

Carlo Biasi

CPLD E VHDL
istruzioni per l'uso

Como, 2005

INDICE

1. Rom	3
2. Pal	4
3. Pla	6
4. Pld	7
5. VHDL	8
5.1 design entity	10
5.2 architecture	10
5.3 processo	13
5.4 iterazioni	15
6. Esempi	16
6.1 flip-flop D	16
6.2 flip-flop D con CL e PR sincroni	17
6.3 flip-flop D con CL asincrono	17
6.4 contatore up/down 16 	18
6.5 contatore 10 	19
7. Funzioni e procedure	20
8. Project navigator	21
9. Data sheet	23

1. ROM – Read Only Memory

Le funzioni booleane si possono realizzare oltre che con porte logiche, multiplexer e decoder anche con dispositivi programmabili quali vari tipi di ROM, PAL, PLA, PLD, CPLD e FPGA (le ultime due utilizzando il VHDL un linguaggio molto simile al C).

In linea generale tutti questi dispositivi, che si possono schematizzare come matrici composte da un piano di porte AND e un piano di porte OR, permettono di ottenere le funzioni booleane nella 1^a forma canonica (somma di prodotti).

Consideriamo, per semplicità, una memoria programmabile di tipo ROM (EPROM, EEROM ecc.) con capacità di 8 byte (fig.1); essa avrà 3 ingressi di indirizzo che a secondo del loro valore selezioneranno 1 degli 8 registri da connettere al buffer d'uscita.

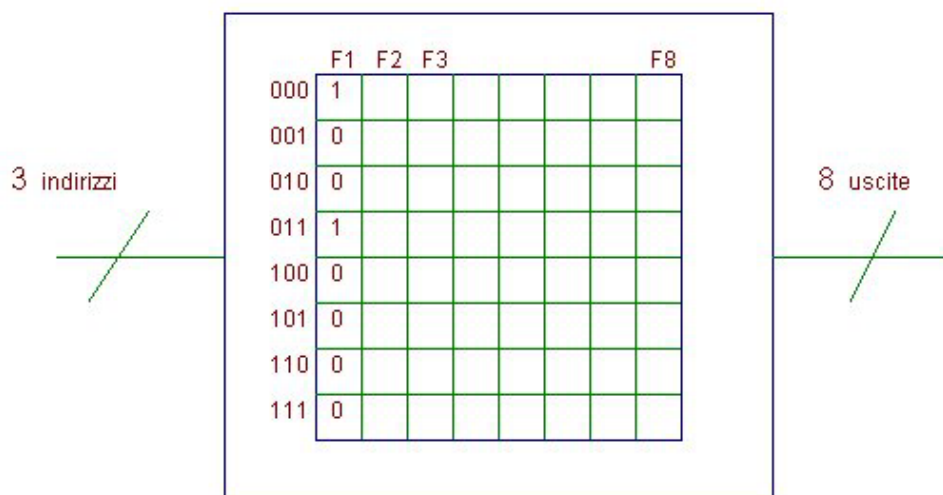


fig.1

Una memoria di questo tipo permette di sintetizzare 8 diverse funzioni booleane ciascuna di 3 variabili.

La 1^a cella del 1^o registro contiene il valore che deve assumere la 1^a funzione d'uscita in corrispondenza della combinazione '000' degli indirizzi (variabili d'ingresso).

La 2^a cella del 1^o registro contiene il valore che deve assumere la 2^a funzione d'uscita in corrispondenza della combinazione '000' degli indirizzi.

L' 8^a cella dell' 8^o registro contiene il valore che deve assumere la 8^a funzione d'uscita in corrispondenza della combinazione '111' degli indirizzi.

In sintesi, quando fornisco la combinazione A=0, B=1, C=1 sto fornendo il mintermine A'BC. Nella 1^a colonna in fig.1 è riportata la funzione $F_1 = A'BC + A'B'C'$.

Nella fig.2, che rappresenta l'architettura di una memoria ROM, si vede un piano AND fisso (non programmabile) dove sono presenti tutte le possibili combinazioni di 4

variabili (gli indirizzi) e un piano OR programmabile (le celle nelle quali si memorizzerà '1' o '0' a secondo dello stato che dovrà assumere la funzione).

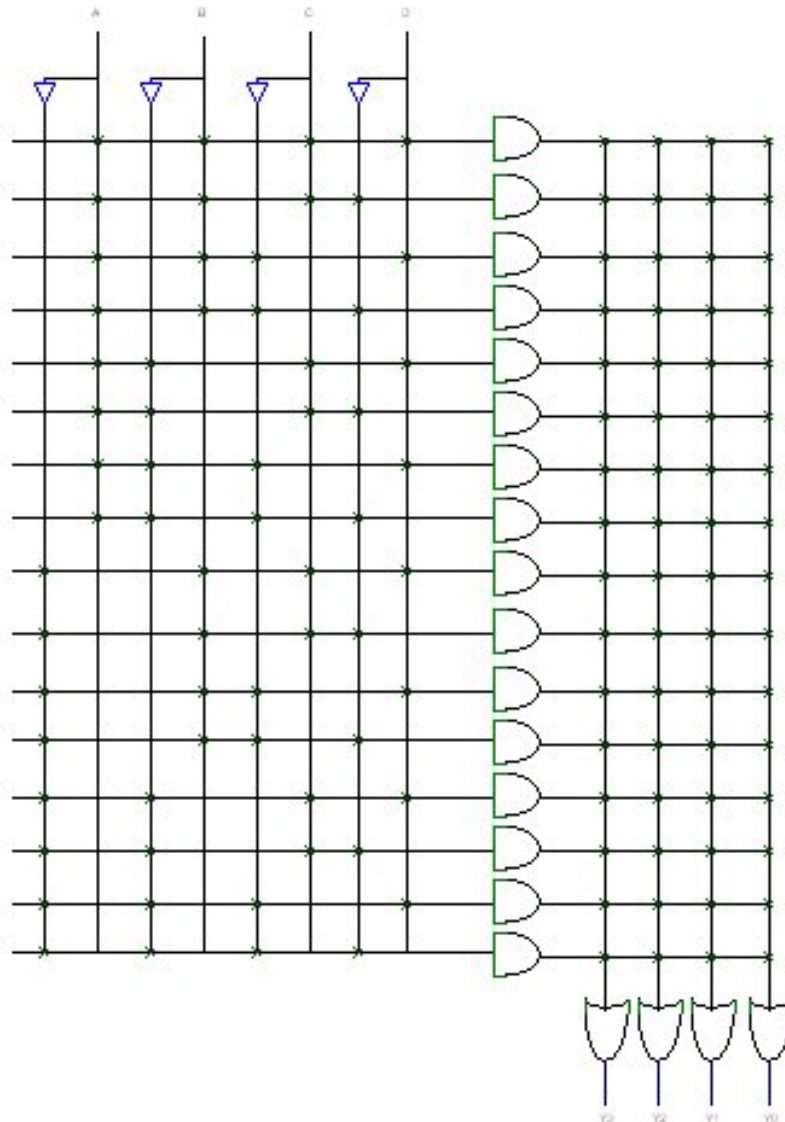


fig. 2

Utilizzare una memoria di tipo ROM per realizzare funzioni booleane è utile quando il numero delle variabili e delle funzioni è elevato.

2. PAL – Programmable Array Logic (logica a matrice programmabile)

Poiché per avere una variabile in più la capacità della ROM deve raddoppiare e molto spesso i mintermini presenti in una funzione booleana sono in numero molto inferiore di tutte le possibili combinazioni delle variabili d'ingresso sono state costruite matrici con piano AND programmabile e piano OR fisso come mostrato in fig. 3.

