede l'andamento del duty-cycle preso sui due pin di uscita `P` e `N` che varia con una frequenza che già si intuisce circa 50Hz. Si nota subito però che vi sono dei "buchi" nelle due PWM quando uno dei pin rimane acceso quasi totalmente: in questo momento l'altro pin rimane spento per un certo tempo. Ciò è dovuto al tempo morto che, essendo pari a $5\mu s$ fa si che di fatto non vi è abbastanza tempo per accendere il pin (figura 4.9-b).

Il secondo test prevede invece di verificare la sinusoide: per farlo si preleva il segnale in uscita sul pin P e la tensione sul condensatore. Si noti che l'FPGA esce con una tensione che varia tra 0 e 3.3V; questo implica che se la tensione viene filtrata, il suo valore medio sarà pari a 1.65V. Quindi per avere una migliore risoluzione, il canale 1 è stato triggerato in AC in modo da avere un valore medio nullo. In figura 4.10 è riportata la forma d'onda filtrata. La frequenza letta è mediamente di 50Hz e la forma d'onda sembra leggermente attenuata e sfasata rispetto alla PWM. Questo è corretto perchè se si calcola l'attenuazione dalla funzione di trasferimento si ha:

$$G(2\pi 50) = \frac{1}{\sqrt{(2\pi 50 \cdot R \cdot C)^2 + 1}} \backsimeq 0,561$$

che quindi genera una tensione picco-picco di:

$$v_{pp} = G(2\pi 50) \cdot V \backsimeq 1.85$$

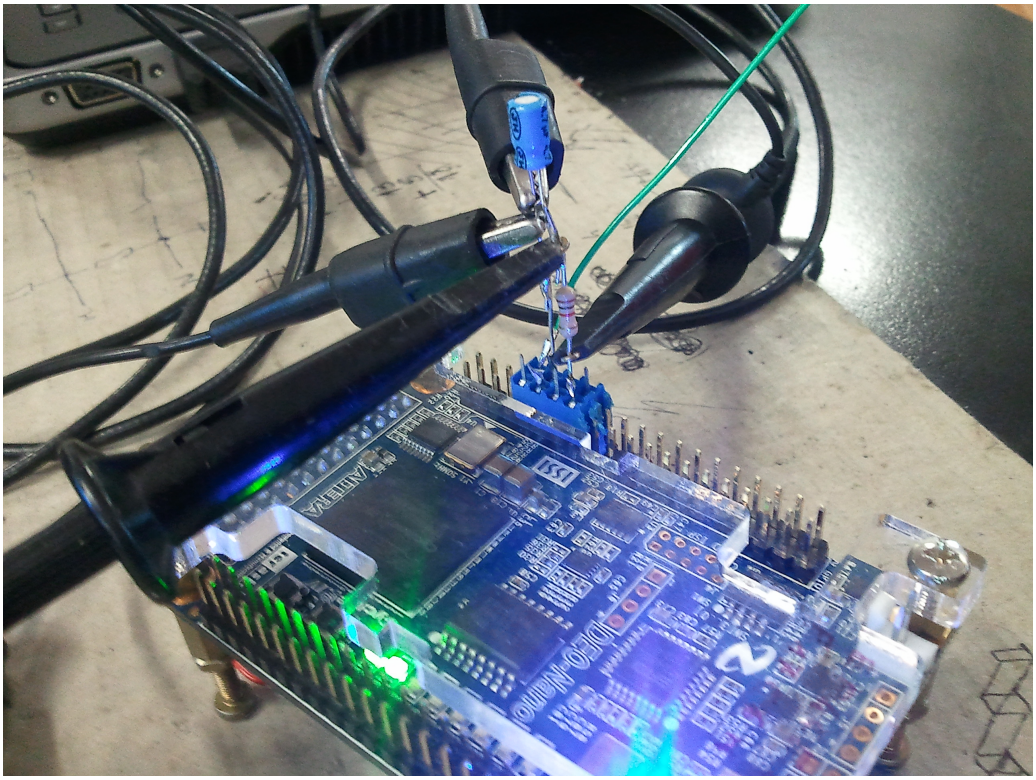
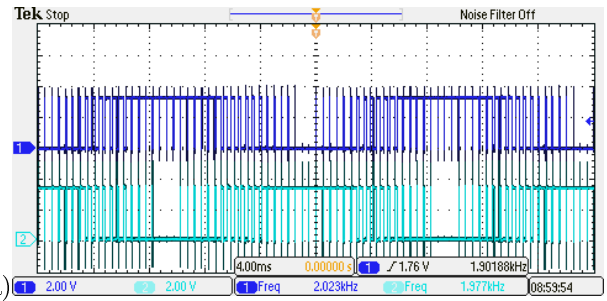


