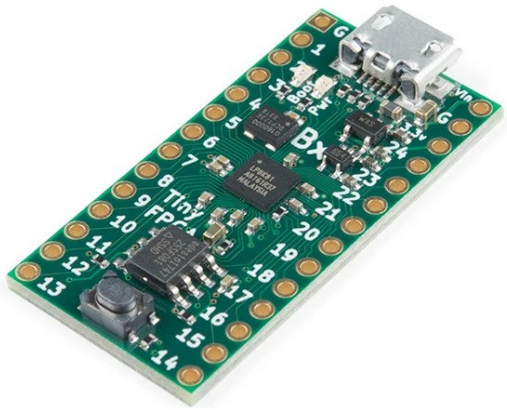
Introduzione a TinyFPGA BX
e alla sintesi RTL in Verilog

v1.05



Il termine FPGA[1] (Field Programmable Gate Array) raggruppa una classe di dispositivi che vengono utilizzati per realizzare complessi circuiti digitali. Pur essendo programmabili, le FPGA sono molto diverse dai microcontrollori: il codice sorgente non definisce un programma da eseguire bensì un circuito logico da implementare. Ogni elemento logico del progetto "consuma" una parte fisica della FPGA, che viene opportunamente configurata per realizzare funzionalmente quel particolare elemento. Inoltre, mentre in un microcontrollore le istruzioni vengono eseguite una alla volta, in una FPGA tutti gli elementi definiti sono attivi contemporaneamente, per cui le FPGA sono particolarmente utili per realizzare operazioni veloci e altamente parallele. Con le FPGA si può realizzare qualsiasi sistema digitale, compresi microcontrollori dotati di periferiche personalizzate.

La maggior parte delle FPGA vengono prodotte da Altera e Xilinx (che insieme controllano circa l'80% del mercato). Tra le compagnie minori figura Lattice Semiconductor che produce (tra le altre cose) le famiglie iCE40 ed ECP5. Queste sono FPGA molto apprezzate da sviluppatori e hobbisti poiché possono essere programmate mediante applicazioni completamente libere. La FPGA iCE40 in particolare è alla base della scheda TinyFPGA BX, che ha basso costo, ed è quindi perfetta per muovere i primi passi nel mondo delle logiche programmabili: basta inserirla su una breadboard e mettersi al lavoro interfacciandola con LED, interruttori, encoder e sensori digitali.

TinyFPGA BX fornisce agli sviluppatori la potenza e la flessibilità di una FPGA dotata di 7680 celle, in un piccolo fattore di forma, perfetto per breadboard e PCB realizzati a livello hobbistico. Attualmente non esistono altre piccole schede FPGA a basso costo con tante risorse logiche. La quantità di celle è sufficiente ad implementare una piccola CPU RISC-V a 32 bit! (PicoRV32)[2].

La programmazione può essere effettuata tramite un linguaggio di progettazione digitale (Verilog/VHDL[3], Migen[4], Chisel[5]) o mediante uno strumento di immissione schematica come

1 https://it.wikipedia.org/wiki/Field_Programmable_Gate_Array
2 https://github.com/mattvenn/TinyFPGA-BX/tree/master/examples/picosoc
3 http://www.fpga4fun.com/HDLtutorials.html
4 https://m-labs.hk/migen/manual/introduction.html
5 https://chisel.eecs.berkeley.edu/

Icestudio[6], che consente di disegnare graficamente il circuito digitale da implementare. Una volta ottenuto il design desiderato, questo viene trasferito sul modulo TinyFPGA BX tramite USB.

Le specifiche della scheda TinyFPGA BX sono le seguenti:

- Dimensioni contenute (altezza 1.4 pollici, larghezza 0.7 pollici)
- PCB di elevata qualità (4 layer con piani di alimentazione dedicati)
- Programmazione tramite interfaccia USB 2.0 full-speed (12 Mbit/s)
- Lattice FPGA iCE40[7] LP8K-CM81 (low power 8K gate - package “81 ucBGA”):
  - 7.680 celle logiche (con tabelle di associazione a quattro input)
  - 128 Kbit RAM (16KB)
  - 1 circuito PLL (notare che la iCE40 LP8K prevede due PLL ma solo uno è disponibile nel minuscolo package “81 ucBGA” di 4x4mm utilizzato sulla scheda TinyFPGA BX)
  - 41 user I/O (la iCE40 LP8K prevede 178 user I/O, ma il package “81 ucBGA” ne rende disponibili solo 63, di cui solo 41 sono portati sui pin della scheda TinyFPGA BX)
- 8 Mbit SPI Flash (per la memorizzazione di bootloader, circuito logico e dati utente)
- Regolatori LDO on-board da 3.3 V (300 mA) e 1.2 V (150 mA)
- Oscillatore MEMS 16 MHz a basso consumo:
  - Solo 1.3 mA quando attivo
  - Stabilità 50 ppm
- Bootloader USB open-source[8]

NOTA: il clock a 16MHz dell’oscillatore MEMS non può essere modificato, ma è possibile generare un modulo logico Verilog per scalare verso l’alto tale valore grazie al PLL (entro certi limiti)[9], ed eventualmente anche verso il basso usando dei divisori.

All’accensione la TinyFPGA BX carica il bootloader USB dalla flash SPI apparendo sul computer host come un dispositivo con porta seriale virtuale. Il software di programmazione rileva automaticamente tale dispositivo e utilizza l'interfaccia seriale per programmare il circuito logico sulla scheda. Una volta che il progetto è stato caricato nella flash SPI, la scheda si riavvia e carica direttamente il circuito logico implementato. Per aggiornare il design basta premere il pulsante di ripristino e il bootloader sarà nuovamente attivato. Il bootloader include inoltre dei metadati memorizzati sulla flash SPI, in formato JSON aperto. Questi metadati contengono:

- Un ID univoco per ogni scheda: indipendentemente dalla designazione della porta seriale che il sistema operativo assegna alla scheda, si è sempre certi di programmare quella giusta.
- Un nome in formato “human-readable” assegnato alla scheda (e alla FPGA).
- Informazioni su dove immagazzinare il programma utente ed eventuali dati opzionali.
- Un URL modificabile, per il recupero degli aggiornamenti del bootloader e del firmware.

Per i dettagli tecnici e il codice sorgente vedere il repository GitHub della TinyFPGA BX[10].

6 https://github.com/FPGAwars/icestudio
7 https://en.wikipedia.org/wiki/ICE_%28FPGA%29
8 https://github.com/tinyfpga/TinyFPGA-Bootloader
9 https://discourse.tinyfpga.com/t/i-i-am-using-tinyfpga-bx-for-the-first-time-i-have-questions/1080/2
10 https://github.com/tinyfpga/TinyFPGA-BX

1 Aggiornamento del bootloader

Per prima cosa occorre installare il programma “*tinyprog*”, verificare che la scheda sia riconosciuta dal PC ed eventualmente aggiornare il bootloader all’ultima versione. Le istruzioni indicate di seguito sono per Linux Arch e Manjaro, nel caso di altre distribuzioni Linux basta installare i pacchetti equivalenti, nel caso di Windows e MacOS vedere la guida di riferimento[11]:

- Verificare che l’utente appartenga al gruppo “*uucp*” (se così non fosse aggiungere l’utente a tale gruppo e ripetere il login). Notare che su alcune distribuzioni Linux potrebbe essere necessaria l’appartenenza al gruppo “*dialup*” anziché “*uucp*”.

- Installare “*tinyprog-1.0.23*” (o successivo) dai repository ufficiali della distribuzione Linux usata.

- Connettere la TinyFPGA BX al PC tramite un apposito cavo USB (fare attenzione che il minuscolo connettore USB è abbastanza delicato). Si raccomanda, se possibile, di inserire la scheda su una breadboard (dopo aver saldato gli header) per garantire una maggiore stabilità: la scheda da sola è molto leggera e la rigidità del cavo potrebbe spostarla facilmente portandola a contatto di eventuali parti metalliche presenti nelle vicinanze, con conseguente danneggiamento sia della TinyFPGA, sia del computer a cui essa è collegata.

- Nel terminale digitare “*dmesg*” (o “*sudo dmesg*” se necessario). Le ultime righe dovrebbero riportare i messaggi seguenti (con possibili variazioni secondo il kernel Linux usato):

  […] usb 2-1: new full-speed USB device number 2 using xhci_hcd
  […] usb 2-1: New USB device found, idVendor=1d50, idProduct=6130, bcdDevice= 0.00
  […] usb 2-1: New USB device strings: Mfr=0, Product=0, SerialNumber=0
  […] cdc_acm 2-1:1.0: ttyACM0: USB ACM device
  […] usbcore: registered new interface driver cdc_acm
  […] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters

- Digitare “*lsusb*” e cercare la linea contenente i valori 1d50:6130 (*idVendor:idProduct*, vedi sopra). La stringa descrittiva riportata dovrebbe essere “*OpenMoko, Inc*”.

- Digitare “*tinyprog -l*”. Dovrebbero essere visualizzate le seguenti linee:

  TinyProg CLI
  Using device id 1d50:6130
  Only one board with active bootloader, using it.
  Boards with active bootloaders:

  /dev/ttyACM0: TinyFPGA BX 1.0.0
  UUID: 626e3f5d-09a8-468f-a17c-6313e22488f9
  FPGA: ice40lp8k-cm81

  Notare che l’UUID è specifico di ogni scheda, mentre il valore 1.0.0 riportato indica la revisione della scheda, non del bootloader. Per verificare la versione di bootloader occorre stampare i metadati memorizzati nella flash SPI, tramite il comando “*tinyprog -m*”:

11 https://tinyfpga.com/bx/guide.html

```
[
 {
  "boardmeta": {
   "name": "TinyFPGA BX",
   "fpga": "ice40lp8k-cm81",
   "hver": "1.0.0",
   "uuid": "626e3f5d-09a8-468f-a17c-6313e22488f9"
  },
  "bootmeta": {
   "bootloader": "TinyFPGA USB Bootloader",
   "bver": "1.0.1",
   "update": "https://tinyfpga.com/update/tinyfpga-bx",
   "addrmap": {
    "bootloader": "0x000a0-0x28000",
    "userimage": "0x28000-0x50000",
    "userdata": "0x50000-0x100000"
   }
  },
  "port": "/dev/ttyACM0"
 }
]
```


- Se il termine "*bver*" nella sezione "*bootmeta*" non compare (oppure se è inferiore a 1.0.1) occorre effettuare l'aggiornamento del bootloader (occorre una connessione ad internet). Digitare il comando "*tinyprog --update-bootloader*" (notare il doppio trattino). Se il bootloader è già aggiornato sarà stampata come ultima riga la stringa "All connected and active boards are up to date!", altrimenti verrà scaricato ed aggiornato il bootloader[12].

- Notare che qualora si abbiano più TinyFPGA BX connesse al PC, il programma tinyprog provvederà a riportare i dati di tutte le schede e ad effettuare gli aggiornamenti dei bootloader nelle singole schede ove ciò è necessario. Per forzare l'indirizzamento di una specifica scheda basta specificare il suo UUID mediante l'opzione "*--id*" (bastano i primi 2 o 4 caratteri se già permettono di distinguere una scheda dall'altra).

- Qualora il bootloader venga corrotto (con conseguente scheda bloccata) è possibile ripristinarlo usando l'interfaccia SPI presente sulla faccia inferiore della TinyFPGA BX[13].