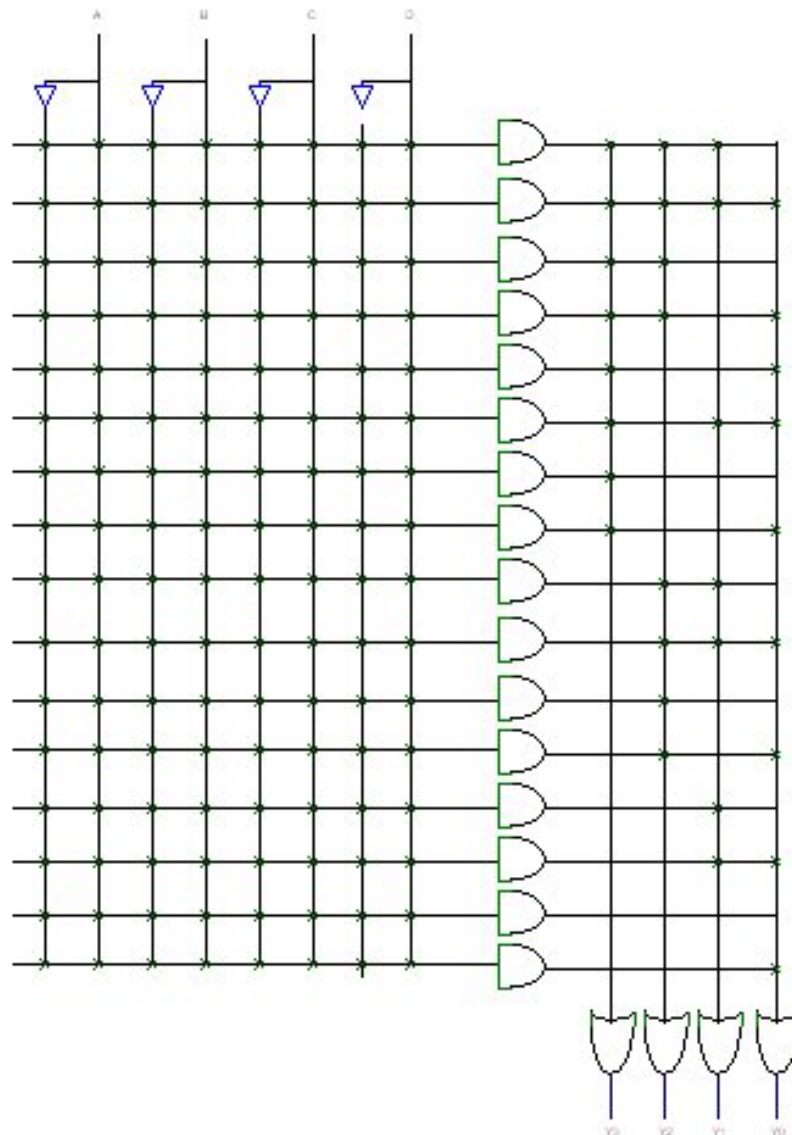


fig. 3

Se chiamiamo P_0 il mintermine generato dalla 1^a porta AND vediamo che le funzioni y_0 e y_1 sono date da:

$$y_0 = P_1 + P_4 + P_5 + P_6 + P_8 + P_{10} + P_{12} + P_{14} + P_{16}$$

$$y_1 = P_0 + P_1 + P_5 + P_8 + P_9 + P_{12} + P_{13}$$

in questo modo si riescono a realizzare, in pratica, tutte le funzioni booleane senza dover raddoppiare le dimensioni del dispositivo.

3. PLA – Programmable Logic Array (matrici logiche programmabili)

Rappresentano una evoluzione delle PAL dal punto di vista della versatilità. Come si vede dalla fig. 3 sia il piano AND che il piano OR sono programmabili. A differenza delle PAL, nelle quali i mintermini di uscita sono fissi, in questo caso è l'utente che sceglie il mintermine che deve essere presente in una determinata funzione d'uscita.

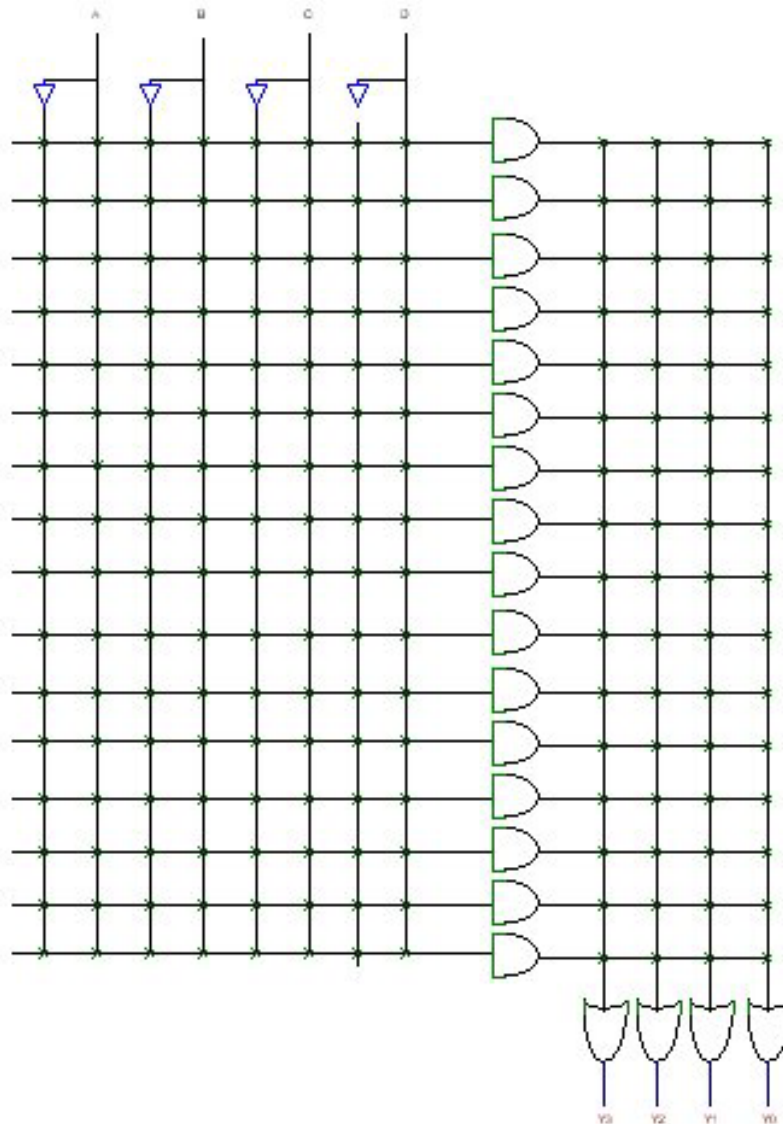


fig. 4

Le porte OR però non hanno 16 ingressi e i mintermini detti prodotti parziali non contengono tutte e 4 le variabili (o le variabili negate), ma un numero inferiore, altrimenti il dispositivo dovrebbe avere dimensioni enormi.

4. PLD – Programmable Logic Device (dispositivo logico programmabile)

Un'ulteriore evoluzione dei dispositivi precedenti consiste nel dotare la PLA di flip flop e di linee di retroazione in modo che si possano realizzare circuiti non solo combinatori ma anche sequenziali. Le CPLD (complex PLD) sono costituite da almeno qualche decina di macrocelle fig.5, ognuna costituita da una PLD (o meglio PLS – sequenziatore logico programmabile), interconnesse tra loro come mostrato in fig. 6. (Xilinx).