Figura 4.8: Connessione dell'oscilloscopio

a)




b)

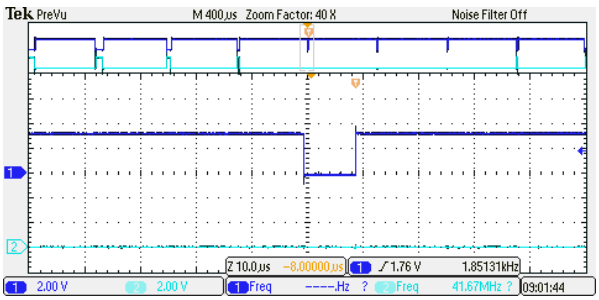



Figura 4.9: Verifica PWM a 50 Hz - a) canale CH1:P e CH2:N - b) zoom sulla cresta della sinusoide

il che è analogo a quanto misurato in 4.10-a.

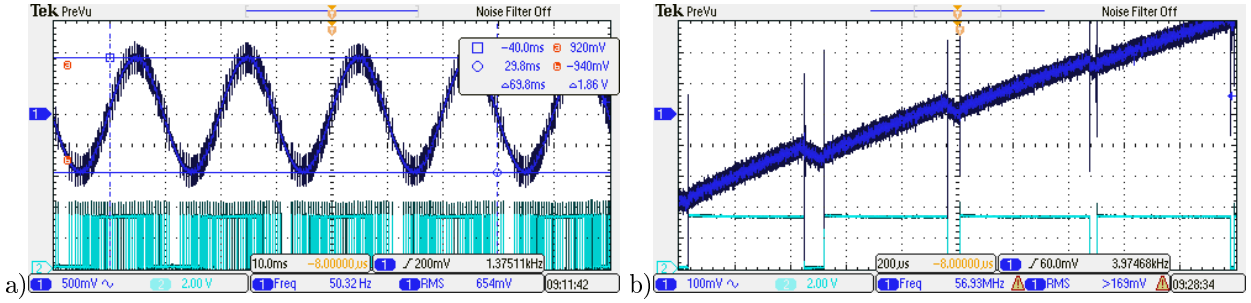



Figura 4.10: Verifica PWM a 50 Hz - a) canale CH1:Vc e CH2:P - b) zoom in un punto di commutazione

# 5 Virtual JTAG, TCL & Qt

Il capitolo che segue è fortemente orientato alle FPGA ALTERA in quanto verrà utilizzata una IP distribuita solo con il sofware Quartus II, ossia Virtual JTAG[7] [4]. La comunicazione con questo dispositivo avviene direttamente con l'USBBlaster, ma occorre necessariamente:

- un server TCL che permetta, tramite opportune chiamate, di dialogare con il vJTAG: il server è basato sull'esempio descritto in [6]
- un client che si colleghi al server TCL: per ragioni di semplicità e portabilità è stato scelto di scriverlo in Qt, ma si può scegliere qualunque altro linguaggio

Ovviamente chi è esperto di TCL può editare lo script che esegue il server per inviare i comandi senza sfruttare il socket TCP vero e proprio.

**JTAG e Virtual JTAG** Il JTAG[8] è un dispositivo che, tramite un opportuno protocollo, permette di prendere il controllo di un dispositivo integrato attraverso alcuni pin dedicati. Di fatto questo è un dispositivo seriale che permette di mettere in catena anche più dispositivi comunizzando alcuni segnali come il clock e serializzando alcuni comandi di input e output. Tramite il JTAG è possibile programmare un dispositivo o semplicemente interrogarlo per esempio per attività di debug.

Il JTAG è già presente sulla scheda DE0-Nano ed è l'USBBlaster e serve come noto per la programmazione. Però, una volta creato il nostro circuito, il canale JTAG non viene più utilizzato a meno che in si aggiungano uno o più JTAG virtuali sfruttando l'IP fornita da ALTERA.

Uno degli esempi tipici del vJTAG è per il debug e programmazione dei microprocessori NIOS II. In figura [6]-Figure5 è riportato lo schema funzionale del circuito. Non si vuole ora entrare nel merito di tutti i pin e delle varie funzioni del JTAG; ciò che basta sapere è il significato dei seguenti segnali:

`TCK` è il clock generato. Da alcuni test (non riportati in questo documento) il clock misurato aveva una frequenza di 4MHz.