12 https://github.com/tinyfpga/TinyFPGA-Bootloader/tree/master/programmer
13 https://discourse.tinyfpga.com/t/bx-bootloader-bricked/852/2

2 TinyFPGA BX Quick Start

Per la programmazione della FPGA iCE40 presente sulla TinyFPGA BX esistono diverse soluzioni:

- il programma proprietario *iCEcube2* di Lattice: disponibile solo in versione a 32-bit per Windows e Linux, permette la programmazione della FPGA in Verilog e VHDL; l'ultima release è di Dicembre 2020, per cui l'installazione su distribuzioni Linux a 64-bit recenti potrebbe essere complicata per via di vecchie dipendenze a 32-bit necessarie; inoltre è necessario chiedere una licenza a Lattice Semiconductor (gratuita), che viene legata al MAC address della scheda di rete e va rinnovata periodicamente (ha una scadenza);

- l'ambiente open-source *Apio*: permette attualmente solo la programmazione in Verilog; la serie 0.8 ha diversi problemi per cui si consiglia di utilizzare una versione precedente;

- il mio makefile (disponibile in https://gitlab.com/flaviocappelli/fpga-code), assieme ai necessari tool open-source di simulazione e sintesi (vedere il repository); tale makefile fornisce anche la simulazione post-sintesi e tramite codice C++ (non disponibili in Apio);

- l'editor di schematici *Icestudio*, che si basa a sua volta su Apio; su Linux è fornito in un pacchetto AppImage;

Per semplicità di trattazione, in questo documento considereremo Apio. Esso sfrutta i progetti IceStorm e PrjTrellis, che mirano a documentare il formato bitstream di tutte le FPGA iCE40 ed ECP5, fornendo al contempo un ambiente completamente open-source per la loro programmazione.

NOTA: Vecchi esempi su internet riportano l'uso di Apio con l'editor Atom (attraverso l'estensione Apio-IDE). Lo sviluppo di Apio-IDE è fermo da tempo ed Apio > 0.4.0b5 non è più compatibile con tale estensione. Utilizzeremo quindi Apio in modo autonomo, lanciando i comandi da terminale. Per caricare un semplice esempio nella TinyFPGA BX procedere con i seguenti passi:

- Installare Apio, in particolare una versione compresa tra la 0.5.4 e la 0.7.6: se una tale versione non fosse disponibile per il sistema in uso (o non funzionante), utilizzare una VM (macchina virtuale) con un ambiente in grado di supportare tali versioni.

- Digitare "*apio install --all*" (notare il doppio trattino) per scaricare tutte le dipendenze necessarie (che saranno installate nella cartella "*.apio*", creata nella home dell'utente); notare che è necessario un collegamento ad Internet attivo.

- Entrare nella cartella "*~/.apio/packages/examples/TinyFPGA-BX/blinkSOS*".

- Connettere la TinyFPGA BX al sistema (e alla VM se necessario) e digitare "*apio upload*".

L'esempio verrà compilato (generando i file "*hardware.asc*", "*hardware.json*" e "*hardware.bin*"), quindi caricato nella scheda. In caso di successo, la scritta "SUCCESS" campeggerà in verde nel terminale e il LED sulla scheda inizierà a lampeggiare emettendo una semplice sequenza SOS.

Successivamente vedremo i principali comandi dell'ambiente Apio e apprenderemo come si crea, si sintetizza e si simula un semplice progetto in Verilog. Per il momento è utile osservare la struttura della cartella "*.apio/package/*"[14], che nel caso di Apio < 0.8.0 è la seguente:

14 https://github.com/FPGAwars/apio

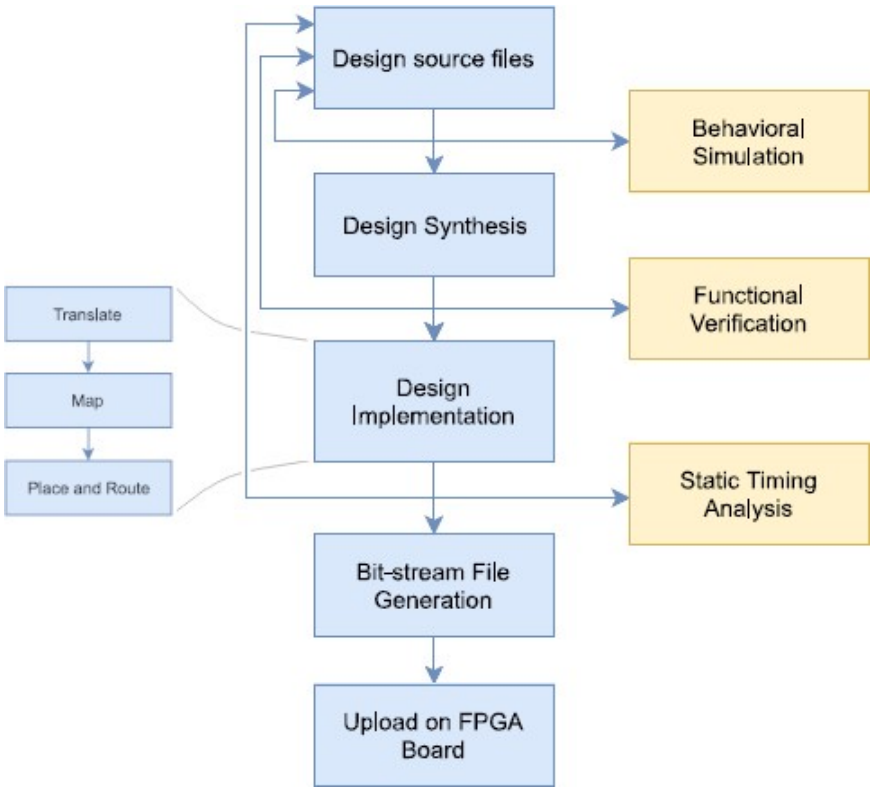
- examples – esempi e template per diverse schede, tra cui la TinyFPGA BX
- toolchain-ecp5 – tool di programmazione per FPGA Lattice ECP5 (dal progetto Trellis[15])
- toolchain-ice40 – tool di programmazione per FPGA Lattice iCE40 (dal progetto IceStorm[16])
- toolchain-iverilog – tool per sintesi e simulazione in Verilog (dal progetto Icarus[17])
- toolchain-verilator – simulatore C++ per Verilog (dal progetto Verilator[18])
- toolchain-yosys – tool per sintesi FPGA (dal progetto Yosys[19])
- tool-… - altre cartelle poco rilevanti

All'interno di ciascuna di tali sottodirectory è presente la cartella "*bin*" contenente gli eseguibili dei singoli pacchetti. Normalmente Apio richiama gli eseguibili necessari sulla base del flusso di lavoro, ma questi possono essere anche invocati direttamente se necessario (ad esempio "*.apio/package/toolchain-ice40/bin/icepll*" è molto utile per generare il modulo Verilog per scalare verso l'alto il clock di 16MHz). Notare che in Windows Apio crea anche le due seguenti cartelle:

- drivers – FPGA Windows driver (necessari per la programmazione)
- gtkwave – Visore per la simulazione in Verilog (dal progetto GTKWave[20])

In Linux e MacOS non servono driver e l'applicazione GTKWave deve essere installata manualmente tramite i tool di amministrazione dei pacchetti del sistema operativo in uso.

La sintesi di un sistema logico su FPGA segue il seguente flusso:

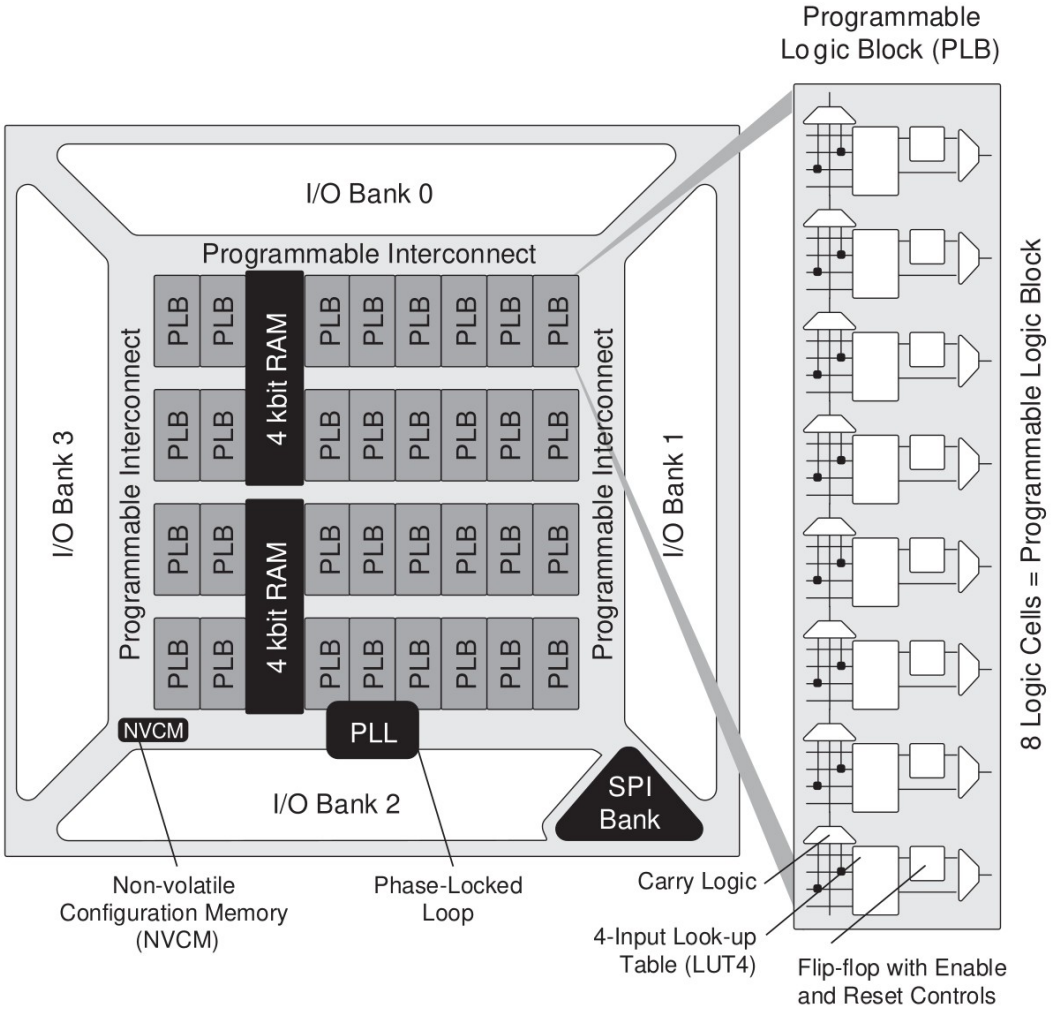


I tool sopra elencati intervengono in una o più fasi di tale flusso, come vedremo più avanti.

15 https://github.com/YosysHQ/prjtrellis
16 https://github.com/YosysHQ/icestorm
17 https://github.com/steveicarus/iverilog
18 https://www.veripool.org/wiki/verilator
19 https://github.com/YosysHQ/yosys
20 http://gtkwave.sourceforge.net/

3 Architettura della FPGA iCE40

I dispositivi iCE40 (il numero si riferisce alla tecnologia costruttiva a 40nm) sono costruiti come un array di blocchi logici programmabili (PLB), ove un PLB è un blocco di 8 celle logiche. Ogni cella logica è costituita da una tabella di associazione (lookup table) a quattro ingressi (chiamata 4-LUT o LUT4) con l'uscita collegata a un flip-flop di tipo D. All'interno di un PLB, ciascuna cella logica è collegata alla cella successiva e precedente mediante una logica di riporto (carry logic) intesa a migliorare le prestazioni di costrutti come sommatori e sottrattori. Intervallati con i PLB ci sono i blocchi di RAM, ciascuno di 4Kbit. Il numero di blocchi RAM varia a seconda del dispositivo (nel caso della iCE40 LP8K utilizzata per la TinyFPGA BX ci sono 32 blocchi, per un totale di 128Kbit di RAM). Tutti i blocchi sono collegati in orizzontale e verticale tramite canali di routing che vengono configurati dal design caricato in opportuno formato bitstream. Nella FPGA sono inoltre presenti uno o due PLL, un banco SPI dedicato all'acquisizione della configurazione iniziale e opzionalmente una memoria di configurazione NVCM non volatile.



Alla periferia del dispositivo, suddivise in un massimo di quattro banchi, si trovano invece le celle PIO. Mentre i PLB contengono gli elementi costitutivi per le funzioni di logica, aritmetica e registri, le celle PIO contengono dei buffer di I/O flessibili, in grado di supportare una varietà di interfacce standard. Su alcuni dispositivi iCE40 ogni banco ha il proprio pin di alimentazione (VCCIO_x, con x=0,1,2,3), che consente di utilizzare tensioni differenti per i banchi di I/O, in modo che la FPGA possa supportare contemporaneamente più standard di interfaccia (con livelli di tensione sempre compresi tra 1.8V e 3.3V). Comunque, nella TinyFPGA BX tutti e quattro i pin VCCIO_x sono

connessi alla sezione di alimentazione a 3.3V, quindi tutti i pin di I/O della scheda lavorano con la logica a 3.3V (standard LVCMOS33).