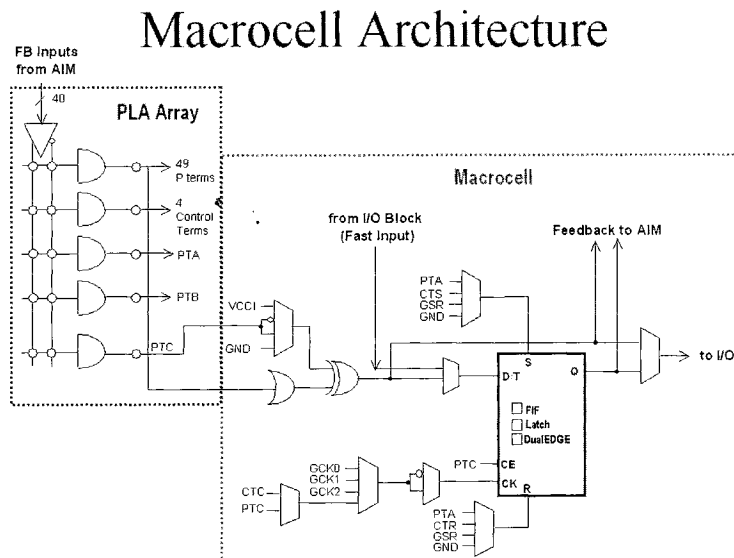


fig. 5

High Level Architecture

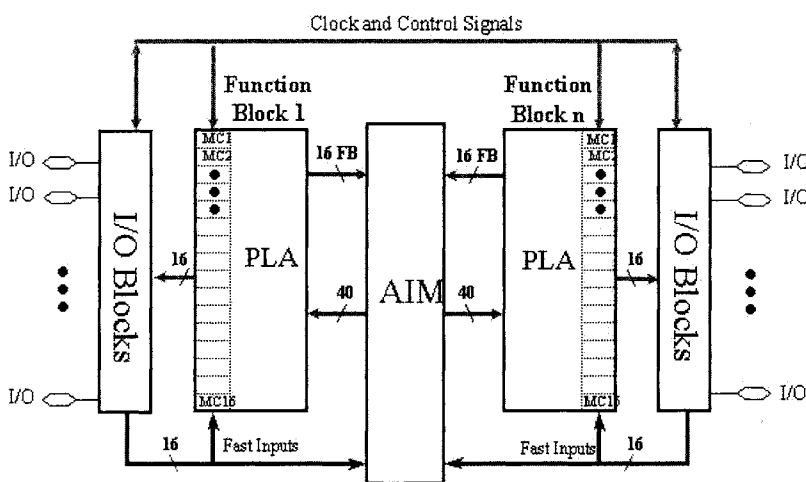


fig. 6

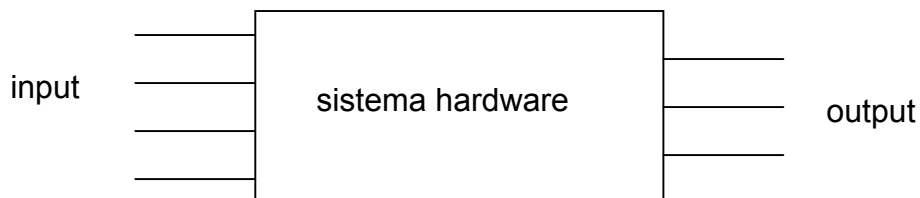
Per programmare questi dispositivi esistono diversi linguaggi i più importanti dei quali sono VHDL, VERILOG e ABEL.

In queste pagine si forniranno gli elementi base del VHDL e si utilizzerà la CPLD “Cool Runner II” della Xilinx con software “ Project Navigator” montata su demoboard della Digilent. Le specifiche dei dispositivi e dei pacchetti software utilizzati sono riportate alla fine.

5. VHDL – Very High Speed Integrated Circuit Hardware Description Language

E' un linguaggio utilizzato per la sintesi automatica e la simulazione di circuiti digitali.

Un sistema hardware si può schematizzare nella maniera più semplice dicendo che riceve degli input, esegue operazioni e fornisce degli output.

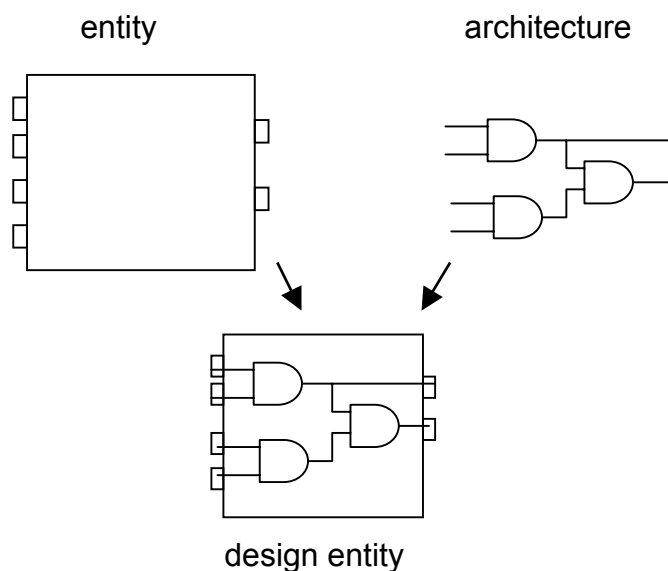


quindi per descrivere il sistema devo fare un modello che specifichi:

- la sua interfaccia esterna cioè gli ingressi e le uscite
- il suo funzionamento interno.

In VHDL questo modello è detto **design entity** ed è costituito da 2 parti:

- **entity** che descrive l'interfaccia
- **architecture** che descrive il funzionamento

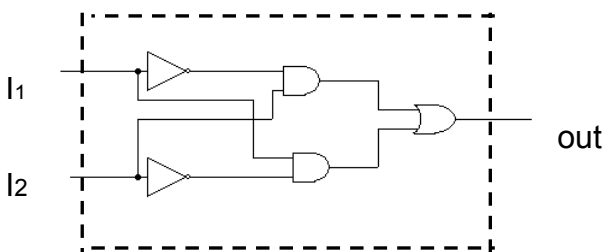


Per quanto riguarda l'architettura analizzeremo due metodi di possibile realizzazione:

- **structural** (strutturale): il sistema viene rappresentato come rete di porte logiche è detta anche *gate-level*;
- **behavioural** (comportamentale): si definisce cosa il sistema deve fare utilizzando istruzioni procedurali che realizzano algoritmi.

Esempio di porta XOR

Nella descrizione strutturale un componente è descritto connettendo tra loro più blocchi sempre però in una descrizione testuale

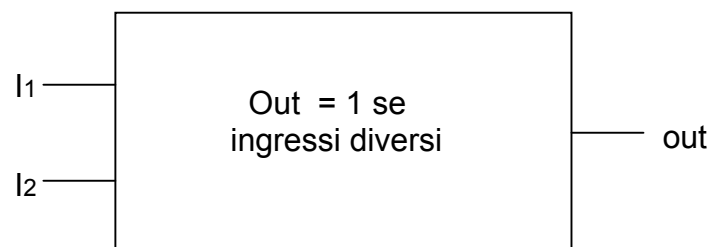


$$S_1 = (\text{NOT } I_1) \text{ AND } I_2$$

$$S_2 = I_1 \text{ AND } (\text{NOT } I_2)$$

$$\text{out} = S_1 \text{ OR } S_2$$

Nella descrizione comportamentale un componente viene descritto mediante il suo comportamento ingresso uscita. Si descrive come dovrà rispondere la rete ma non la sua struttura.



```

if I1 ≠ I2 then
  out = 1
else
  out = 0
endif

```

Quando viene utilizzata una descrizione di tipo strutturale il linguaggio mi obbliga a scrivere le istruzioni una di seguito all'altra, ma la loro esecuzione è indipendente dall'ordine con il quale vengono scritte. Tutto ciò rispecchia il comportamento delle reti combinatorie ideali e si dice che le istruzioni sono concorrenti.

Quando viene fornita una descrizione algoritmica le istruzioni vengono eseguite nell'ordine con il quale sono state scritte e si dice che le istruzioni sono sequenziali. Non bisogna confondere le istruzioni sequenziali con le reti sequenziali; una serie di istruzioni sequenziali realizzano anche reti combinatorie.

Le istruzioni sequenziali utilizzate sono:

- **IF THEN ELSE ELSIF;**
- **CASE WHEN;**
- **LOOP;**
- **WAIT.**

5.1 DESIGN ENTITY

L'istruzione **entity** dichiara una nuova design entity e ne definisce il nome; consideriamo per esempio di voler realizzare un full-adder.

```
entity full-adder is
  port (a,b,Cin : in bit ;
        S,Cout : out bit);
end full-adder
```

L'istruzione **port** definisce l'interfaccia, dice cioè quali variabili sono da considerare ingressi (**in**) e quali uscite (**out**) e di che tipo sono; nell'esempio sono di tipo **bit**.

Oltre ai modi **in** e **out** esiste anche **inout**.