`TDI` Transfer Data IN: è un segnale che viene serializzato in funzione del clock `TCK`. Di fatto sono i dati che possono provenire dal sistema di debug; nel nostro caso saranno i dati inviati dal client QT al server TCL e da quest'ultimo all'FPGA. La lunghezza in bit di questi dati può essere qualsivoglia, ma si vedrà che di fatto saranno multipli di 4 bit.

`TDO` Transfer Data OUT: è un segnale che viene serializzato in funzione del clock `TCK`. Di fatto sono i dati che l'FPGA invia verso sistema di debug. Questi dati potranno essere eventualmente scartati dal sistema. La lunghezza di questi dati è pari a quella del `TDI` (in bit).

`IR in` Input Instruction Register: questo registro appare all'FPGA come un bus che può essere al massimo di 26 bit. Viene inviato dal programma di debug.

[7] Le altre case hanno normalmente la loro versione di questo tipo di dispositivo
[8] Standard IEEE 1149.1

`IR out` Output Instruction Register: equivalente dell'`IR in`, ma in uscita fall'FPGA verso il server. Può essere ignorato dal programma.

`DR` Sono i dati che verranno poi serializzati sulla linea `TDI`.

## 5.1 Inserimento del Virtual JTAG

L'esempio che si vuole realizzare prevede di:

- accendere i led tramite il registro IR da 8 bit
- far tornare all'interfaccia un valore a 8 bit che incrementa in funzione dalla pressione del tasto KEY0
- verificare il valore dei dati ricevuti premendo KEY1 (8 bit)

Una volta creato il progetto come indicato al capitolo 1, avviare il Megafunction Wizard (come al paragrafo 2.3) e scrivere "*jtag*" nella parte relativa al filtro dei nomi; scegliere quindi "Virtual JTAG" e indicare un nome per le istanze, per esempio `vjtag` come indicato in figura 5.1.

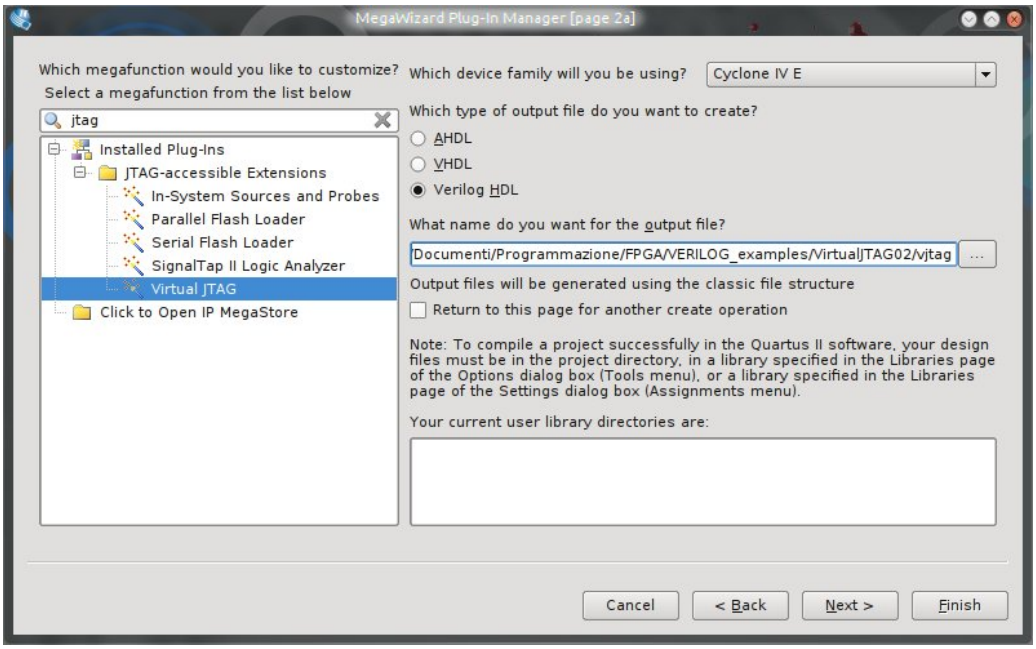



Figura 5.1: Virtual JTAG: crazione della Megafunction

La configurazione di base a titolo di esempio prevede un IR pari a 8 bit e settare a Manual = 0 l'istanza (figura 5.2). Le pagine seguenti permettono di realizzare degli stimoli in automatico. Lasciare tutto inalterato. Proseguendo si nota che il wizard indica che la libreria da usare è *altera_mf* (ossia ALTERA MegaFunction). Questa libreria andrà inclusa in simulazione e dovrà essere *altera_mf_ver* in quanto il software è scritto in Verilog.