Struttura delle celle

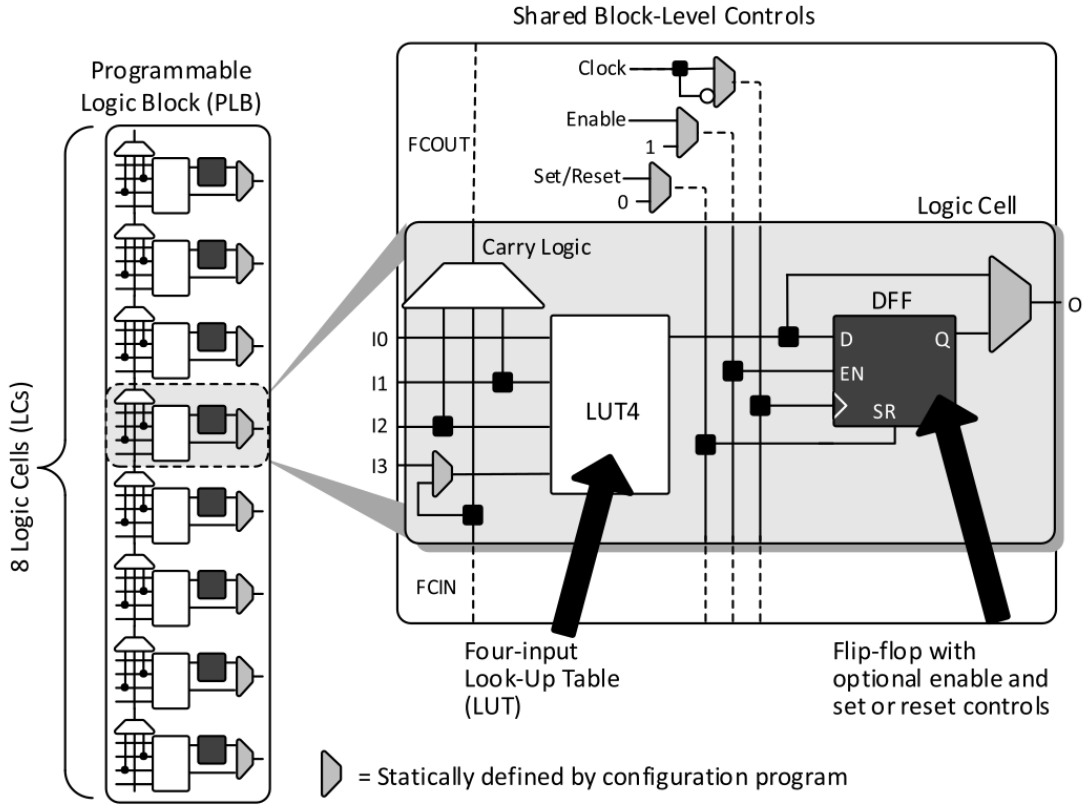
Si è visto brevemente che il nucleo della FPGA iCE40 è costituito da un'elevata quantità di blocchi logici programmabili, costituiti ciascuno da otto celle logiche interconnesse. Ogni cella include diversi elementi logici, come mostrato di seguito:

- Una look-up table, in grado di implementare una qualsiasi funzione logica combinatoria con un massimo di 4 ingressi (LUT4). Sostanzialmente la tabella si comporta come una ROM 16x1. Combinando in cascata più LUT4 è possibile creare funzioni logiche più ampie.

- Un flip-flop di tipo D, corredato di ingresso set/reset e di ingresso di abilitazione (enable), utilizzato per la realizzazione di funzioni logiche sequenziali. Ogni flip-flop è anche connesso ad un segnale di reset globale (non mostrato nella figura seguente) che viene attivato automaticamente subito dopo la configurazione della FPGA.

- Un segnale di "Carry Logic" il cui scopo è ottimizzare (in termini di celle utilizzate e tempo di propagazione) l'implementazione di alcune funzioni aritmetiche di base (quali sommatori, sottrattori e comparatori), contatori binari e ampie funzioni logiche in cascata. Notare che la iCE40 non ha moltiplicatori hardware dedicati (alcune FPGA, come la ECP5, li hanno).

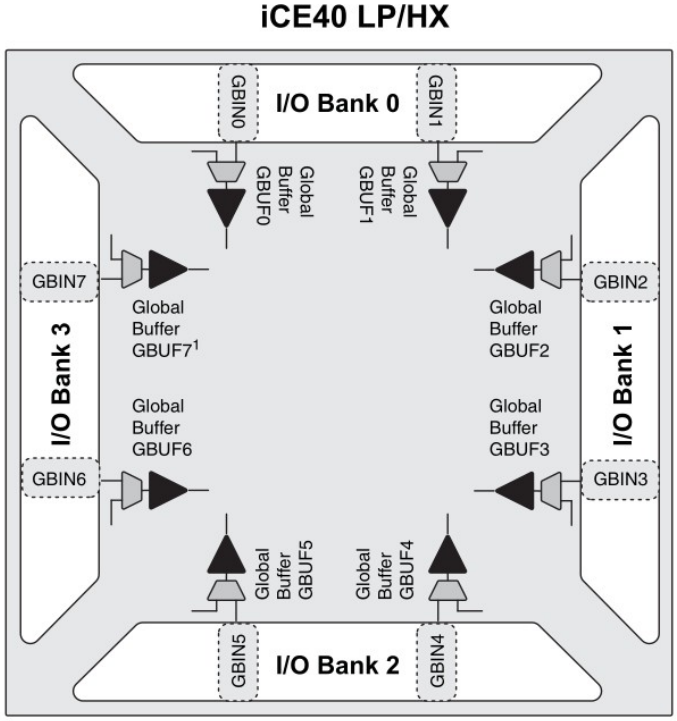
- Diversi multiplexer a 2 ingressi (elementi in grigio chiaro nella figura seguente) in cui l'ingresso selezionato è definito staticamente dalla configurazione caricata.



Come è possibile osservare nella figura sopra, il segnale di set/reset, quello di abilitazione (enable) ed il clock sono condivisi tra tutte le celle del medesimo PLB.

Global Buffer

La FPGA iCE40 dispone internamente di otto linee ad elevato "*fan-out*"[21] e basso "*skew*"[22], chiamati "*Global Buffer*" (GBUFx, con x=0..7). Tali linee sono progettate principalmente per la distribuzione del clock e di altri segnali (set/reset, enable, ecc) che devono poter pilotare molti ingressi logici con il minimo ritardo possibile. In ogni banco di I/O sono presenti ingressi specializzati per due Global Buffer:



Agli ingressi dei Global Buffer GBUFx possono essere applicati:

- l'uscita di un PLL;
- il corrispondente "*Global Buffer Input*" (GBINx/Gx) presente sul banco di I/O;
- una interconnessione interna;
- una cella PIO.

Il pin GBINx/Gx associato rappresenta il sistema migliore per pilotare un Global Buffer da una sorgente esterna e va quindi preferito (se possibile) rispetto alle altre celle PIO.

L'uso ottimale di un Global Buffer non è necessariamente automatico: sebbene il programma di allocazione delle risorse della FPGA cerchi di ottimizzare l'allocazione dei Global Buffer in funzione del fan-out utilizzato, lo sviluppatore potrebbe in alcuni casi effettuare scelte migliori. È sempre possibile forzare l'uso di un Global Buffer mediante la primitiva "SB_GB"[23] o tramite la

21 https://it.wikipedia.org/wiki/Fan-out
22 https://it.wikipedia.org/wiki/Skew_%28elettronica%29
23 https://www.mjoldfield.com/atelier/2018/02/ice40-blinky-icestick.html

"SB_GB_IO". Per i dettagli si rimanda al link fornito e al documento "*Lattice iCE Technology Library*" di Lattice Semiconductor[24].

Programmazione

I dispositivi iCE40 sono FPGA basate su SRAM. Le celle di memoria SRAM sono volatili, il che significa che una volta che l'alimentazione viene rimossa, la configurazione del dispositivo è persa e deve essere ricaricata alla successiva accensione. Questo approccio è utile in fase di prototipazione poiché permette di poter riprogrammare il dispositivo quante volte si vuole, ma richiede anche che le informazioni di configurazione siano memorizzate altrove e lette ogni volta che viene applicata l'alimentazione alla FPGA. Per tale motivo i dispositivi iCE40 hanno solitamente a fianco una memoria flash esterna per memorizzare tali dati di configurazione, oppure un microcontrollore in grado di trasferirli sulla FPGA. Alcuni dispositivi iCE40 possiedono una memoria OTP (One Time Programmable) non volatile interna chiamata NVCM: l'utilizzo della NVCM elimina la necessità di un chip di memoria flash (o microcontrollore), consente alla FPGA di configurarsi istantaneamente e migliora la sicurezza rendendo il reverse engineering più difficile, però impedisce qualsiasi upgrade del design o il riutilizzo della FPGA per altri progetti. La iCE40 LP8K della TinyFPGA BX include la memoria NVCM ma è meglio dimenticarsi di tale peculiarità!

Il caricamento della configurazione della FPGA dall'esterno può avvenire in due modi diversi, entrambi basati sull'interfaccia SPI:

- *Master SPI Configuration Mode*: la configurazione di funzionamento viene letta da una memoria flash esterna, a cui la FPGA accede dopo la fase di accensione;

- *Slave SPI Configuration Mode*: la configurazione viene trasferita da un microcontrollore, DSP o normale CPU quando ritenuto opportuno (nel frattempo la FPGA resta in attesa).

Di seguito è riportata la sequenza di configurazione standard:

1. all'inizio della configurazione tutta la SRAM viene azzerata, i pull-up abilitati, i pin di I/O posti in alta impedenza;

2. man mano che il download prosegue, i bit della SRAM vengono sostituiti gradualmente con i valori ricevuti (i nuovi bit hanno effetto immediato);

3. Alla fine del trasferimento, i pin di I/O vengono rilasciati al design caricato, eccetto i pin SPI che restano a disposizione della FPGA per ulteriori 49 cicli di clock prima di diventare I/O normali per l'applicazione.

Per maggiori dettagli sulla sequenza e i diversi metodi di configurazione si rimanda alla nota tecnica "*iCE40 Programming and Configuration*"[25].

La FPGA iCE40 può essere anche programmata direttamente dal kernel Linux[26]. Un driver apposito è stato infatti creato per permettere l'integrazione di tale FPGA in schede *Linux Embedded*.

24 http://www.latticesemi.com/view_document?document_id=52046
25 http://www.latticesemi.com/-/media/LatticeSemi/Documents/ApplicationNotes/IK/FPGA-TN-02001-3-2-iCE40-Programming-Configuration.ashx?document_id=46502
26 https://hackaday.com/2017/04/13/lattice-ice40-fpga-configured-by-linux-kernel/

I dispositivi iCE40 (come la maggior parte delle FPGA) sono progettati per essere programmati mediante un linguaggio di descrizione hardware (HDL). I dettagli del formato bitstream binario (che definisce come gli elementi interni dell'FPGA sono collegati e interagiscono tra loro) non sono generalmente pubblici dai produttori di FPGA, per cui gli sviluppatori sono costretti a utilizzare strumenti forniti dal produttore (che hanno limitazioni e talvolta sono anche vecchi e buggati). Per esempio, per i dispositivi iCE40, Lattice Semiconductor fornisce il programma ICEcube2 per Windows e Linux (ma non MacOS) che necessita di una licenza. Fortunatamente, per iCE40, esiste anche un toolchain completamente open-source costantemente aggiornato, composto da "Yosys" (front-end di sintesi di Verilog), "NextPNR" (generazione bitstream in formato testo) e "IcePack" (conversione da bitstream in formato testo a bitstream in formato binario)[27]. IcePack fa parte del progetto IceStorm[28] che mira a descrivere il formato bitstream binario di tutte le FPGA iCE40.