Per quanto riguarda il tipo ve ne sono 9 diversi. Quelli che considereremo sono **bit** e **standard logic** anche nella variante **vector** (vettore). Per usare il tipo **standard logic** bisogna rendere visibile il package scrivendo prima della definizione di entity le seguenti istruzioni:

```
library ieee;
use ieee.standard_logic_1164.all
```

5.2 ARCHITECTURE

L'architettura descrive l'effettivo funzionamento interno di una design entity ed ha in generale la seguente forma:

```
architecture nome architettura of nome entity is
  dichiarazioni (eventuali)
begin
  istruzioni
end nome architettura
```

Riprendendo l'esempio del full-adder abbiamo per la descrizione di tipo strutturale la seguente architettura:

```
architecture full_adder1 of full_adder is
begin
    s <= a xor b xor Cin;
    Cout <= (a and b) or (b and c) or (a and c);
end full_adder1
```

e per la descrizione di tipo comportamentale:

```
architecture full_adder2 of full_adder is
signal p1,p2,p3 : bit
begin
    s <= a xor b xor Cin;
    p1 <= a and b;
    p2 <= b and c;
    p3 <= a and c;
    Cout <= p1 or p2 or p3;
end full_adder2
```

Un altro modo per definire l'architettura consiste nel fornire la tabella della verità della rete che si vuole realizzare. A tal scopo consideriamo ancora il full-adder la cui tabella della verità è:

a	b	C _{in}	S	C _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

le istruzioni che calcolano il riporto d'uscita sono le seguenti:

```
Cout <='1' when a='0' and b='1' and Cin='1' else
'1' when a='1' and b='0' and Cin='1' else
'1' when a='1' and b='1' and Cin='0' else
'1' when a='1' and b='1' and Cin='1' else
'0' when a='0' and b='0' and Cin='0' else
'0' when a='0' and b='0' and Cin='1' else
'0' when a='0' and b='1' and Cin='0' else
'0' when a='1' and b='0' and Cin='0' ;
```

molto più breve e dallo stesso significato è il seguente listato

```

Cout <= '1' when a='0' and b='1' and Cin='1' else
         '1' when a='1' and b='0' and Cin='1' else
         '1' when a='1' and b='1' and Cin='0' else
         '1' when a='1' and b='1' and Cin='1' else
         '0';

```

Le espressioni di assegnamento sono corrette se tutti i casi sono contemplati.

Se la tabella della verità prevede delle condizioni d'indifferenza, ad es. se si vuole progettare un decoder da BCD a 7 segmenti, bisogna utilizzare il tipo `std_logic`. Consideriamo ad esempio la seguente tabella della verità:

a	b	F
0	0	1
0	1	0
1	0	X
1	1	1

in questo caso il listato è il seguente:

```

F <= '0' when a='0' and b='1' else
     'X' when a='1' and b='0' else
     '1' when a='1' and b='1' else
     '1' when a='0' and b='0' else
     '0' when others;

```

Utilizzando il tipo standard logic non si hanno più 2 soli livelli e per utilizzare questo metodo detto di assegnamento condizionato bisogna considerare tutte le possibili alternative.

Tramite l'operatore di **concatenamento** si ha la possibilità di scrivere espressioni di confronto più sintetiche per l'assegnamento condizionato. Se infatti consideriamo ancora l'esempio precedente si ha:

```

architecture esempio of esempio is
    signal pippo : std_logic_vector(0 to 1);
begin
    bus <= a & b;
    with pippo select
    f <= '1' when "00",
        '0' when "01",
        '-' when "10",
        '1' when "11",
        '-' when others;
end esempio;

```

In questo caso la verifica delle condizioni non avviene più sulle variabili d'ingresso, ma sul vettore temporaneo bus.

5.3 PROCESSO

E' uno statement concorrente come ad es. un assegnamento condizionale, ma è anche uno statement composto da una parte centrale detta corpo (body) a sua volta composta da statement.

Però gli statement all'interno di un processo non sono concorrenti ma sequenziali e quindi l'ordine con cui vengono scritti influenza la logica di funzionamento.

La sintassi generale è:

```
[process_name] : process (sensitivity list)
  begin
    [body]
  end process;
```

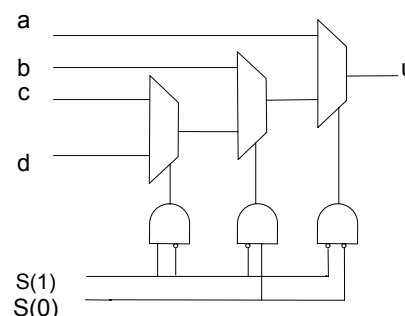
il process name è opzionale e la sensitivity list è l'elenco dei segnali in grado di attivare il processo.

Il fatto nuovo e importante della sintesi sequenziale è che l'utilizzo dell'istruzione if offre la possibilità di controllo dell'architettura.

Per comprendere meglio questo punto si può provare a realizzare un multiplexer da 4 in 1 in due modi diversi utilizzando l'istruzione if.

```
if (s="00") then
  u <= a;
elsif (s="01") then
  u <= b;
elsif (s="10") then
  u <= c;
elsif (s="11") then
  u <= d;
else u <= X;
```

end if



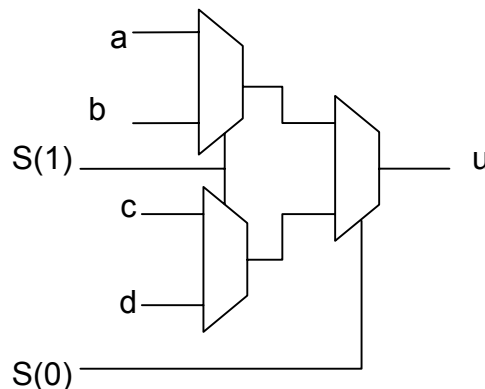
Con le istruzioni a sinistra l'architettura che ne deriva è quella a destra. Si vede che i multiplexer quando vengono selezionati dalla porta AND portano in uscita ciò che è presente sull'ingresso in alto.

Se scriviamo il programma in modo diverso otteniamo un'architettura diversa infatti:

```

if (s(0)='0') then
  if (s(1)='0') then
    u <= a;
  else
    u <= b;
  end if
else
  if (s(1)='1') then
    u <= c;
  else
    u <= d;
  end if;
end if;

```



Tramite la selezione prioritaria **if** una condizione viene valutata e viene selezionata tra due differenti alternative (ottenute con **if** annidati). Le prime condizioni avranno la priorità e ciò comporta lo sbilanciamento della rete dal punto di vista dei ritardi di propagazione delle diverse linee.

E' evidente che utilizzando la selezione parallela **case** un'espressione viene valutata e, in base al suo risultato, viene selezionata una tra più alternative allo stesso livello di priorità. E' l'equivalente del costrutto concorrente **with-select**. Considerando l'esempio del multiplexer da 4 in 1 si ha:

```

Case selezione is
  when "00" =>
    u <= a ;
  when "01" =>
    u <= b ;
  when "10" =>
    u <= c ;
  when "11" =>
    u <= d ;
  when others =>
    u <= 'X';
end case

```

In questo caso si sono utilizzate come condizioni delle costanti ma in generale si possono utilizzare range per variabili di tipo integer. Da tener presente però che il tipo integer conduce a sprechi nella sintesi della rete perché a prescindere dalla dimensione viene considerato bus a 32 bit. Ad es. per un contatore [8] sarebbero istanziate connessioni a 32 bit benchè siano sufficienti 3 bit.

5.4 ITERAZIONI

Si utilizzano per eseguire ripetutamente una serie di istruzioni sequenziali. Ve ne sono di 3 tipi:

1° - **while**: l'esecuzione è controllata dal verificarsi di una condizione specificata.

2° - **for**: l'esecuzione è controllata dalla variazione del valore di un parametro specificato.

3° - **senza schema**: l'esecuzione è controllata solo dalle stesse istruzioni interne.

Per uscire da un loop del 3° tipo vi sono 2 istruzioni:

1° - **exit**: termina ed esce dal loop

2° - **next**: termina l'esecuzione delle istruzioni e salta alla ripetizione successiva

Di seguito vengono proposti alcuni esempi di utilizzo delle precedenti istruzioni:

1) assegnamento di un segnale vettoriale b di 16 bit ad un segnale vettoriale a della stessa dimensione, svolto bit a bit

```

...
signal a,b: std_logic_vector (0 to 15)
.....
for l in 0 to 15 loop
    a(l) <= b(l);
end loop;

```

2) clock senza schema

```

loop
    CLK <= not CLK
    wait for CLK_CYCLE/2;
    exit when STOP = '1';
end loop;

```

L'istruzione **wait for** sospende il loop per un tempo pari alla metà del ciclo di clock definito esternamente dalla costante CLK_CYCLE.