A questo punto occorre inserire il JTAG all'interno dell'esempio. Si noti che tipicamente occorre creare un interprete comandi per gestire i comandi `IR` e un data-shifter che trasferisca, per esempio in un registro, i dati dal `TDI` ed un altro data-shifter che trasferisca i dati in uscita sul `TDO`. Tutto sincronizzato con il clock del JTAG `TCK`.

Un altro registro molto importante è il `vsdr`, ossia il *virtual state shift data register*: quando questo segnale è alto, il vJTAG sta trasferendo i dati.

Il codice del modulo principale è il seguente:

```
module main(
    input   clk,
    input   key0,
```

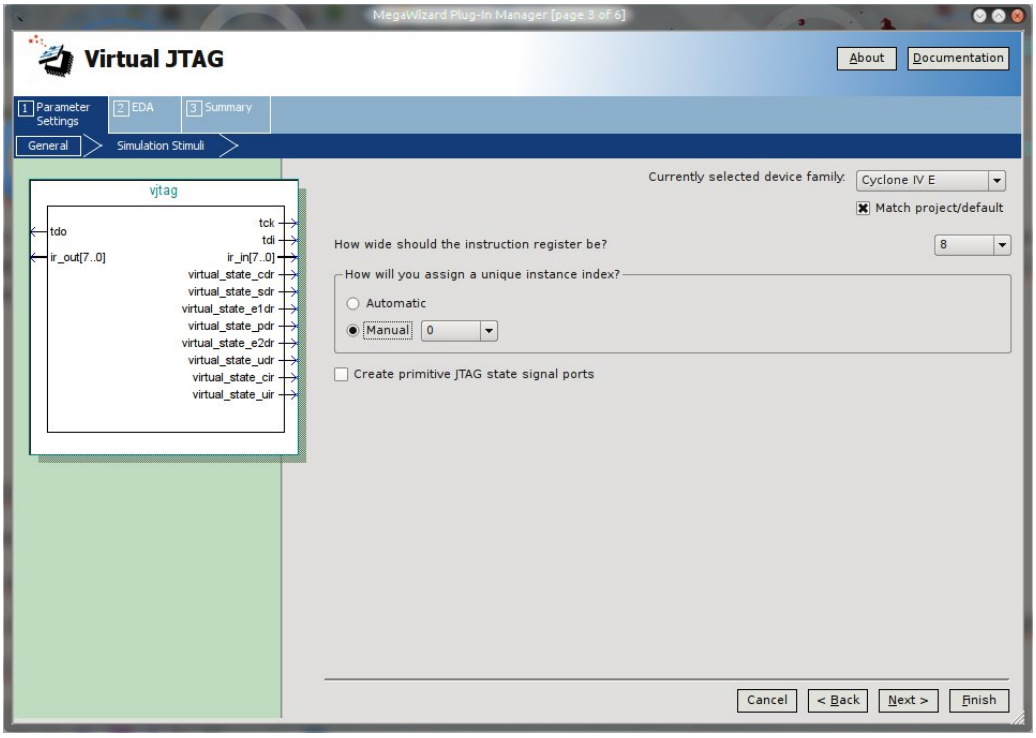



Figura 5.2: Virtual JTAG - Config Step 1

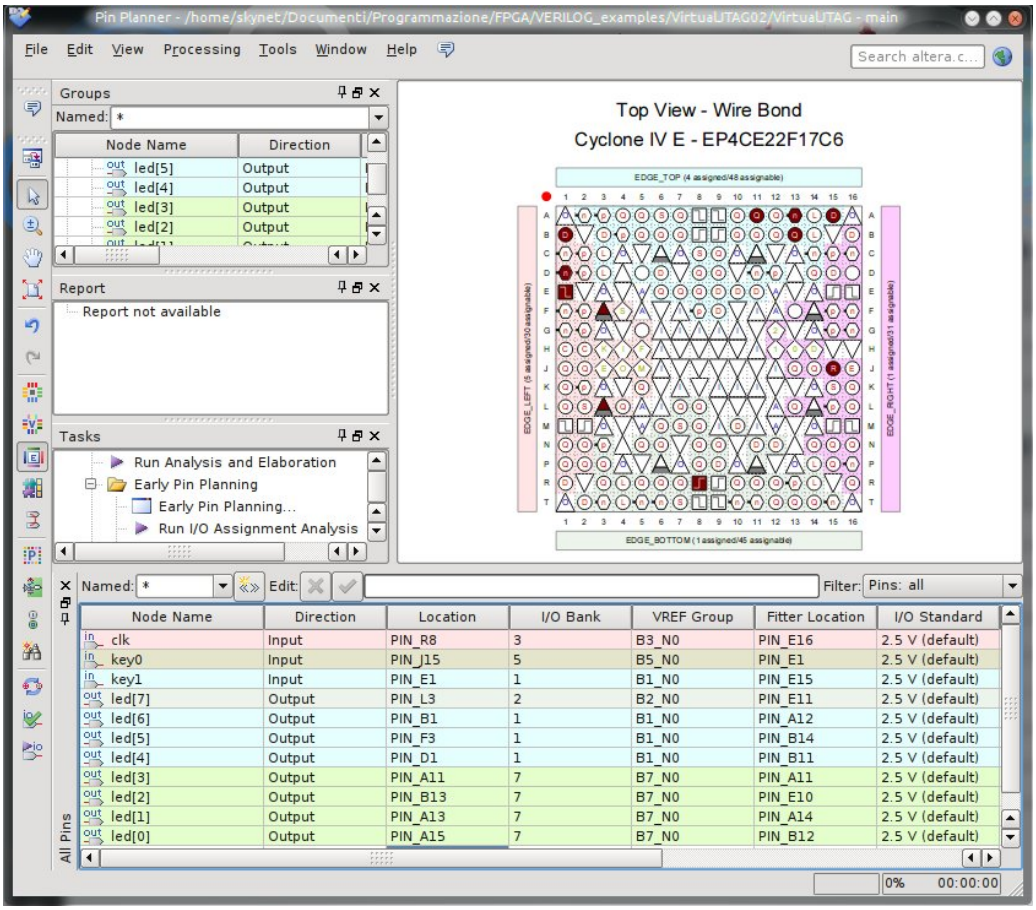



Figura 5.3: Virtual JTAG - Pin assignment

```
    input   key1,
    output  [SIZE-1:0]led);

parameter SIZE = 8;

wire TDo, TDi;              // Transfer data In/Out
wire [SIZE-1:0] IRi, IRo;   // Instruction register In/Out
wire tck;                   // clock virtuale del JTAG
wire vsdr;                  // Virtual State Data register
wire [SIZE-1:0] DataIn;     // registro 0 e registro 1 che verranno moltiplicati

reg [SIZE-1:0]  DataOut = 8'd0; // registro dati in uscita


// Istanza del JTAG
vjtag vJTAG(
    .ir_out(IRo),       // IR in uscita, non considerare qui
    .tdo(TDo),          // Transfer data in uscita
    .ir_in(IRi),        // IR in entrata inviato da PC
    .tdi(TDi),          // Transfer data in ingresso
    .tck(tck),          // clock del JTAG
    .virtual_state_sdr(vsdr));  // Stato del trasferimento dati

// Istanza dello shifter per caricare i dati in ingresso (NON i comandi)