È stato dimostrato come sia possibile, con tale toolchain, costruire un SoC RISC-V su un dispositivo iCE40 HX8K perfettamente funzionante. Il toolchain supporta diversi dispositivi iCE40, tra cui LP8K (presente sulla TinyFPGA BX) ed è completamente integrato nell'ambiente Apio (che ha il pregio di rendere molto semplice l'uso del toolchain). In alternativa, l'intero toolchain può essere utilizzato mediante il comando "make"[29] o tramite il sistema di build "CMake"[30].

Conclusioni

La FPGA iCE40 è un dispositivo low-end a basso costo, che può essere usato per la prototipazione e la produzione di piccoli volumi. Per i grandi volumi (milioni di pezzi) non è mai conveniente utilizzare una FPGA poiché essa avrà sempre un costo nettamente superiore ad ogni altro circuito integrato di pari complessità logica. Infatti, a parità di numero di gate e flip-flop a disposizione per l'applicazione, una FPGA ha bisogno di molto più spazio e logica dedicata alle interconnessioni programmabili, che ne aumentano notevolmente il costo (esistono FPGA che superano i 100000$). È quindi prassi comune, per i grandi volumi, utilizzare una FPGA nella fase di sviluppo e poi realizzare la produzione vera e propria mediante ASIC (Application-Specific Integrated Circuit). Notare che, per i piccoli volumi, gli ASIC non sono convenienti a causa del costo di progettazione del processo di produzione, che seppur non ricorrente è in genere molto elevato (milioni di $)[31].

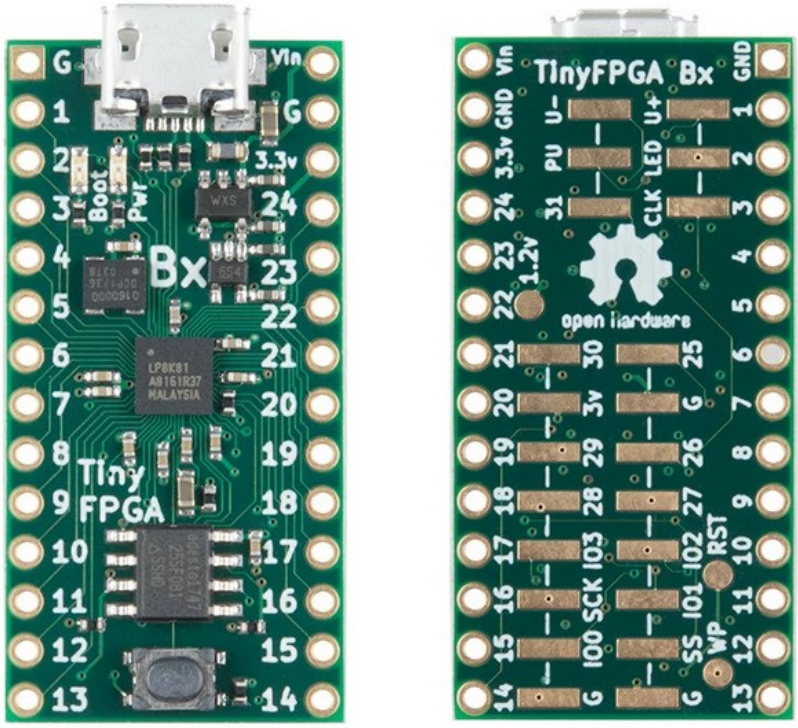
Per semplificare il design e rendere la serie iCE40 più economica, Lattice Semiconductor ha scelto di implementare le funzioni logiche mediante tabelle LUT4. Rispetto alle costose FPGA basate su tabelle LUT6 (come i dispositivi della serie Xilinx 7 e gli Altera Stratix), le FPGA basate su LUT4 possono necessitare di più celle per implementare le stesse funzioni logiche, specie se complesse. Ad esempio, una funzione logica con sette ingressi necessita di due sole LUT6 ma richiede due o più LUT4. Si è visto che, in media, un progetto complesso realizzato con tabelle LUT4, richiede dal 15% al 20% di celle in più rispetto all'implementazione con LUT6. Le LUT4 sono però più piccole e richiedono meno energia: la miniaturizzazione sta aumentando il numero di LUT4 per die e, grazie ad algoritmi di mapping più efficaci in grado di sfruttare meglio le LUT4, le tabelle LUT6 stanno gradualmente perdendo il loro originale vantaggio[32].

Nonostante la semplice struttura la iCE40 è un'ottima FPGA e il suo basso costo la rende ideale come strumento per iniziare a conoscere il mondo delle logiche programmabili e per implementare circuiti logici di controllo, piccoli microcontrollori personalizzati e semplici game console.

27 https://hackaday.com/2015/12/29/32c3-a-free-and-open-source-verilog-to-bitstream-flow-for-ice40-fpgas/
28 https://github.com/YosysHQ/icestorm
29 https://github.com/nesl/ice40_examples
30 https://www.walknsqualk.com/post/015_fpga_design_p2/
31 https://en.wikipedia.org/wiki/Application-specific_integrated_circuit
32 https://people.eecs.berkeley.edu/~alanmi/publications/2018/fpga18_s44.pdf

4 Scheda TinyFPGA BX in dettaglio

Di seguito sono riportate le informazioni essenziali al corretto utilizzo della TinyFPGA BX. Come è possibile vedere nelle foto seguenti, la scheda fornisce una parte di segnali ed alimentazioni sugli header laterali, e il restante su piazzole SMD presenti sul retro. Qui prenderemo in considerazione l'uso della scheda su breadboard, per cui esamineremo principalmente i segnali e le alimentazioni presenti sugli header. Qualora fosse necessario, si può sempre saldare (con attenzione) un flat cable sulle piazzole SMD necessarie, per portare i relativi segnali alla breadboard.



La TinyFPGA BX può essere alimentata direttamente attraverso il connettore USB (+5V) oppure applicando una tensione da +3.5V a +5.5V sul pin serigrafato "Vin" (a destra del connettore USB).

Sulla scheda viene generata una tensione stabilizzata di +3.3V, utilizzata per alimentare le sezioni di I/O della FPGA. Tale tensione è inoltre disponibile sul pin serigrafato "3.3v" (secondo pin dopo Vin). Sulla documentazione della TinyFPGA BX è indicato di non prelevare più di 100mA da tale pin, probabilmente per mantenere il dropout del regolatore MIC5504-3.3YM5-TR sotto i 200mV. Ma garantendo almeno +3.7V in ingresso sul pin "Vin" dovrebbe essere possibile prelevare ben più di 100mA, in funzione della corrente usata dalle porte di I/O della FPGA.

Sulla scheda viene inoltre generata una tensione stabilizzata di +1.2V necessaria per alimentare il core della FPGA. Tale tensione è disponibile sulla piazzola SMD circolare serigrafata "1.2v" (vicino al logo "open hardware"). La massima corrente che può essere prelevata da tale pin è 100mA. Infatti, il regolatore MIC5365-1.2YC5-TR è in grado di fornire un massimo di 150mA e 50mA sono riservati alla FPGA. Sullo schematico tale piazzola è indicata con "TEST TP3".

Le connessioni di massa sono disponibili sui due pin degli header e tre piazzole SMD poste sul retro, tutti serigrafati con "G".

Il clock di 16MHz generato dall'oscillatore DSC6001CI2A-016.0000T è applicato al pin B2 della FPGA (IOL_2B); è inoltre riportato sulla piazzola SMD serigrafata "CLK".

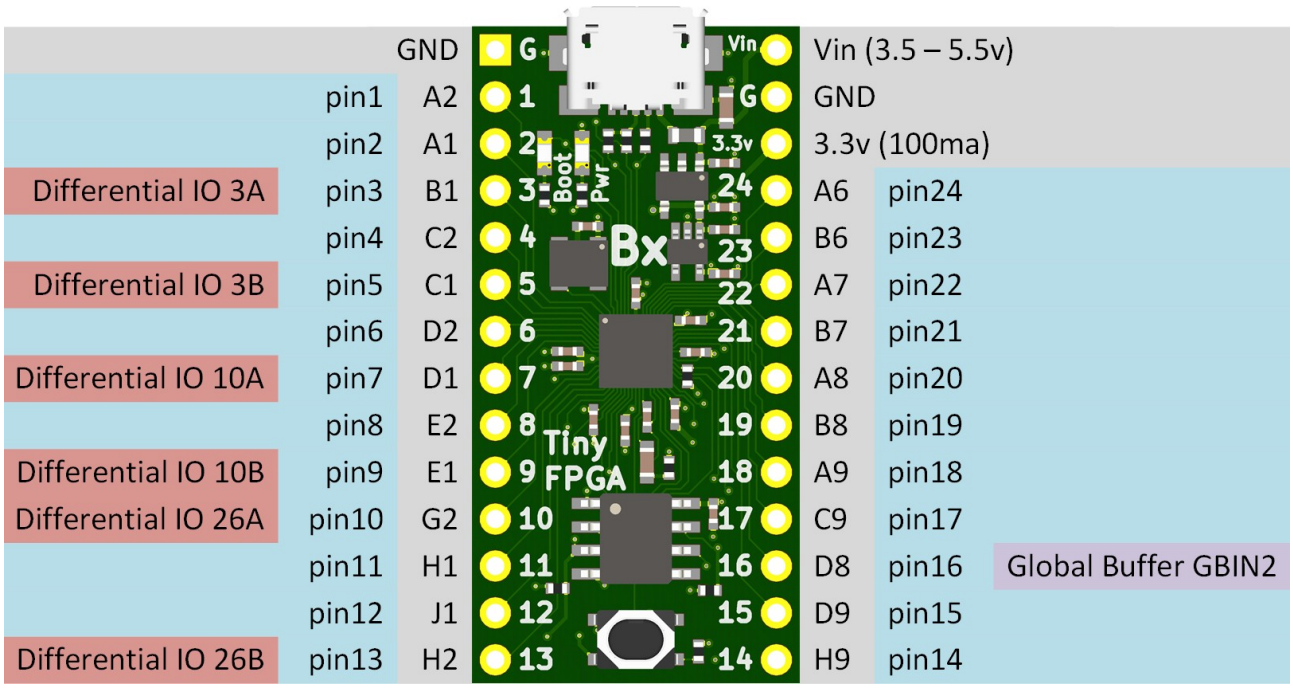
Sulla scheda sono presenti due LED, uno connesso direttamente ai +3.3V ed utilizzato per indicare presenza di alimentazione, l'altro connesso al pin B3 della FPGA (IOT_218) ed utilizzato sia per il bootloader che per il design utente. Tale segnale è riportato sulla piazzola SMD serigrafata "LED".

Sul retro della scheda è presente un segnale di reset (piazzola SMD circolare serigrafata "RST"), ma dallo schematico ("TP2 TEST") si vede che tale segnale, normalmente alto (+3.3V), viene reso basso (GND) solo dal pulsante SW1 che è usato per riattivare il bootloader dopo che un design è stato caricato. Non è quindi un vero segnale di reset che può essere usato da logiche esterne, in quanto non è temporaneamente attivo dopo che viene fornita l'alimentazione e non è filtrato da logica antirimbalzo. Comunque, è possibile generare un vero segnale di reset tramite la FPGA[33].