L'istruzione **exit when** termina il processo ed esce dal loop se il segnale STOP è uguale ad 1 altrimenti riprende il loop.

3) clock con schema while

```

while STOP /= '0' loop
    CLK <= not CLK
    wait for CLK_CYCLE/2;
end loop;

```

Il loop continua fino a che STOP è diverso da 0.

Si vede dagli esempi precedenti che l'istruzione `wait` sospende l'esecuzione e si può realizzare in 3 modi:

1° - specificando la durata della pausa;

wait for 10 ns; - si ferma per 10 ns

2° - specificando una condizione che deve essere verificata per poter continuare;

wait until CLK='1' - attende che il CLK torni a 1

3° - specificando una lista di segnali su cui si deve verificare un evento per poter continuare

wait on X,Y,Z - esecuzione sospesa finchè non cambia uno dei segnali

E' importante sottolineare il fatto che un processo se contiene una `wait` non deve avere la sensitivity list.

6. ESEMPI

Gli esempi seguenti mostrano come si possono implementare dispositivi hardware come flip-flop e contatori di vario genere.

6.1 FLIP FLOP D

```
entity flipflopD is
    port (D,clock : in bit;
          Q       : out bit);
end flipflopD;

architecture arch_flipflopD of flipflopD is
begin
    process
    begin
        wait until clock='1' and clock 'event
            Q<=>D;
        end process;
    end arch_flipflopD;
```

L'istruzione **`wait until clock='1' and clock 'event`** blocca l'esecuzione del processo fino a che il segnale di clock è al livello 1 e si verifica un evento (fronte) sul segnale di clock.

6.2 FLIP FLOP D CON CL E PR SINCRONI

```

Entity FF_D_SYNC_CL_PR is
  port ( CLK:      in std_logic;
         CL:       in std_logic;
         PR:       in std_logic;
         D:        in std_logic;
         Q:        out std_logic);
end FF_D_SYNC_CL_PR ;

architecture dataflow of FF_D_SYNC_CL_PR is
begin
  ff : process ( CLK )
  begin
    if ( CLK 'event and CLK = '0' ) then
      if ( CL = '1' ) then
        Q <= '0' ;
      elsif ( PR = '1' ) then
        Q <= '1' ;
      else
        Q <= D ;
      end if;
    end if;
  end process;
end dataflow;

```

Si vede che per rendere sincroni i comandi di clear e preset basta testarli dopo il segnale di clock. Da notare anche l'utilizzo di **if** al posto di **wait** per la realizzazione del clock. Il flip flop commuta sul fronte di discesa.

6.3 FLIP FLOP D CON CL ASINCRONO

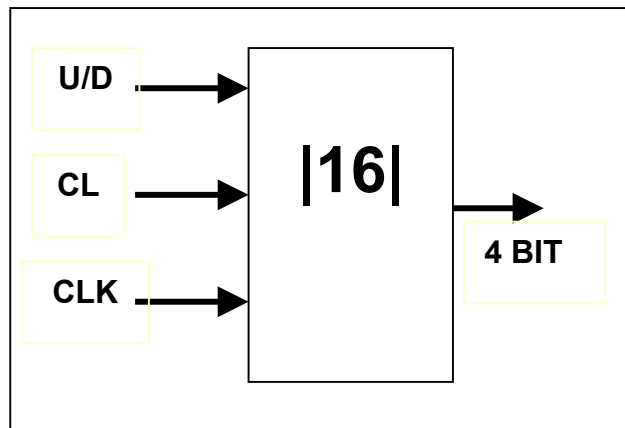
```

architecture dataflow of FF_D_SYNC_CL is
begin
  ff : process ( CLK )
  begin
    if ( CL = '1' ) then
      Q <= '0' ;
    else
      if ( CLK 'event and CLK = '0' ) then
        Q <= D ;
      end if;
    end if;
  end process;
end dataflow;

```

In questo caso il test del clear (attivo alto) avviene prima del segnale di clock realizzando così un comando asincrono.

6.4 CONTATORE UP/DOWN |16|



Il contatore ha le seguenti caratteristiche: commuta sul fronte di salita del clock, quando U/D=1 conta all'indietro e il CL è asincrono e attivo basso.

```

library vector;
use vector. functions.all;
entity cont is
    port ( clk :in bit;
           ud  :in bit;
           cl  :in bit;
           q   :out bit_vector (3 downto 0));
end;

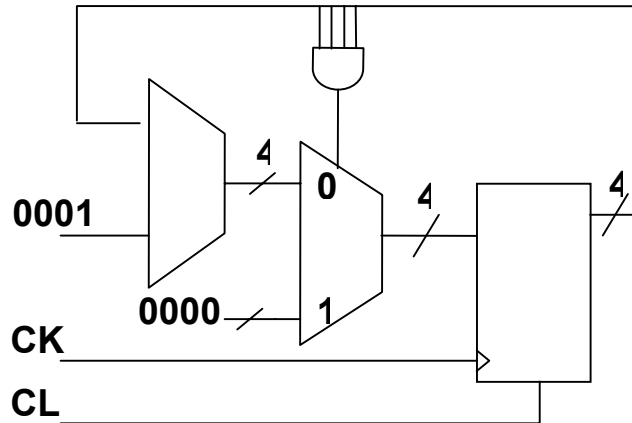
architecture comportamentale of cont is
    signal qqqq: bit_vector (3 downto 0);
begin
    contatore: process ( clk,cl )
    begin
        if ( cl = '0' ) then qqqq <= "0000";
        elsif ( clk 'event and clk = '1' ) then
            if ud = '1' then qqqq <= qqqq + "0001";
            else qqqq <= qqqq - "0001";
            end if;
        end if;
    end process contatore;
    q <= qqqq;
end comportamentale;

```

Si utilizza la nuova libreria "Library vector" perché ci sono operazioni aritmetiche. Per ulteriori informazioni riguardo le librerie presenti nel VHDL e il loro uso si consiglia di rivolgersi a testi di approfondimento.

6.5 CONTATORE |10|

Si realizza con la seguente architettura:



Il primo dispositivo è un sommatore, il secondo un multiplexer e il terzo un registro. Quando arriva un fronte di salita del segnale di clock, se è selezionato l'ingresso '0' del multiplexer, viene fornita al registro l'uscita del sommatore cioè lo stato precedente del registro più '1'; se è selezionato l'ingresso '1' del multiplexer al registro arriva '0'. L'ingresso di selezione del multiplexer è dato dalla porta AND ai cui ingressi vi sono DC'B'A (l'apice indica la variabile negata).

```
entity cont10 is
  port ( ck: in std_logic;
         cl: in std_logic;
         y: out std_logic_vector ( 0 to 3 )
       );
end cont10;
```

```
architecture rtl of cont10 is
  signal q: std_logic_vector ( 0 to 3 );
begin
  contatore: process ( ck,cl )
  begin
    if ( cl = '1' ) then
      q <= "0000";
    elsif ( ck 'event and ck = '1' ) then
      if ( q <= "1001" ) then
        q <= "0000";
      else
        q <= q + "0001";
      end if;
    end if;
  end process;
  y <= q;
end rtl;
```

Da notare che nell'istruzione indicata dalla freccia basta sostituire il valore di 'q' per cambiare il modulo del contatore.

7. FUNZIONI E PROCEDURE

Pur rimandando, per una trattazione più rigorosa, a testi specifici di informatica è importante dire che funzioni e procedure sono sottoprogrammi che permettono una gestione modulare del progetto. Sono cioè gruppi di istruzioni sequenziali, isolate dal resto del sorgente, che vengono utilizzati quando/dove servono attraverso le chiamate.

Più precisamente le funzioni hanno una lista di argomenti di solo ingresso e devono restituire un valore alla fine, dopo l'istruzione return. Le chiamate di funzioni sono espressioni all'interno di altre istruzioni. Le procedure hanno una lista di argomenti sia d'ingresso che di uscita e non restituiscono alcun valore. Le chiamate di procedure sono vere e proprie istruzioni.