DataShifterIN #(SIZE)
DS_in
(   .clk(tck),
    .vsdr(vsdr),
    .tdi(TDi),
    .q(DataIn));

// Istanza dello shifter per i dati in uscita
DataShifterOUT #(SIZE)
DS_out
(   .clk(tck),
    .vsdr(vsdr),
    .data(DataOut),
    .tdo(TDo));

always @(posedge key0)
begin
    DataOut <= DataOut + 1'b1;
end

assign led = key1 ? IRi : DataIn;   // Le KEY sono in logica negata: premuta è 0

endmodule
```

Il codice più importante è pero quello relativo agli shifter dei dati in ingresso e uscita. In particolare `DataShifterIN` prende in ingresso il segnale `TDI` in modo da andare a caricare in un registro i dati che arrivano in modo seriale sincronizzati con il clock del JTAG. `DataShifterOUT` invece ha in ingresso un valore a 8 bit e, tramite uno shift register, preleva i vari bit e li invia ad ogni colpo di clock verso l'uscita, ossia al JTAG. Il codice è il seguente:

```
/** DataShifterIN ********
 ************************/
module DataShifterIN
(   input   clk,
    input   vsdr,
    input   tdi,
    output  [SIZE-1:0] q);

parameter SIZE = 8;

reg [SIZE-1:0] outcome = 16'd0;

always @(posedge clk)
```

```
begin
    if(vsdr)
        outcome <= {outcome[SIZE-2:0], tdi};
end

assign q = outcome;

endmodule

/** DataShifterOUT *******
 ************************/
module DataShifterOUT
(   input   clk,
    input   vsdr,
    input   [SIZE-1:0] data,
    output  tdo);

parameter SIZE = 8;
reg [SIZE:0] count; // servirebbe il log2

always @(posedge clk)
begin
    if(vsdr)
        count <= count + 1'd1;
    else
        count <= 8'd0;
end

assign tdo = data[count];

endmodule
```