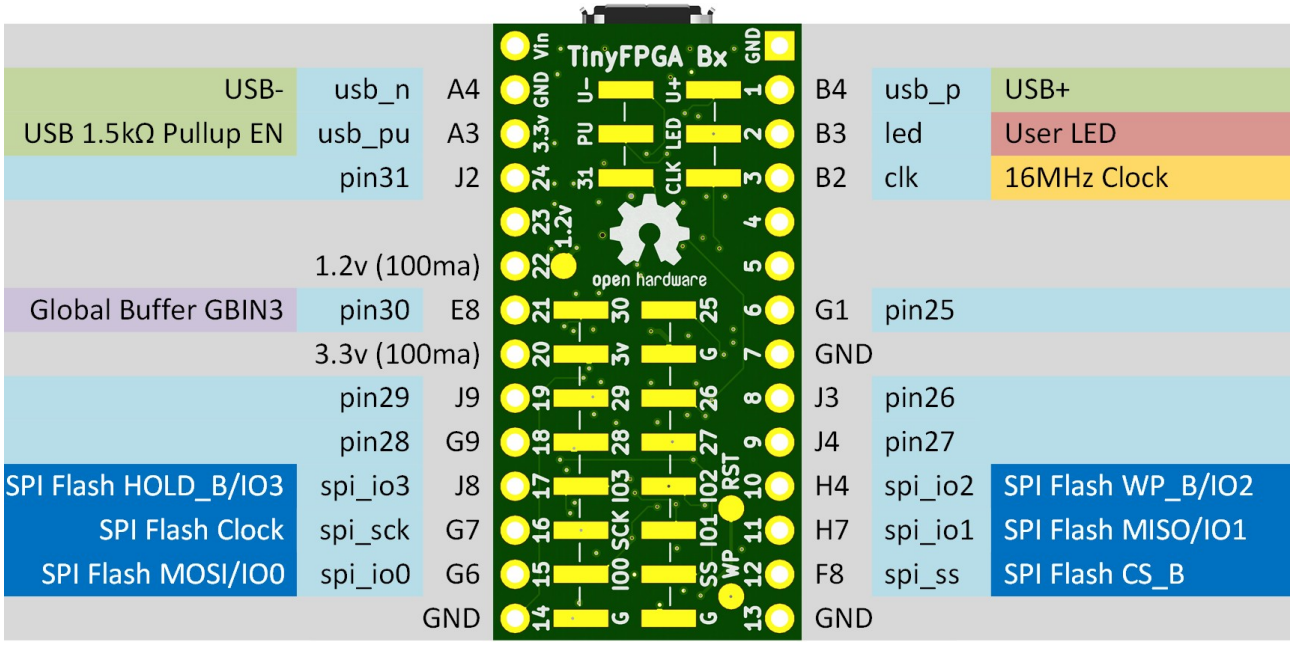
Come si può vedere dalle figure seguenti, sugli header sono riportati 24 I/O liberi della FPGA mentre altri 7 sono disponibili sul retro, per un totale di 31 I/O utilizzabili. Sul retro della TinyFPGA BX sono anche riportati i 10 I/O già utilizzati (USB_N, USB_PU, USB_P, LED, SPI_IO3, SPI_CLK, SPI_IO0, SPI_IO2, SPI_IO1, SPI_SS) per un totale di 41 I/O.

Anche se la TinyFPGA BX può essere alimentata a +5V, i pin di I/O lavorano a 3.3V e tollerano una tensione massima di 3.5V (standard LVCMOS33, vedere il datasheet della FPGA). Non connettere mai la TinyFPGA BX con logiche a 5V senza un opportuno circuito di interfaccia, né applicare direttamente i +5V agli ingressi della scheda, altrimenti si causeranno gravi danni alla FPGA. Inoltre, i pin di I/O della FPGA possono sopportare una corrente massima di 8mA.

Per l'interfacciamento sono possibili soluzioni discrete sia per 3.3V → 5V[34], che per 5V → 3.3V[35], ma l'approccio migliore è utilizzare integrati appositi come il 74LVCH1T45 o il 74LVCH8T245 che permettono la conversione bidirezionale alla massima velocità e basso carico per le porte.



33 https://stackoverflow.com/questions/38030768/icestick-yosys-using-the-global-set-reset-gsr
34 https://next-hack.com/index.php/2020/02/15/how-to-interface-a-3-3v-output-to-a-5v-input/
35 https://next-hack.com/index.php/2017/09/15/how-to-interface-a-5v-output-to-a-3-3v-input/



Sia il bootloader che il circuito applicativo vengono salvati sulla flash SPI AT25SF081-SSHD-B. Tale memoria garantisce 100.000 cicli di scrittura/cancellazione e una durata di ritenzione di 20 anni. Il consumo della flash SPI, in fase di lettura è di soli 4mA, che si riducono a 20uA in standby. La frequenza operativa massima di tale memoria è intorno ai 100MHz. Per il funzionamento della flash SPI si rimanda al relativo datasheet.

Comportamento del bootloader

Il bootloader si comporta diversamente a seconda di cosa è collegato alla porta USB:

- Se la scheda è collegata a un host tramite cavo USB dati, il bootloader rimane attivo in attesa che la scheda venga programmata da "*tinyprog*". È possibile forzare l'uscita dal bootloader e caricare l'immagine di un'applicazione già presente nella flash SPI, mediante il comando "*tinyprog -b*".

- Se la scheda è semplicemente alimentata via USB (ad esempio con una powerbank oppure tramite cavo senza linee dati) o attraverso il pin Vin, il bootloader va in timeout dopo un secondo, caricando l'immagine dell'applicazione dalla flash SPI.

Premendo il pulsante di ripristino, la scheda riattiverà sempre il bootloader, ma essa rimarrà nel bootloader (in attesa di un nuovo firmware) solo se è collegata a un host con un cavo USB dati.

Pull-up

Su alcuni pin della FPGA iCE40 (e quindi della TinyFPGA BX) è possibile abilitare dei resistori di pull-up. Ci sono due modi diversi per farlo[36]: modificando il file .pcf (aggiungendo "*-pullup yes*" a "*set_io*") oppure usando la direttiva "*SB_IO*". Per quest'ultimo caso si rimanda al link fornito e al documento "*Lattice iCE Technology Library*"[37]. Notare che tali pull-up sono statici, non possono essere modificati dopo il caricamento della configurazione della FPGA. Inoltre, solo i banchi di I/O denominati 0,1,2 hanno i resistori di pull-up (sul banco 3 l'abilitazione viene ignorata).

36 https://discourse.tinyfpga.com/t/internal-pullup-in-bx/800
37 http://www.latticesemi.com/view_document?document_id=52046

5 Simulazione e programmazione della TinyFPGA BX

Di seguito vengono elencati i principali passi per creare e simulare un progetto per la FPGA iCE40 (in particolare per la scheda TinyFPGA BX) mediante l'ambiente Apio:

- Creare una cartella pulita per il progetto e configurare l'ambiente Apio per la TinyFPGA:

  apio init --board TinyFPGA-BX

  Nota: con il comando "*apio boards --list*" è possibile vedere tutte le schede supportate.

- Copiare il file "*TinyFPGA-BX-pins.pcf*" presente nella directory dell'esempio blinkSOS nella cartella appena creata (esso rende noti i pin della TinyFPGA BX al codice Verilog).

- Creare uno o più file "**.v*" di progetto con il codice Verilog necessario.

- Verificare, mediante Icarus Verilog, che nel codice non vi siano errori grossolani:

  apio verify

- Verificare, mediante Verilator, la correttezza del codice (controllo più formale):

  apio lint

- Creare un modulo "*testbench*" per simulare il progetto mediante Icarus Verilog e GTKWave. Notare che il file del testbench deve terminare con "*_tb.v*" altrimenti verrà considerato parte del progetto da sintetizzare; inoltre nel testbench deve essere istanziato un unico modulo, gerarchicamente superiore a tutti (non è possibile istanziare più moduli nel testbench).

- Avviare la simulazione mediante il comando:

  apio sim

  Al termine della simulazione verrà avviato GTKWave (vedere più avanti).

- Generare il file di analisi temporale "*hardware.rpt*":

  apio time

  L'ispezione visiva di tale file è essenziale per verificare che il tempo di propagazione dei segnali attraverso la logica sintetizzata sia compatibile con la durata del clock.

- Sintetizzare il progetto mediante gli Icestorm Tools:

  apio build -v

  L'opzione *-v* visualizza il processo di build e riporta la quantità di risorse FPGA usate (collegamenti, celle, LUT4, flip-flop, unità di riporto, blocchi di memoria, global buffer, ecc) prima da "*yosys*" (non ottimizzate) e poi da "*nextpnr-ice40*" (ottimizzate e mappate sulla iCE40 L8K). Nota: le info utili vanno cercate in mezzo alle molte linee visualizzate!

- Connettere la TinyFPGA BX al PC e caricare il codice bitstream mediante tinyprog:

  apio upload

- Eliminare i file non necessari, generati dai precedenti comandi:

  apio clean

Note sulla simulazione

All'inizio GTKWave si avvia senza visualizzare alcuna informazione. Il motivo è che in un progetto possono essere presenti anche centinaia di moduli e migliaia di segnali diversi, è pertanto compito dello sviluppatore decidere quali visualizzare. A tale scopo, in GTKWave:

- cliccare sul modulo "*testbench*" nella finestra in alto a sinistra (sotto la voce SST);
- nella finestrella sotto, verranno evidenziati i segnali definiti in tale modulo; trascinare quelli desiderati in "*Waves*" (finestra a destra);
- allargare l'applicazione GTKWave alla dimensione desiderata, eventualmente fino a ricoprire l'intera area a disposizione;
- nella toolbar cliccare sul pulsante "*Zoom Fit*";
- aggiungere i segnali di eventuali altri moduli del progetto che si intende esaminare (allargare il modulo "testbench" e gli eventuali sottomoduli);
- sulla tastiera digitare *CTRL-S* (o cliccare sulla voce di menu "*File*" → "*Write Save File*") per salvare un file "**_tb.gtkw*" con le impostazioni: in tal modo, se la simulazione viene riavviata, l'applicazione GTKWave visualizzerà automaticamente i segnali scelti con la scala temporale selezionata.

NOTA: se vengono cambiati i nomi dei segnali o la loro dimensione (qualora si tratti di vettori) occorrerà aggiungerli di nuovo al widget "*Waves*".

Se nella simulazione i segnali non vengono propagati tra i moduli, verificare che la loro istanziazione sia corretta. Il classico formato della chiamata a funzione non va mai usato:

```
module_template MODNAME (       /* ISTANZIAZIONE ERRATA */
        wire1,
        wire2,
        ...
);
```

Tutti gli argomenti in Verilog devono essere forniti con il seguente formato:

```
module_template MODNAME (       /* ISTANZIAZIONE CORRETTA */
        .arg1( wire1 ),
        .arg2( wire2 ),
        ...
);
```

6 Brevi note sulla sintesi hardware in Verilog

Verilog è un linguaggio un po' ambiguo, nato per portare gli sviluppatori C nel mondo della sintesi hardware. Ha subito diverse revisioni (l'ultima è la IEEE 1364-2005) che ne hanno incrementato il numero di costrutti linguistici. Chi si avvicina per la prima volta a Verilog e al mondo della sintesi hardware resta inizialmente confuso dal fatto che alcune operazioni possono essere eseguite in modi diversi, alcuni costrutti sono obsoleti o sconsigliati, alcuni termini utilizzati nella descrizione del linguaggio sono disorientanti. Scopo di tale sezione, è quello di chiarire comuni perplessità che sorgono iniziando a studiare il Verilog. Notare che esiste anche un altro linguaggio utilizzato nella sintesi hardware, il VHDL: esso è più formale (ma anche più prolisso), ed è particolarmente apprezzato in ambienti militari, aerospaziali e mission-critical. I due linguaggi sono praticamente equivalenti e nessuno dei due ha un reale predominio sull'altro. Qui non parleremo del VHDL.

Le prossime pagine riportano alcune osservazioni su argomenti che personalmente ho trovato poco chiari durante il mio approccio al Verilog. La sezioni seguenti non costituiscono un corso di Verilog, pertanto si consiglia di leggere una introduzione a Verilog 2005 prima di proseguire.

Modelli "Behavioural", "RTL" e "Gate Level"

Nello studio di Verilog e VHDL si incontrano spesso tali termini, quindi vediamo cosa significano:

- Con modello "*Behavioural"* si intende codice che descrive il comportamento di un sistema ad alto livello, adatto alla simulazione, ma non alla sintesi. Tale codice può essere utilizzato come primo passo per descrivere il progetto nella sua interezza, prima che venga realizzato il codice "*RTL*" dettagliato dei singoli moduli. Ma viene anche utilizzato per il "*testbench*", cioè l'ambiente che circonda il progetto in una simulazione (la parte che simula gli stimoli e permette di rappresentare in modo comprensibile i risultati). L'output del testbench è solitamente costituito da diagrammi temporali, o rappresentazioni visive di più alto livello, a seconda del progetto che si sta realizzando (ad esempio, se il progetto prevede un'uscita video su VGA o HDMI, il testbench potrebbe anche simulare un monitor).