La seguente funzione genera un bit di parità uguale a '0' se la somma degli 8 bit è pari.

```
function parità ( signal D: bit_vector ( 0 to 7 ),
                ) return bit_vector ( 0 to 8 ) is
    variabile I : integer
    variabile P : bit := '0' ;
begin
    for I in 0 to 7 loop
        if D(I)='1' then
            P:= not P ;
        end if;
    end loop;
    return D&P;
end;
```

L'istruzione ' **variabile P : bit := '0'** ' inizializza P a 0 mentre l'istruzione **P:= not P ;** partendo da D(0) ogni volta che un bit è 1 nega P infine l'istruzione **return D&P;** restituisce un valore che è il dato di 8 bit più il bit di parità.

La seguente architettura mostra un es. di chiamata della funzione parità.

```
architecture chiamata_funzione of esempio is
    signal X: bit_vector ( 0 to 7 ) ;
    signal Y: bit_vector ( 0 to 8 ) ;
begin
    -----
    -----
    Y <= parità (D => X);
    -----
end;
```

8. PROJECT NAVIGATOR

Di seguito vengono riportati i passi da seguire per creare un progetto con Project Navigator, il software della Xilinx utilizzato per la CPLD Cool Runner II, e trasferirlo sulla demoboard della Digilent.

Programmare circuiti con clock esterno

1. Aprire project navigator
2. Menù file → new project
3. Chiamare il progetto con un nome quindi salvarlo e premere avanti
4. Appare il box di dialogo del new project. Scegliere il dispositivo e il package e premere avanti
5. Cliccare su New source
6. Selezionare VHDL Module e chiamarlo con un nome
7. Cliccare su avanti
8. Definire l'entity assegnando un nome
9. Definire il nome delle porte (variabili) e direzione (input/output)
10. Cliccare su avanti
11. Cliccare su fine
12. Cliccare su avanti
13. Cliccare su avanti
14. Cliccare su fine
15. Nell'editor scrivere il programma
16. Nel menù file selezionare save per salvare
17. Nella finestra Source in project selezionare il file.vhd col tasto destro e selezionare New source
18. Nella finestra New Source selezionare Implementation constrains file e dare un nome al file
19. Cliccare su avanti
20. Cliccare su avanti
21. Cliccare su fine
22. Notare che nella finestra Source in project appare il file con estensione .ucf
23. Doppio click sul file .ucf
24. Appare la finestra per l'assegnazione dei piedini
25. Nella finestra I/O pins impostare nel campo Loc i piedini (osservare la legenda per evitare di usare piedini inutilizzabili – vedi tabelle pins)
26. Salvare e chiudere la finestra dell'assegnazione dei piedini (ritornando così alla finestra con l'editor)
27. Collegare la CPLD al PC tramite cavo parallelo e alimentarlo con voltaggio compreso tra 5 e 9 volt
28. Nella finestra Source in project selezionare il file .vhd
29. Nella finestra Processes for Source aprire Implement design → Generate programming file
30. Doppio click su "Configure device (iMPACT)"
31. Selezionare Boundary scan mode
32. Cliccare su avanti
33. Selezionare Automatically connect
34. Cliccare su fine
35. Cliccare su OK

36. Cliccare con il tasto destro su xc2c256 e poi su Assign new configuration file
37. Nella finestra che si apre selezionare il file .jed e poi cliccare su apri
38. Tasto destro sullo stesso dispositivo e selezionare Program...
39. Cliccare su OK per terminare la programmazione del CPLD

N.B. assicurarsi che i jumper JP1, JP2, JP3 e JP4 siano posizionati su Vreg per alimentare correttamente la CPLD tramite bancone

Digilent XC2-XL System Board Reference Manual

Revision: May 11, 2004



www.digilentinc.com

246 East Main | Pullman, WA 99163
(509) 334 6306 Voice and Fax

Overview

The Digilent XC2-XL is self-contained circuit development platform that contains a Xilinx CoolRunner-II XC2C256 CPLD and a Xilinx XC9572XL CPLD. The XC2-XL is an ideal platform for CPLD-based circuit design using the latest Xilinx CAD tools. It provides a JTAG programming circuit, power supplies, a clock source, and basic I/O devices, so that circuits can be implemented immediately without the need for any other components. All CPLD signals are brought to expansion connectors, allowing accessory circuits to be constructed in the on-board prototype area, or attached via an expansion board. The XC2-XL is compatible with all versions of Xilinx CAD tools, including the free WebPack tools available at the Xilinx website. XC2-XL features include:

- A Xilinx CoolRunner-II XC2C256 CPLD in a TQ144 package;
- A Xilinx XC9572XL CPLD in a VQ44 package;
- JTAG ports to both CPLDs that can be independently enabled or disabled;
- Flexible power delivery using a wall-plug transformer, batteries, or external supplies;
- A socketed oscillator (1.8432MHz included; clocks up to 100+MHz can be used);
- Full routing of all I/O signals from both CPLDs to expansion connectors;
- A button and two LEDs for basic I/O;
- Non-volatility – as with all Xilinx CPLDs, designs remain after power is removed.

Several expansion boards containing a variety of I/O devices are available for the XC2-XL. These fully assembled and tested boards can be used to quickly and easily enhance the features of the XC2-XL. See the Digilent website at www.digilentinc.com/xc2xl for more information.

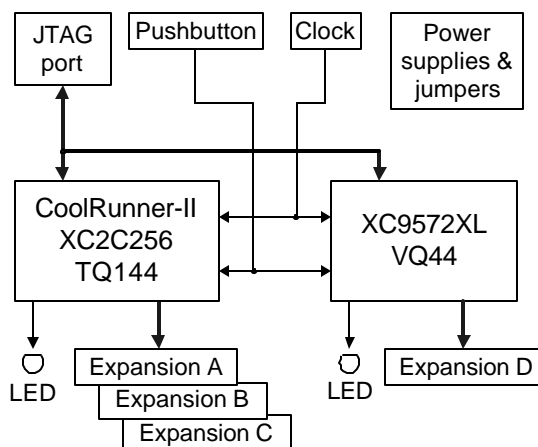


Figure 1. XC2-XL circuit board block diagram

Functional Description

The XC2-XL provides a minimal system that can be used to rapidly implement CPLD-based circuits, and to gain exposure to Xilinx CAD tools and CPLD-oriented design methods. The XC2-XL provides only the essential circuits needed to support the CPLDs, including power supplies, a clock source, and basic I/O (a pushbutton and two LEDs). All available I/O signals are routed to expansion connectors that mate with 40-pin, 100 mil spaced DIP headers available from several distributors.

The on-board power supplies and clock source can easily be disconnected from the CPLDs so that external power and clock signals can be used. Power supply design and decoupling follow recommended design practices, so the XC2-XL has stable, low noise supplies regardless of the power source used.

CPLD programming is accomplished via a 6-pin, 3.3V JTAG programming header that is

compatible with a variety of cables, including the JTAG3 cable from Digilent, the Parallel-3 or -4 cables from Xilinx, as well as cables from other vendors.

The XC2-XL measures 5.25" x 5.25", and it contains an 18-hole x 46-hole wire wrap area. The wire-wrap area can accommodate a self-adhesive solderless breadboard, allowing flexibility in accessory circuit construction.

Both CPLDs on the XC2-XL board are loaded with a sample configuration during board test. This basic configuration flashes the on-board LEDs at different rates that are selectable using the on-board button. This configuration, which can be downloaded from the Digilent or Xilinx websites, serves as a quick board check as well as a basic reference design.