### 5.1.1 Comunicare con il Virtual JTAG

Ora rimane solo la parte di comunicazione. La soluzione proposta riprende l'esempio descritto in [6] e si basa poi sulla documentazione di riferimento del vJTAG [4] che è ovviamente completa e quindi un po' dispersiva (in quanto vasta).

Di fatto ALTERA mette a disposizione un interprete comandi TCL *quartus_stp* che, oltre ai comandi standard TCL, aggiunge alcuni comandi per la comunicazione con il Virtual JTAG. I due comandi principali che interessano sono i seguenti:

`device_virtual_ir_shift` è il comando che trasferisce il valore dell'*instruction register* dal PC verso l'FPGA. I comandi che vuole sono i seguenti:

- `-instance_index` è il valore dell'istanza; normalmente è 0 in quanto quando è stata aggiunta la Megafunction è stata definita manualmente l'istanza settandola a 0.
- `-ir_value` è il valore dell'Instruction Register da inviare. Il valore deve essere espresso rigorosamente in decimale.
- `-no_captured_ir_value` indica che il programma non restituirà un IR proveniente da FPGA; in altre parole il comando non è interruttivo. Questa opzione può essere omessa e in quel caso il comando è interruttivo e restituirà un IR.

`device_virtual_dr_shift` è il comando per trasferire il registro dati DR che verrà letto tramite il segnale TDI nel software.

- `-dr_value` è il valore del registro DR da inviare. Può essere un valore decimale o esadecimale; in quest'ultimo caso non deve avere "0x" davanti.
- `-length` è la lunghezza dei dati. In linea teorica può essere qualsivoglia compatibilmente con i dati inviati. Di fatto è bene che sia un valore multiplo di 4. Questo valore indica i bit di dati da trasferire.

`-value_in_hex` indica che il valore dei dati che segue `-dr_value` è espresso in esadecimale. Se omesso i dati verranno intesi in base decimale.

`-instance_index` è l'istanza del vJTAG, tipicamente 0.

`-no_captured_ir_value` se omesso (e nei nostri esempi è omesso) attende il trasferimento dati dalla FPGA.