- Con modello "*RTL"* (Register Transfer Level) si intende codice sintetizzabile, solitamente suddiviso in moduli, in grado di descrive il comportamento del sistema in termini di logica combinatoria e sequenziale di medio livello (registri, mux/demux, rom, ram, ecc). Il codice RTL è sempre simulato per verificare la correttezza del progetto (*simulazione pre-sintesi*), ma serve anche come input per la sintesi, per generare codice "*Gate Level*" (vedi dopo). Nell'RTL non sono considerati ritardi espliciti e le temporizzazione sono determinate dai fronti del clock. Le strutture "latch" (attivate da livelli anziché dai fronti) sono generalmente evitate, i registri sono sincronizzati con uno o più clock, e si presume che la propagazione della logica combinatoria si stabilizzi all'interno di un singolo ciclo di clock. I costrutti utilizzabili nella descrizione RTL sono soltanto un sottoinsieme specifico di quello che Verilog o VHDL mettono a disposizione, poiché gran parte di questi linguaggi è dedicata esclusivamente alla simulazione behavioural (Verilog e VHDL sono nati inizialmente come strumenti per la simulazione e solo successivamente sono stati applicati alla sintesi).

- Con modello "*Gate Level*" si intende codice che descrive il sistema in termini di flip-flop, porte logiche e altri costrutti di base che possono essere direttamente mappati sugli elementi forniti da FPGA ed ASIC. Il modello "Gate Level" è di livello più basso rispetto a quello RTL: infatti dalla sintesi del codice RTL si ottiene in pratica codice "Gate Level". La simulazione di tale codice (*simulazione post-sintesi*) è fondamentale per verificare che la

sintesi sia stata eseguita correttamente, ma generalmente richiede molto più tempo rispetto alla simulazione RTL (poiché operazioni che prima venivano eseguite con una sola istruzione, ad esempio una moltiplicazione tra numeri, sono ora suddivise nella simulazione di decine, centinaia o migliaia di componenti elementari).

Blocchi “always” e “initial”

Un blocco in Verilog racchiude una sezione di codice a se stante, simulata e sintetizzata in maniera indipendente dal codice presente negli altri blocchi. In pratica, ogni blocco viene sintetizzato su hardware diverso e simulato in un processo indipendente e parallelo agli altri. I blocchi in Verilog si dividono in due grandi categorie: ripetitivi e non ripetitivi. Quelli ripetitivi sono definiti da:

- *always @ (<sensitivity list>)*
  block_of_statements;

- *initial forever*
  block_of_statements;

- *initial while(1)*
  block_of_statements;

- *initial begin forever*
  block_of_statements;
  end

Per quanto riguarda la simulazione, le quattro definizioni sono del tutto equivalenti, ma nella sintesi viene accettato solo “*always*”, per cui all’interno dei moduli da sintetizzare occorre utilizzare questo costrutto. Notare che “*forever*” e “*while(1)*” (che esprimono un ciclo infinito) possono essere utilizzati anche nel corpo di una condizione “*if*” ma solo in simulazione, in quanto non è possibile “srotolare” un ciclo infinito per trasformalo in un gruppo di circuiti logici funzionanti in parallelo (mentre ciò può essere fatto per un ciclo finito, entro certi limiti). Occorre sempre tenere a mente che il codice Verilog rappresenta della logica hardware e non un programma che viene eseguito!

Per quanto riguarda i blocchi non ripetitivi, essi sono eseguiti una sola volta e sono definiti da:

- *initial begin*
  block_of_statements;
  end

Per terminare la simulazione dopo un po' di tempo, si deve usare la parola chiave “*$finish*” alla fine del blocco non ripetitivo: essa provoca l'interruzione di tutti i blocchi, compresi quelli ripetitivi. Il testbench di solito possiede un blocco non ripetitivo contenente uno o più ritardi (*#N*) prima della parola chiave “$finish”: la somma di tutti i ritardi determinerà la durata della simulazione.

Tipi “reg” e “wire”

Tutte le variabili e i registri assegnati dentro ai blocchi “*always*” (o “*initial*”) devono essere di tipo “*reg”*; tutte le variabili e i segnali assegnati al di fuori dei blocchi “*always*” (o “*initial*”) devono essere di tipo “*wire”*. Notare che la keyword “*reg”* non comporta necessariamente l’istanziazione di un “registro” hardware. Ingressi e uscite senza definizione di tipo sono considerate di tipo “*wire*”.

Assegnazioni ed eventi

L'assegnazione in Verilog è di due tipi:

- "*Assegnazione non bloccante*" (<=): caratterizza le assegnazioni che vengono eseguite tutte simultaneamente quando si verifica l'evento scelto (o gli eventi scelti) come trigger. È normalmente usata con il tipo "*reg*" per descrivere la logica sequenziale, ad esempio:

```
reg ff1, ff2;
always @ (posedge clk) begin
        ff1 <= ff2;
        ff2 <= ff1;
end
```

  Al trigger (fronte positivo del clock) il valore di ff2 è "catturato" da ff1; nello stesso momento ff1 è "catturato" da ff2. Le due assegnazioni sono quindi eseguite "in parallelo". Ciò avviene in quanto, essendo ff1 ed ff2 definiti come tipi "reg", ed essendo assegnati mediante assegnazioni non bloccanti, vengono automaticamente implementati come uscite di due flip-flop di tipo D. Come accennato è possibile specificare più eventi trigger:

```
always @ (posedge clk, negedge reset) begin
        if (~reset) begin
                ff1 <= 1'b0;
                ff2 <= 1'b0;
        end else begin
                ff1 <= ff2;
                ff2 <= ff1;
        end
end
```

  Ora, il fonte positivo del clock e il fronte negativo del reset, causano entrambi l'esecuzione del blocco, e l'istruzione "*if (~reset)*" discrimina la fase di reset (azzeramento dei flip-flop) dal funzionamento normale. In pratica abbiamo aggiunto un reset asincrono (attivo basso, poiché il reset è effettuato quando "*~reset*" è vero, cioè quando "*reset*" è falso, cioè a 0V).

- "*Assegnazione bloccante*" (=): normalmente usata con il tipo "*wire*" per definire la logica combinatoria, essa blocca l'esecuzione (o l'interpretazione) dell'istruzione successiva fino al "completamento". In termini hardware e di simulazione ciò significa che il risultato dell'istruzione successiva può dipendere da quella corrente. Ad esempio la sequenza:

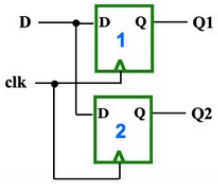
```
c = a & b;
e = c ^ d;
```

  si traduce in `e = (a & b) ^ d`. Si può dunque pensare alle assegnazioni bloccanti come interpretate "in serie", specialmente se vi è una relazione di dipendenza tra le espressioni (altrimenti sono eseguite in parallelo). A livello circuitale l'assegnazione bloccante definisce una propagazione (quasi) istantanea e continua del valore attraverso le porte logiche.

Notare che le due tipologie di assegnazioni, quando usate su logica sequenziale, determinano in molti casi dei circuiti completamente diversi, come evidenziato nel seguente esempio:

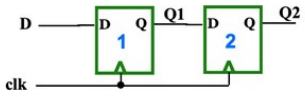
Blocking Statement:

```
module block (
   input D, clk,
   output reg Q1, Q2
);

   always @ (posedge clk)
   begin
      Q1 = D;
      Q2 = Q1;
   end
endmodule
```



Non-Blocking Statement:

```
module block (
   input D, clk,
   output reg Q1, Q2
);

   always @ (posedge clk)
   begin
      Q1 <= D;
      Q2 <= Q1;
   end
endmodule
```



Occorre quindi fare molta attenzione: una semplice distrazione può trasformare una assegnazione non bloccante in una bloccante stravolgendo il circuito (basta dimenticarsi il '<'). Come regola generale è bene non utilizzare mai l'assegnazione bloccante all'interno dei blocchi "*always*" che abbiano fronti di salita e discesa come parametri della "*sensitivity list*".

La "*sensitivity list*" può essere anche composta da un elenco di segnali privi degli specificatori "*posedge*" o "*negedge*". In tal caso il blocco "*always*" definisce della logica combinatoria oppure sequenziale asincrona (esso va considerato come un blocco attivo continuamente). Ad esempio il seguente codice implementa un semplice latch con doppia uscita "q" e "q_n" (q negato):

```
reg q;
always @ (enable, d) begin
        if (enable) begin
                q    <= d;
                q_n <= ~d;
        end
end
```

(notare che l'utilizzo di latch è sconsigliato, è bene preferire sempre la logica sincrona a quella asincrona). Il prossimo esempio definisce un multiplexer a 4 ingressi (a,b,c,d → y):

```
reg y;
wire [1:0] sel;          // input selection
wire a, b, c, d;
always @ (sel, a, b, c, d) begin
        case (sel)
                2'b00 : y = a;
                2'b01 : y = b;
                2'b10 : y = c;
                2'b11 : y = d;
        endcase
end
```

Come detto in precedenza, tutte le variabili assegnate all'interno di un blocco vanno sempre definite di tipo "reg" (tale tipo non comporta necessariamente l'istanziazione di un registro hardware).

Si noti che per modellare la logica combinatoria nei blocchi si può usare sia l'assegnazione bloccante che quella non bloccante, ma è preferibile usare quella bloccante in quanto più leggibile (e in alcuni tool di sintesi buggati anche più efficiente). Seguendo questa regola, basta un istante per distinguere un blocco con memoria da uno con sola logica combinatoria.

Dichiarazione "assign"

Un altro modo per definire della logica combinatoria (senza latch) è tramite la dichiarazione "*assign*", ad esempio:

```
assign {c_out, sum} = a + b + c_in;
```

Essa va sempre usata al di fuori dei blocchi "*always*" e "*initial*". Tutte le dichiarazioni "*assign"* vanno interpretate come eseguite in parallelo a meno che non vi siano relazioni di dipendenza.

Qualche regola fondamentale

Nell'uso delle assegnazioni ci sono alcune regole da seguire obbligatoriamente[38]:

- Mai mescolare assegnazioni bloccanti e non bloccanti sulla stessa variabile.

- Mai assegnare la stessa variabile in due blocchi diversi in quanto ciò causa un conflitto (due entità diverse che tentano di imporre il loro valore alla stessa risorsa fisica). La regola vale ovviamente per entrambe le assegnazioni. Se una risorsa deve dipendere da due blocchi diversi, occorre utilizzare due variabili distinte (una per ogni blocco) e definire con la keyword "*assign"* una terza variabile esterna ai blocchi (di tipo "*wire")*, il cui valore è dato da un'opportuna logica combinatoria operante sulle altre due variabili, ad esempio:

```
always @ (posedge clk1)
        cnt1 <= cnt1 + 1;

always @ (posedge clk2)
        cnt2 <= cnt2 – 1;

assign out = cnt1 ^ cnt2;
```

- Notare che la lettura in un blocco, di una variabile modificata in un altro blocco, non pone problemi se le due azioni avvengono in momenti sicuramente distinti (tenuto conto dei ritardi di propagazione), ad esempio uno sul fronte positivo e l'altro su quello negativo dello stesso clock. Qualora non fosse possibile garantire questa condizione (ad esempio perché i due blocchi vengono attivati da eventi asincroni) e vi è il rischio che la variabile letta venga anche modificata nello stesso momento allora va rivisto il design del progetto.

- Si noti che non si dovrebbe mai attivare lo stesso blocco sia sul fronte positivo sia su quello negativo dello stesso segnale. Questo perché in genere le FPGA non possiedono flip-flop "*Double Edge Triggered*" (attivi su entrambi i fronti). Solitamente gli strumenti di sintesi riconoscono questa incongruenza ma è sempre bene evitare la condizione.

38 http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf

La violazione di queste regole può comportare errori nella sintesi del circuito, malfunzionamenti o differenze di comportamento tra simulazione e sintesi.