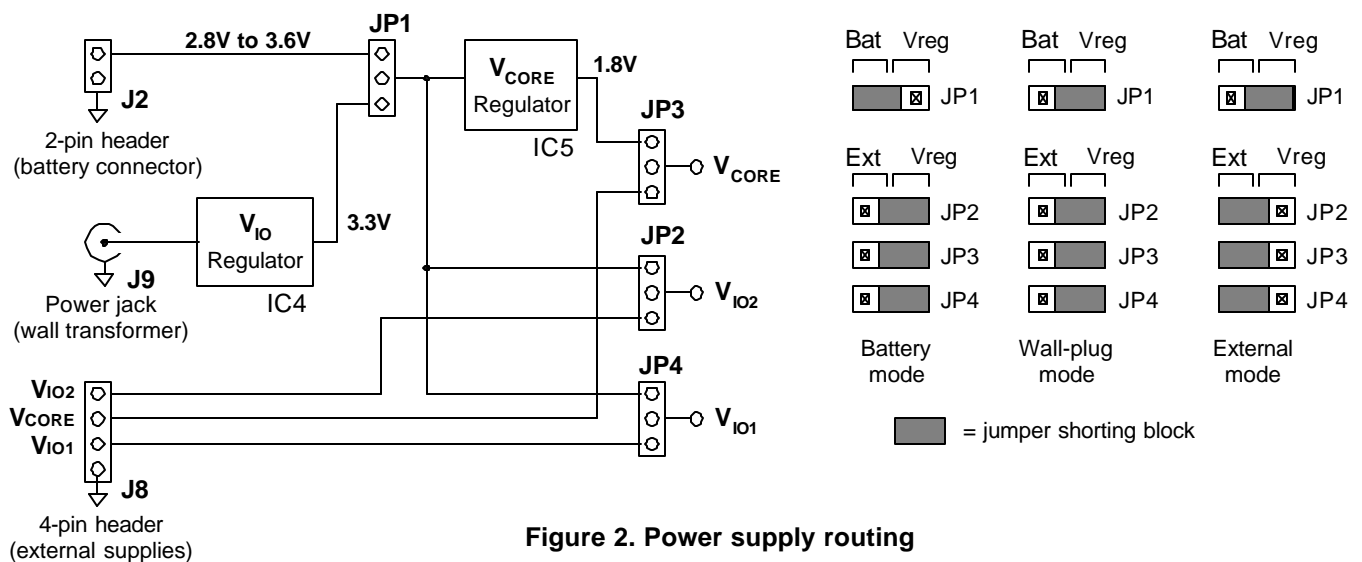
Power Supplies

The CoolRunner-II requires three power supplies (V_{CORE} , V_{IO1} , and V_{IO2}), so the XC2-XL has three separate circuits for power delivery. The V_{CORE} and V_{IO1} supplies are routed only to the CoolRunner 2, and they are set to 1.8V and 3.3V respectively. V_{IO2} is routed to both CPLDs, and it is also set to 3.3V. V_{CORE} is generated from a 1.8V LM1117 LDO regulator that can supply up to 800mA of current. Jumper block JP3 allows V_{CORE} to be disconnected from the on-board regulator so that it can be brought in from an external supply. V_{IO1} and V_{IO2} both arise from the same

3.3V LM317 regulator that can supply up to 1.5A of current. Jumper-blocks JP2 and JP4 can disconnect V_{IO1} and V_{IO2} from the on-board regulators so that external I/O supplies can be used.

Jumper block JP1 selects whether V_{IO} is supplied from the 3.3V regulator, or from a source connected to J2. Regardless of the JP1 setting, the V_{CORE} regulator is used to generate the 1.8V supply. Jumper blocks JP2, JP3, and JP4 select whether V_{IO1} , V_{CORE} , and V_{IO2} are supplied from on-board or external supplies. The figure below shows the power supply routing and typical jumper settings.

Power can be supplied to the XC2-XL using any one of three modes. *Wall-plug mode* supplies power from any 5VCD-9VDC wall-plug-transformer supply connected to the power jack (J9) on the XC2-XL. The supply must source at least 250mA of current, and it must use a center-positive, 2.1mm ID/5.5mm OD connector. *Battery mode* supplies power from a battery pack (or other DC source) connected to the J2 header on the XC2-XL. The batteries must output between 2.8V and 3.6V. *External mode* uses the J8 header to bring regulated supplies from any external source. In the wall-plug mode, the V_{CORE} and V_{IO} regulators are used, so there is little chance of damaging the CPLDs by using incorrect supply voltages (both regulators can handle up to 18VDC). In battery mode, the V_{CORE} regulator is used, but the supplied



voltage directly drives the V_{IO} connections, so care must be taken to ensure no more than 3.8V is applied. In external mode, no regulators are used, so care must be taken to ensure CPLD voltage requirements are met.

CPLD Configuration

The CPLDs on the XC2-XL are connected in a JTAG scan chain as shown in the figure below. Either device can be removed from the chain by setting jumper blocks P5, JP6, JP9, and JP10 appropriately. The scan chain originates from a 6-pin header connector that is compatible with the JTAG3 cable from Digilent, and the P-3 and P-4 cables from Xilinx.

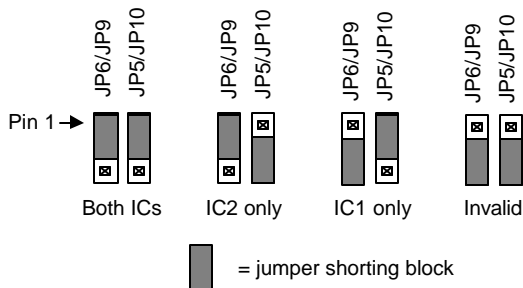
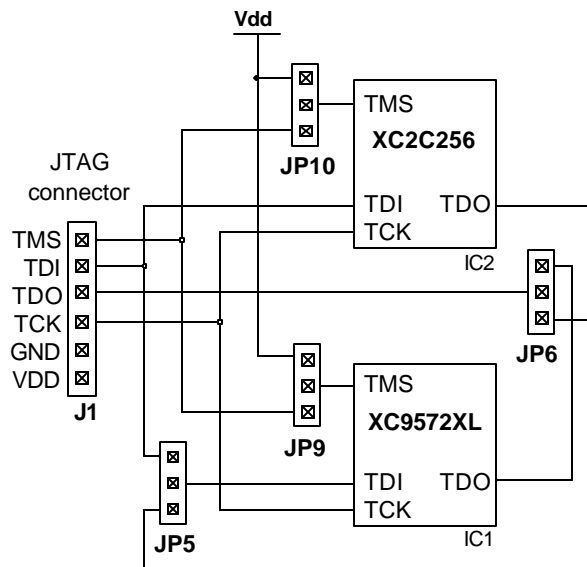


Figure 3. JTAG signal routing

Oscillator

The XC2-XL uses a half-size 8-pin DIP oscillator in an 8-pin socket. The board ships with a 1.842MHz oscillator, but oscillators from 32KHz to 100MHz can be used. The oscillator is connected to the GCK2 input on both CPLDs.

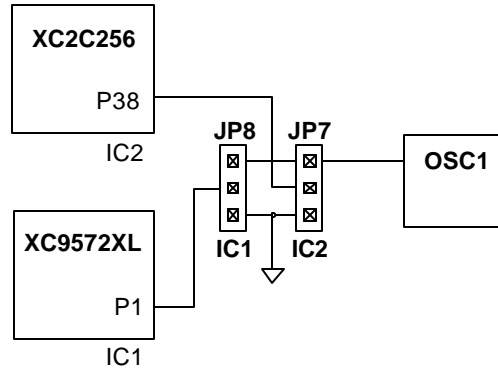


Figure 4. Clock signals

Note: The IC1 and IC2 legends are incorrect on the circuit board silkscreen; they are correct in the figure above.

Pushbutton and LEDs

A pushbutton and two LEDs provide basic I/O functions on the XC2-XL. The LED can be illuminated to verify that configuration was successful, or flashed at a given rate to indicate a particular status. The pushbutton can be used to provide a basic reset function, or to select an operating mode. The pushbutton drives the GSR input on both CPLDs, and the LED is driven from a general I/O pin.

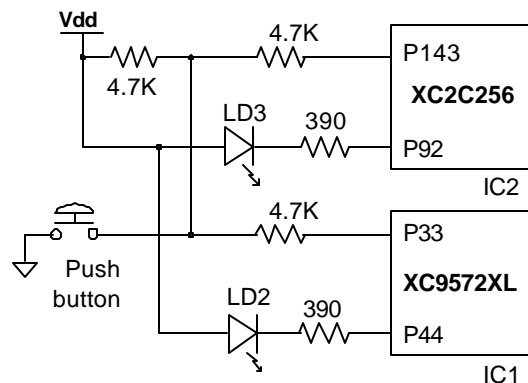


Figure 5. Pushbutton and LEDs Expansion Connectors

Four expansion connectors have been provided to allow XC2 designs to be expanded on the included prototype area, or by attaching peripheral boards. The connectors use 2 x 20, 100-mil spaced right-angle sockets so that standard headers can be used on peripheral boards. Several Digilent expansion boards can be used with the XC2-XL, including solderless breadboards, wire-wrap boards, and analog and digital I/O boards. See Digilent’s website at www.digilentinc.com/xc2xl for more information on available expansion boards.