Di seguito è riportato il codice del server TCL senza troppi commenti per ragioni di spazio. Quello che è importante è capirne il funzionamento, ossia l'apertura di un socket in ascolto su una porta dedicata scelta a piacere in attesa di due comandi IR e DR provenienti da un programma esterno (nel nostro caso scritto in Qt). Il programma è il seguente:

```
#This portion of the script is derived from some of the examples from Altera

global usbblaster_name
global test_device
# List all available programming hardwares, and select the USBBlaster.
# (Note: this example assumes only one USBBlaster connected.)
# Programming Hardwares:
foreach hardware_name [get_hardware_names] {
    if { [string match "USB-Blaster*" $hardware_name] } {
        set usbblaster_name $hardware_name
    }
}

puts "\nSelect JTAG chain connected to $usbblaster_name.\n";

# List all devices on the chain, and select the first device on the chain.
foreach device_name [get_device_names -hardware_name $usbblaster_name] {
    if { [string match "@1*" $device_name] } {
        set test_device $device_name
    }
}
puts "\nSelect device: $test_device.\n";

# Open device
proc openport {} {
    global usbblaster_name
        global test_device
    open_device -hardware_name $usbblaster_name -device_name $test_device
}


# Close device.  Just used if communication error occurs
proc closeport { } {
    catch {device_unlock}
    catch {close_device}
}


# Funzione di scrittura dell'"Instruction e Data Register"
# ------------------------------------------------------
proc set_IR {JTAG_cmd sock} {
    variable x

    openport
    device_lock -timeout 10000

    # La stringa deve essere formattata come <IR DR>, esempio: <20 0x0040A1>
    # Occorre quindi separare i campi

    # Calocolo la lunghezza totale
    set len [string length $JTAG_cmd]

    # Determino dove finisce il primo valore e dove comincia il secondo
    set idx [string wordend $JTAG_cmd 0]
    set w1_end [expr $idx-1]
    set w2_begin [expr $idx+3]

    # Determino il comando Instruction Register: non serve conoscere la dimensione
    set IR_cmd [string range $JTAG_cmd 0 $w1_end]
```

```
    #determino il comando Data Register: devo calcolare anche la lunghezza
    set DR_cmd [string range $JTAG_cmd $w2_begin $len]
    set DR_len [expr [string length $DR_cmd]*4]

    device_virtual_ir_shift -instance_index 0 -ir_value $IR_cmd -no_captured_ir_value
    set x [device_virtual_dr_shift -dr_value $DR_cmd -instance_index 0  -length $DR_len -value_in_hex]

    puts $sock $x

    closeport
}

################################## TCP/IP Server ####################################
#Code Dervied from Tcl Developer Exchange - http://www.tcl.tk/about/netserver.html

proc Start_Server {port} {
    set s [socket -server ConnAccept $port]
    puts "Started Socket Server on port - $port"
    vwait forever
}

proc ConnAccept {sock addr port} {
    global conn

    puts "Accept $sock from $addr port $port"
    set conn(addr,$sock) [list $addr $port]

    fconfigure $sock -buffering line
    fileevent $sock readable [list IncomingData $sock]

    puts $sock "\nRichiesta comandi esadecimali <IR> <DR>"
}


proc IncomingData {sock} {
    global conn

    # Check end of file or abnormal connection drop,
    # then write the data to the vJTAG

    if {[eof $sock] || [catch {gets $sock line}]} {
    close $sock
    puts "Close $conn(addr,$sock)"
    unset conn(addr,$sock)
    } else {
        #Before the connection is closed we get an emtpy data transmission.
        #Let's check for it and trap it
        set data_len [string length $line]

        if {$data_len != "0"} then {
            set_IR $line $sock
        }
    }
}

#Start this Server at Port 2540
Start_Server 2540
```

Per quanto riguarda Qt, il programma è relativamente lungo e quindi non verrà riportato, ma sarà distribuito con i sergenti degli esempi; di fatto non fa altro che inviare una stringa formattata in due numeri: il primo a base decimale come comando IR, il secondo a base esadecimale nel formato "0x...". Sarà poi lo script TCL a togliere ciò che non serve.

A questo punto avviare server e client come segue:

- Avviare una shell e spostarsi nella directory dei sorgenti
- eseguire il comando: quartus_stp -t TCL_Server_vJTAG_Test.tcl
- Per il client eseguire da QTCreator o direttamente dalla directory dei sorgenti il programma vJTAG

A questo punto cliccando sul pulsante "Connect" dell'interfaccia Qt si troverà una situazione simile a quella riportata in figura 5.4. Il server TCL, come si vede, ha accettato la connessione dal client.