Inizializzazione dei registri

L'inizializzazione di registri (flip-flop) in fase di dichiarazione (oppure mediante blocco *initial begin…end*) è possibile, se la sintesi ha come obiettivo una FPGA basata su SRAM. Ad esempio, la seguente dichiarazione indica al compilatore che il valore del registro all'accensione deve essere 5:

```
reg [3:0] count = 4‘b0101;
```

Purtroppo tale pratica non è portabile[39]. Essa funziona perfettamente su FPGA iCE40 ed ECP5 con i tool open-source descritti in precedenza, mentre non funziona con iCEcube2, che ignora i valori specificati e forza sempre l’inizializzazione a 0. Tra i produttori di FPGA la questione è alquanto dibattuta: per Xilinx tale inizializzazione è addirittura raccomandata, mentre per altri va evitata! Personalmente ritengo che la pratica migliore sia quella che consente la massima portabilità del codice, cioè l’uso di un segnale di reset per tutti i registri e variabili di controllo:

```
always @ (posedge clk, posedge reset) begin
        if (reset) begin
                // Reset value goes in here.
                count <= 4‘b0101;
                ...
        end else begin
                // Code executed when the reset is deasserted.
                ...
        end
end
```

Oltre al fatto che i tool di sintesi di alcuni produttori di FPGA non l’accettano (o peggio, la ignorano), ci sono altri motivi per cui si dovrebbe evitare l'inizializzazione nella dichiarazione:

- In una FPGA i valori di inizializzazione diventano parte del file bitstream e vengono caricati durante la “configurazione” della FPGA. Tale pratica è possibile solo per le FPGA basate su SRAM: altre tecnologie (come gli ASIC) non supportano l'inizializzazione all'accensione.

- Tale inizializzazione può essere effettuata solo immediatamente dopo il caricamento della configurazione (cioè all'accensione della FPGA). Viceversa, un ingresso di reset esplicito per ogni modulo, permette di riportare il sistema in uno stato di reset in seguito a diversi eventi: accensione, pulsante (con logica antirimbalzo), cali di tensione, errori interni, ecc.

- Mescolare l'inizializzazione nella dichiarazione con quella tramite reset esplicito può creare problemi, soprattutto in grandi progetti: c'è il rischio che un segnale di reset (generato dopo l'accensione) resetti solo una parte della logica, creando due sezioni logiche desincronizzate!

Riepilogando: è buona cosa riservare l’inizializzazione nella dichiarazione a casi molto particolari e prevedere invece sempre un segnale di reset nella logica sequenziale, implementando esplicitamente l’opportuna inizializzazione. Per evitare che nella simulazione appaiano stati e segnali indefiniti basta simulare l’attivazione del reset per qualche ciclo di clock all'inizio della simulazione.

39 https://stackoverflow.com/questions/10005411/assign-a-synthesizable-initial-value-to-a-reg-in-verilog

Reset sincrono e asincrono

Come accennato sopra, il reset realizza una componente fondamentale del circuito che permette di porre flip-flop, registri ed altri elementi di memoria in uno stato ben definito. Esistono due modalità di implementazione del reset: *sincrona* e *asincrona*. Come esempio, osserviamo il codice Verilog di un flip-flop di tipo D con reset sincrono (a sinistra) e asincrono (a destra):

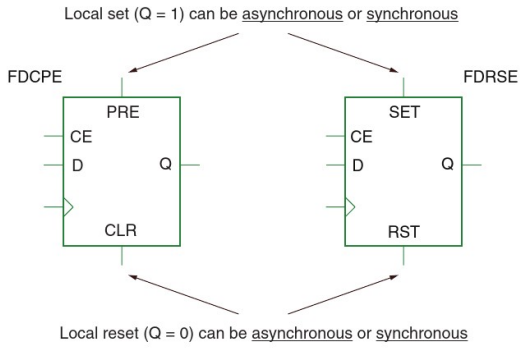
```
always @ (posedge clk) begin
        if (reset) begin
                q <= 1'b0;
        end else begin
                q <= d;
        end
end
```

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

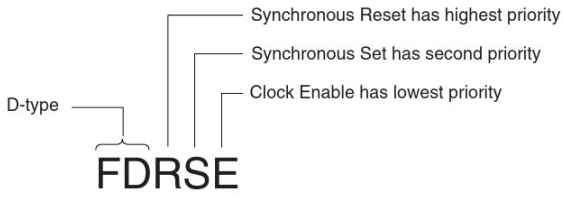
Come si vede, il codice differisce solo per gli argomenti presenti nella "*sensitivity list*" del blocco "*always*": il **reset sincrono**, per definizione, ha effetto solo sui fronti del clock, quindi l'attivazione del blocco deve poter avvenire solo in corrispondenza di tali fronti; viceversa, il **reset asincrono**, deve poter agire indipendentemente dal clock, per tale motivo esso deve essere presente nella "*sensitivity list*" del blocco. Entrambi gli esempi implementano il reset "*attivo alto*"; per ottenere quello "*attivo basso*" sostituire "*posedge reset*" con "*negedge reset*" e "*if (reset)*" con "*if (~reset)*".

Notare che, sebbene il codice Verilog delle due implementazioni sia molto simile, i rispettivi circuiti generati nella FPGA potrebbero risultare sensibilmente diversi:

Sembrerebbe quindi che il reset sincrono, in quanto segnale agente sull'ingresso D dei flip-flop, consumi logica combinatoria della FPGA, determinando anche un incremento dei tempi di propagazione. In realtà, ciò non è necessariamente vero: la configurazione hardware finale dipende infatti sia dal tipo di elementi base che la FPGA mette a disposizione, sia dall'ordine con cui i vari segnali vengono specificati nel codice Verilog. Per comprendere tale concetto occorre una premessa. Molte famiglie FPGA (inclusa l'iCE40) possono istanziare due tipi di flip-flop, che hanno gli stessi controlli, ma differiscono per il tipo di inizializzazione (asincrona o sincrona):



I flip-flop di tipo FDCPE hanno ingressi asincroni denominati CLR (c*lear*) e PRE (p*reset*) che possono agire in qualsiasi momento. Viceversa, negli FDRSE gli ingressi SET e RST (r*eset*) sono sincroni, cioè agiscono solo in corrispondenza del fronte di clock scelto (vedi dopo). Quando vengono utilizzati più controlli, ciascuno segue la priorità assegnata, indicata dalla sigla stessa:



Il circuito generato risulta ottimizzato se (e solo se) l'ordine di controllo dei segnali nel codice segue lo schema sopra indicato!

Prima di addentrarci in esempi pratici di facile comprensione occorre vedere più in dettaglio i sottotipi di flip-flop che possono essere istanziati. Come detto, essi dipendono fortemente dalla famiglia della FPGA utilizzata. Ad esempio per la iCE40 abbiamo i seguenti tipi base[40]:

SB_DFF	D flip-flop, clock attivo sul fronte positivo
SB_DFFE	D flip-flop, clock attivo sul fronte positivo, ingresso Clock Enable
SB_DFFSR	D flip-flop, clock attivo sul fronte positivo, ingresso Reset Sincrono
SB_DFFR	D flip-flop, clock attivo sul fronte positivo, ingresso Reset Asincrono (clear)
SB_DFFSS	D flip-flop, clock attivo sul fronte positivo, ingresso Set Sincrono
SB_DFFS	D flip-flop, clock attivo sul fronte positivo, ingresso Set Asincrono (preset)
SB_DFFESR	D flip-flop, clock attivo sul fronte positivo, ingressi Clock Enable & Reset Sincrono
SB_DFFER	D flip-flop, clock attivo sul fronte positivo, ingressi Clock Enable & Reset Asincrono
SB_DFFESS	D flip-flop, clock attivo sul fronte positivo, ingressi Clock Enable & Set Sincrono
SB_DFFES	D flip-flop, clock attivo sul fronte positivo, ingressi Clock Enable & Set Asincrono
SB_DFFN	D flip-flop, clock attivo sul fronte negativo
SB_DFFNE	D flip-flop, clock attivo sul fronte negativo, ingresso Clock Enable
SB_DFFNSR	D flip-flop, clock attivo sul fronte negativo, ingresso Reset Sincrono
SB_DFFNR	D flip-flop, clock attivo sul fronte negativo, ingresso Reset Asincrono (clear)
SB_DFFNSS	D flip-flop, clock attivo sul fronte negativo, ingresso Set Sincrono
SB_DFFNS	D flip-flop, clock attivo sul fronte negativo, ingresso Set Asincrono (preset)
SB_DFFNESR	D flip-flop, clock attivo sul fronte negativo, ingressi Clock Enable & Reset Sincrono
SB_DFFNER	D flip-flop, clock attivo sul fronte negativo, ingressi Clock Enable & Reset Asincrono
SB_DFFNESS	D flip-flop, clock attivo sul fronte negativo, ingressi Clock Enable & Set Sincrono
SB_DFFNES	D flip-flop, clock attivo sul fronte negativo, ingressi Clock Enable & Set Asincrono

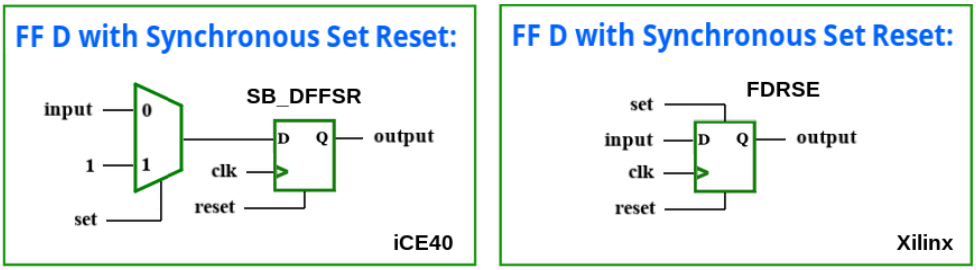
Dunque i due esempi di reset sopra visti, nel caso della iCE40, generano i seguenti circuiti:

Pertanto in questo caso, per un reset di tipo sincrono, non viene consumata logica FPGA aggiuntiva (né introdotti ritardi di propagazione). Vediamo ora un altro esempio:

40 http://www.latticesemi.com/view_document?document_id=52046

```
always @ (posedge clk) begin
        if (reset)
                q <= 1'b0;
        else if (set)
                q <= 1'b1;
        else
                q <= d;
end
```

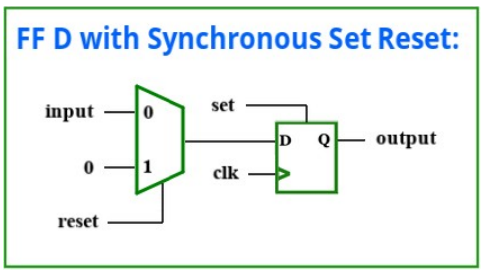
Osservare che nell'elenco sopra riportato non esistono flip-flop con entrambi gli ingressi Set/Reset (o Clear/Preset). La iCE40 non ha infatti questo tipo di elementi. Le FPGA Xilinx invece possono istanziare tale tipologia di flip-flop, per cui il circuito generato sarà diverso nelle due famiglie:



Cosa succede se nel codice precedente invertiamo per distrazione l'ordine dei segnali set e reset?

```
always @ (posedge clk) begin
        if (set)
                q <= 1'b1;
        else if (reset)
                q <= 1'b0;
        else
                q <= d;
end
```

In tal caso la priorità specificata differisce da quella naturale (indicata dalla sigla FDRSE), ed il programma di sintesi non ha modo di sapere che ciò è frutto di una disattenzione! Egli assumerà che tale comportamento è quanto desiderato dal progettista e utilizzerà della logica aggiuntiva per soddisfare tale requisito. In pratica verrà generato il circuito seguente, anche per la FPGA Xilinx:



Considerazioni analoghe possono essere fatte per l'uso del Clock Enable in relazione al set/reset. Inoltre, tali considerazioni si estendono (con dovute modifiche) al set/reset asincroni (preset/clear). Notare che non si dovrebbe mescolare reset asincrono con set sincrono (o viceversa) nello stesso blocco, in quanto nessuna FPGA in commercio è in grado di istanziare elementi che li supportano entrambi allo stesso tempo (in tal caso l'uso indesiderato di livelli logici aggiuntivi è una certezza!).