All available signals are routed from the two CPLDs to the connectors as shown in the figure below. VCC (3.3V regulated), VU (depends on power supply used), and GND are also routed to the connector so that attached devices can draw power from the XC2-XL board. If the 3.3V regulated supply is used, no more than 1.5A should be drawn. Table 1 below shows XC2-XL expansion connector pinouts.

The XC2-XL contains a Xilinx XC2C256 CoolRunner II CPLD in a TQ144 package, and an XC9572 CPLD in a VQ44 package. Both CPLDs have a clock source, pushbutton input, LED output, and connections to the JTAG programming signals. Other than these connections, all I/O signals are routed to the expansion connectors. CPLD pinouts are provided in tables 2 and 3 below.

The CPLDs can be programmed using the Xilinx ISE/WebPack software and the JTAG3 cable from Digilent (the P-3 or P-4 programming cable from Xilinx can also be used).

Please see the data sheets for the CPLDs available at the Xilinx web site for more information.

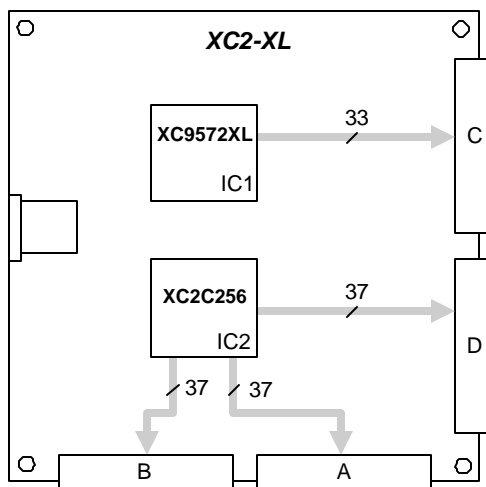
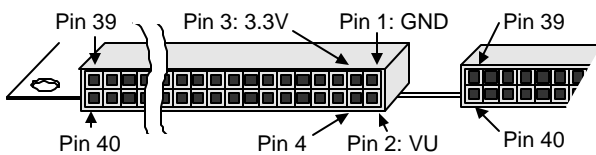


Figure 5. Expansion Connectors

CPLDs

Table 1. XC2-XL board expansion connector pinouts

A connector (J3)			B connector (J4)			C connector (J5)			D connector (J6)		
Pin	Signal	CR2 pin	Pin	Signal	CR2 pin	Pin	Signal	XC95 pin	Pin	Signal	CR2 pin
1	GND	-	1	GND	-	1	GND	-	1	GND	-
2	VU	-	2	VU	-	2	VU	-	2	VU	-
3	VDD33	-	3	VDD33	-	3	VDD33	-	3	VDD33	-
4	A4	43	4	B4/GSR	143	4	C4/LD3	44	4	D4	91
5	A5	42	5	B5	142	5	C5	43	5	L5	88
6	A6	41	6	B6	140	6	C6	42	6	D6	87
7	A7	39	7	B7	139	7	C7	41	7	D7	86
8	A8	40	8	B8	138	8	C8	40	8	D8	85
9	XCCLK	38	9	B9	137	9	C9	39	9	D9	83
10	A10	34	10	B10	136	10	C10	38	10	D10	82
11	A11	35	11	B11	135	11	C11	37	11	D11	81
12	A12	33	12	B12	134	12	C12	36	12	D12	80
13	A13	32	13	B13	133	13	C13	34	13	D13	79
14	A14	31	14	B14	132	14	C14/GSR	33	14	D14	78
15	A15	30	15	B15	131	15	C15	32	15	D15	77
16	A16	28	16	B16	130	16	C16	31	16	D16	76
17	A17	26	17	B17	129	17	C17	30	17	D17	75
18	A18	25	18	B18	128	18	C18	29	18	D18	74
19	A19	24	19	B19	126	19	C19	28	19	D19	71
20	A20	23	20	B20	125	20	C20	27	20	D20	70
21	A21	22	21	B21	124	21	C21	23	21	D21	69
22	A22	21	22	B22	121	22	C22	22	22	D22	68
23	A23	20	23	B23	120	23	C23	21	23	D23	66
24	A24	19	24	B24	119	24	C24	20	24	D24	64
25	A25	18	25	B25	118	25	C25	19	25	D25	61
26	A26	17	26	B26	117	26	C26	7	26	D26	60
27	A27	16	27	B27	116	27	C27		27	D27	59
28	A28	15	28	B28	115	28	C28		28	D28	58
29	A29	14	29	B29	114	29	C29		29	D29	57
30	A30	13	30	B30	113	30	C30	3	30	D30	56
31	A31	12	31	B31	112	31	C31	2	31	D31	54
32	A32	11	32	B32	111	32	XLCLK	1	32	D32	53
33	A33	10	33	B33	110	33	C33	18	33	D33	52
34	A34	9	34	B34	107	34	C34	16	34	D34	51
35	A35	7	35	B35	106	35	C35	14	35	D35	50
36	A36	6	36	B36	105	36	C36	13	36	D36	49
37	A37	5	37	B37	104	37	C37	12	37	D37	48
38	A38	4	38	B38	103	38	C38	8	38	D38	46
39	A39	3	39	B39	102	39	C39	6	39	D39	45
40	A40	2	40	B40	101	40	C40	5	40	D40	44

Pin	Function	Pin	Function	Pin	Function	Pin	Function
1	VCORE	37	VCORE	73	VIO1	109	VIO2
2	A40	38	XCCLK	74	D18	110	B33
3	A39	39	A7	75	D17	111	B32
4	A38	40	A8	76	D16	112	B31
5	A37	41	A6	77	D15	113	B30
6	A36	42	A5	78	D14	114	B29
7	A35	43	A4	79	D13	115	B28
8	VAUX	44	D40	80	D12	116	B27
9	A34	45	D39	81	V18	117	B26
10	A33	46	D38	82	V16	118	B25
11	A32	47	GND	83	V14	119	B24
12	A31	48	D37	84	VCORE	120	B23
13	A30	49	D36	85	V13	121	B22
14	A29	50	D35	86	V12	122	TDO
15	A28	51	D34	87	V8	123	GND
16	A27	52	D33	88	V6	124	B21
17	A26	53	D32	89	GND	125	B20
18	A25	54	D31	90	GND	126	B19
19	A24	55	VIO1	91	V5	127	VIO2
20	A23	56	D30	92	LD2	128	B18
21	A22	57	D29	93	VIO1	129	B17
22	A21	58	D28	94	SCK	130	B16
23	A20	59	D27	95	SDO	131	B15
24	A19	60	D26	96	SDI	132	B14
25	A18	61	D25	97	C32	133	B13
26	A17	62	GND	98	V2	134	B12
27	VIO1	63	TDI	99	GND	135	B11
28	A16	64	D24	100	V3	136	B10
29	GND	65	TMS	101	B40	137	B9
30	A15	66	D23	102	B39	138	B8
31	A14	67	TCK	103	B38	139	B7
32	A13	68	D22	104	B37	140	B6
33	A12	69	D21	105	B36	141	VIO2
34	A10	70	D20	106	B35	142	B5
35	A11	71	D19	107	B34	143	BTN
36	GND	72	GND	108	GND	144	GND

Pin	Function	Pin	Function	Pin	Function	Pin	Function
1	XLCLK	12	C37	23	C21	34	C13
2	C31	13	C36	24	TDO	35	VCORE
3	C30	14	C35	25	GND	36	C12
4	GND	15	VCORE	26	VIO	37	C11
5	C40	16	C34	27	C20	38	C10
6	C39	17	GND	28	C19	39	C9
7	C26	18	C33	29	C18	40	C8
8	C38	19	C25	30	C17	41	C7
9	TDI	20	C24	31	C16	42	C6
10	TMS	21	C23	32	C15	43	C5
11	TCK	22	C22	33	BTN	44	LD3