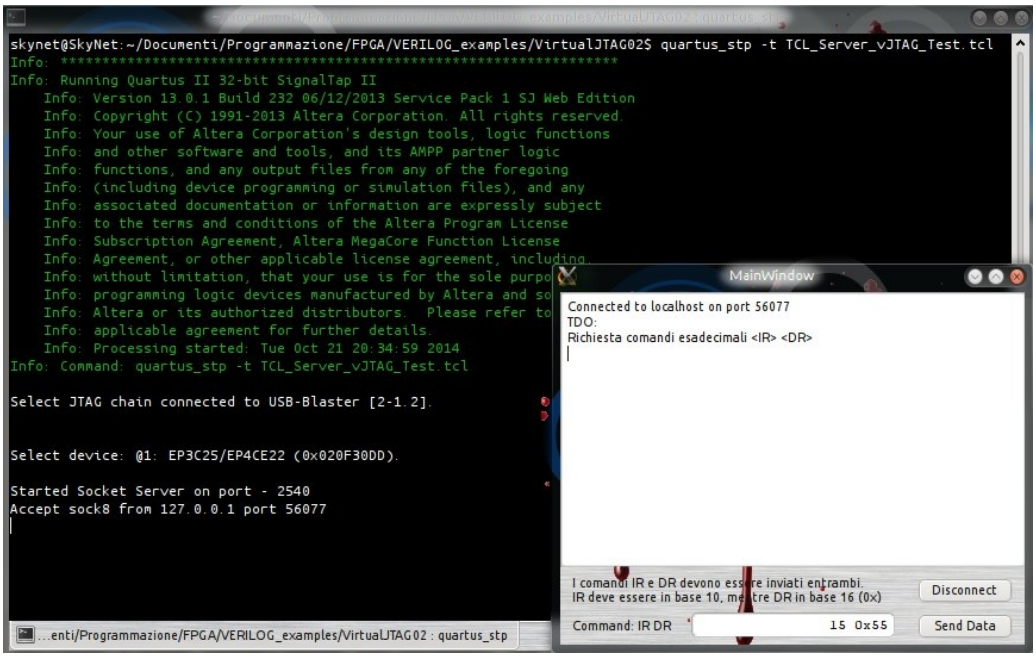



Figura 5.4: Virtual JTAG - Connessione Clietn/Server

Il programma Qt è generico per inviare un qualunque dato di qualunque lunghezza. Per come è concepito il programma in FPGA occorre inviare un numero a 8 bit sia in IR che in DR. Facendolo, i led si dovrebbero accendere rappresentando il numero di IR. Tenendo premuto su KEY1 i led si illumineranno rappresentando il registro DR. Schiacciando poi qualche volta KEY0 e reinviando un dato all'FPGA si vedrà che i dati incrementati dalla pressione di KEY0 verranno letti dall'interfaccia Qt come rappresentato in figura 5.5.

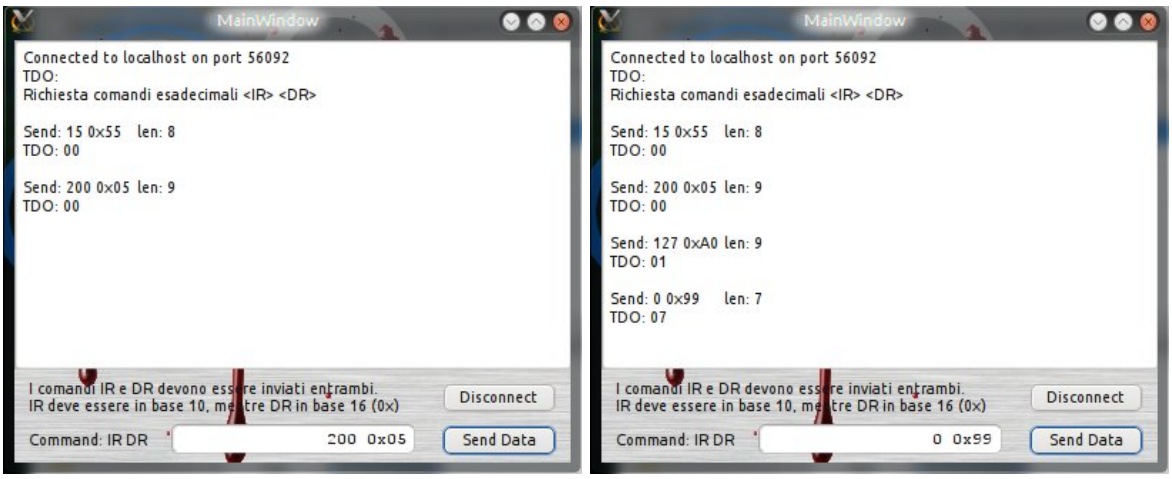



Figura 5.5: Virtual JTAG - Interfaccia Qt

# Indice



# Riferimenti bibliografici

[1] *Manuale Verilog HDL versione 2001* - Synopsys, FPGA Compiler II / FPGA Express Reference Manual

[2] *Lezioni in Verilog con esempi* – http://csg.csail.mit.edu/6.375/6_375_2006_www/handouts/lectures/

[3] *Regole di codifica e metodidi scrittura* - ALTERA, Recommended HDL Coding Styles

[4] *Altera Virtual JTAG (altera_virtual_jtag) IP Core User Guide* - http://www.altera.com/literature/ug/ug_virtualjtag.pdf

[5] *Aritmetica Signed&Unsigned* - http://www.uccs.edu/~gtumbush/published_papers/Tumbush%20DVCon%2005.pdf

[6] *Esempio sul Virtual JTAG* - http://idle-logic.com/2012/04/15/talking-to-the-de0-nano-using-the-virtual-jtag-interface/

[7] *Processore NIOS II* - http://www.altera.com/devices/processor/nios2/ni2-index.html

[8] *Cyclone IV Internal Memory & Megafunction* - http://www.altera.com/literature/hb/cyclone-iv/cyiv-51003.pdf – http://www.altera.com/literature/ug/ug_ram_rom.pdf

[9] *Cyclone IV Device Datasheet* - http://www.altera.com/literature/hb/cyclone-iv/cyiv-53001.pdf