Occorre quindi prestare attenzione al codice che scriviamo, in modo da evitare l'introduzione di logiche inutili (che aumenterebbero i tempi di propagazione di alcuni segnali, causando possibili violazioni delle tempistiche di progetto). A tale scopo occorre studiare gli elementi di base della FPGA, ed eseguire test di sintesi osservando come vengono utilizzate le risorse dell'FPGA.

Tornando al tipo di reset, è lecito domandarsi quale è il migliore, se quello sincrono o asincrono. Una risposta semplice non esiste: ambedue hanno vantaggi e svantaggi, ma anche accaniti fautori e detrattori! Conviene esaminare entrambe le soluzioni e decidere caso per caso quale adottare:

- Tutte le FPGA hanno elementi di tipo FDCPE (classici flip-flop con Preset e Clear) ma non necessariamente flip-flop con set/reset sincroni. In assenza di elementi FDRSE l'uso del **reset sincrono** consuma logica combinatoria della FPGA in quanto esso agisce sull'ingresso D dei flip-flop. Per ridurre l'uso di tale logica a volte è possibile limitare il reset solo ad alcuni registri a patto che tale reset venga mantenuto attivo per un certo numero di cicli di clock. Ad esempio, in uno "*shift register*" possiamo far agire il reset sincrono solo sul primo flip-flop ed utilizzare alcuni cicli di clock per propagare lo stato di reset ai flip-flop successivi. In un progetto complesso, il reset sincrono potrebbe necessitare di centinaia o migliaia di cicli di clock (ed è facile perdere traccia di quale sia il modulo più critico). Inoltre, occorre prestare attenzione a non generare combinazioni logiche inattese, ad esempio l'uso involontario dell'enable nei flip-flop[41]. Per riassumere il reset sincrono può risultare più lento di quello asincrono e consumare logica aggiuntiva, richiede che il clock sia costantemente attivo ed è più subdolo da gestire a livello di codice HDL.

- Il **reset asincrono,** come detto, è disponibile nativamente: opera sull'ingresso di reset (o clear) dei flip-flop FDCPE e non consuma logica combinatoria (a meno di non modificare la priorità dei segnali, come descritto in precedenza). Esso inoltre non richiede che il clock sia fin dall'inizio attivo, agendo immediatamente in tutti i moduli che lo implementano.

  Purtroppo, tale tipologia di reset presenta una criticità che, se non trattata, è fonte di enormi problemi: il "*fronte del de-assert*". Se il reset asincrono viene rilasciato (cioè riportato allo stato non attivo) vicino a una transizione del clock, l'uscita di alcuni flip-flop potrebbe finire in uno stato *metastabile* (vedere il più avanti) provocando la fuoriuscita dal reset su fronti di clock diversi per le diverse parti del circuito. Ciò vanificherebbe l'operazione di reset, rendendo il circuito malfunzionante in modo totalmente aleatorio (e il debugging difficile).

  Come esempio consideriamo una macchina a stati: se i flip-flop che contengono i bit di stato entrassero in uno stato metastabile, la macchina a stati acquisirebbe molto probabilmente uno stato iniziale invalido. Inconvenienti del genere possono accadere più frequentemente di quanto si pensi nei circuiti reali, è pertanto imperativo sincronizzare il fronte del "*de-assert*" del segnale di reset con il clock mediante un opportuno "*sincronizzatore di reset*"[42]. Altra accortezza necessaria per il reset asincrono è quella di filtrare *glitch spuri*, che causerebbero reset indesiderati: si può agire sul circuito fisico rendendolo sufficientemente immune ai disturbi o implementare una tecnica di filtraggio digitale che verifichi che il reset sia attivo per un certo numero di cicli di clock prima di propagarlo a tutti i moduli.

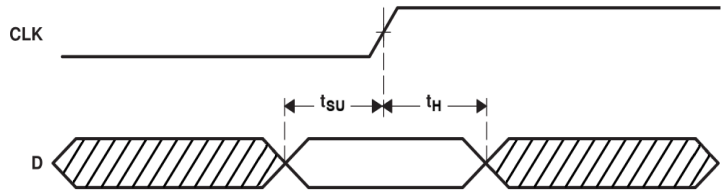
La scelta di utilizzare un reset sincrono o asincrono dipende dal progetto, ma anche da preferenze individuali. Personalmente, preferisco il reset asincrono "attivo alto" nella definizione dei moduli (in quanto più semplice da gestire), e la creazione di un "*master reset*" pilotato attraverso un sincronizzatore (per eliminare il problema del de-assert) ed eventualmente un filtro di glitch.

41 http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf
42 https://www.youtube.com/watch?v=Wdzo_DpKKGA

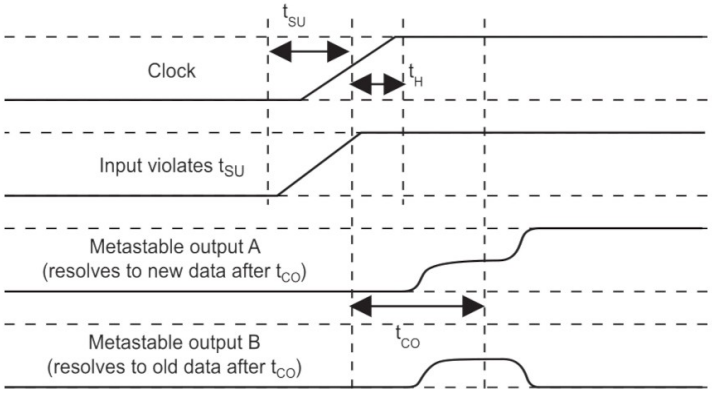
In ogni caso è bene essere coerenti, cioè non mescolare le due tipologie di reset all'interno dello stesso sistema (soprattutto all'interno dello stesso modulo o blocco) a meno che ciò non sia voluto.

Cross Domain Clock (CDC)

Come visto in precedenza, per evitare che un segnale di reset asincrono possa forzare i flip-flop in uno stato *metastabile*, occorre far transitare il reset attraverso un circuito sincronizzatore. In realtà tale necessità si applica a qualsiasi segnale proveniente dall'esterno, e all'interfacciamento tra segnali appartenenti a domini di clock diversi (cioè controllati da due clock non legati da una relazione di fase costante). Infatti, in tali situazioni, non è noto quando nel segnale in ingresso si verificherà un fronte di salita o di discesa e se tale fronte si manifesta mentre il segnale è acquisito da un dispositivo di memoria quest'ultimo potrebbe entrare in uno stato *metastabile*. Più precisamente, per questioni fisiche legate al funzionamento del flip-flop, affinché un segnale logico sia correttamente memorizzato, esso deve restare stabile almeno per un certo tempo t_{SU} (Set-Up Time) prima dell'arrivo del fronte del clock, ed essere mantenuto ugualmente stabile almeno per un certo tempo t_H (Hold Time) dopo il fronte:



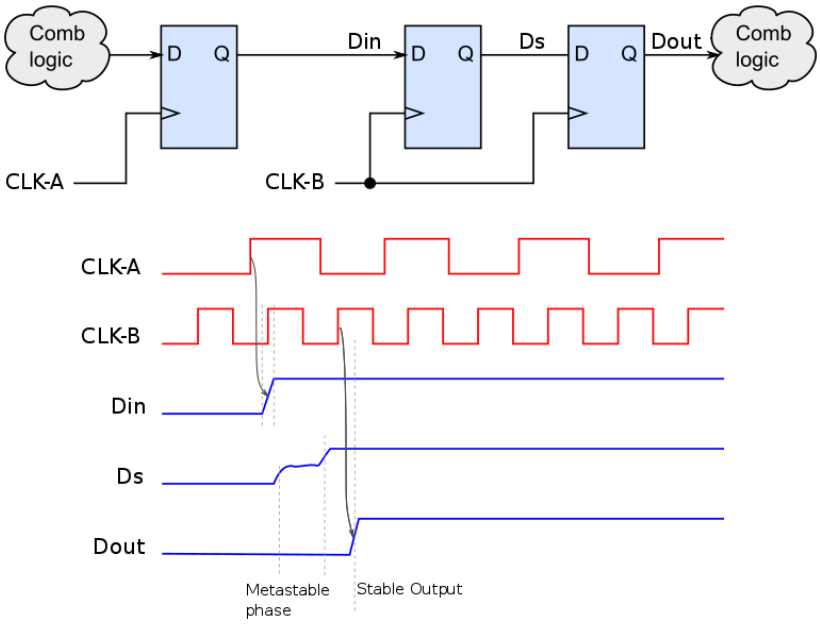
Se tale condizione viene violata il segnale acquisito assumerà dei valori intermedi (metastabilità), eventualmente oscillando, ed impiegherà del tempo ad assestarsi verso l'alto o verso il basso:



Nella figura, è possibile notare, come il tempo richiesto per l'assestamento possa essere anche superiore a t_{CO} (Clock to Output delay: ritardo dal clock affinché l'uscita si stabilizzi in condizioni di funzionamento normale), determinando una variazione sull'uscita oltre il limite di tempo consentito e creando quindi potenziali difficoltà alla logica a valle.

Per evitare tale inconveniente vi sono varie tecniche che si possono adottare, in funzione del fatto che si stia considerando un singolo segnale, oppure segnali multipli appartenenti ad un bus, e che essi siano lenti o veloci rispetto al clock del dominio di destinazione.

Il caso più semplice è quello di un singolo segnale lento (rispetto al clock del dominio di destinazione) come nel caso di un livello logico proveniente da un pulsante. In tal caso si utilizza un circuito composto da due flip-flop in cascata, chiamato "*two flip-flop synchronizer*" (tale circuito è leggermente diverso dal sincronizzatore per il reset asincrono, vedere il video linkato sopra):



Supponiamo che “Din” sia un segnale proveniente da un dominio controllato dal clock “CLK-A”. Esso per il secondo dominio, controllato dal clock “CLK-B”, è un segnale asincrono. È probabile che durante il campionamento di “Din”, l'uscita “Ds” possa entrare in uno stato metastabile. Ma è anche molto probabile che, in un tempo inferiore al periodo del clock “CLK-B” l’uscita “Ds” converga ad un valore logico stabile e possa quindi essere correttamente campionata dal secondo flip-flop (sul successivo fronte del clock), per essere riportato sull’uscita “Dout”.

Dunque, la condizione necessaria affinché questo circuito funzioni, è che la condizione di metastabilità si risolva entro un tempo leggermente inferiore al periodo di “CLK-B” (poiché occorre tenere conto di t_{SU}). Se ciò non avviene è possibile aggiungere un ulteriore stadio. Comunque, anche con solo due flip-flop la probabilità che anche l’uscita “Dout” sia metastabile è molto vicina a zero.

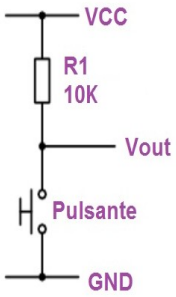
L’implementazione Verilog del circuito è data da:

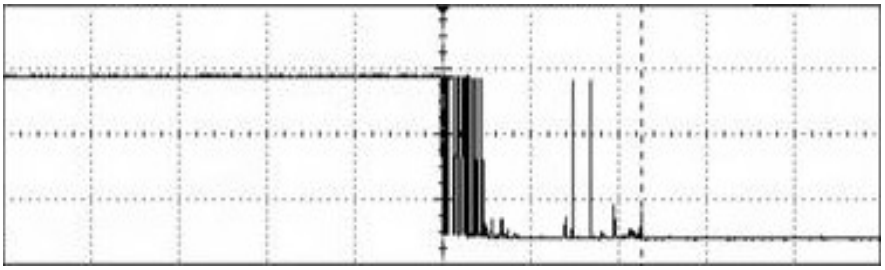
```
reg Dout, Ds;
always @ (posedge clk_b, posedge reset) begin
        if (reset)
                {Dout, Ds} <= 2'b00;
        else
                {Dout, Ds} <= {Ds, Din};
end
```

Per quanto riguarda le altre situazioni (bus, o segnali veloci), vi sono diverse tecniche, tutte piuttosto complesse per essere trattate in questa sede. Si rimanda ai numerosi articoli ed esempi disponibili su Internet.

Debouncing

La chiusura o apertura di un contatto meccanico non è mai istantanea e decisa. Per quanto possa essere veloce, è sempre accompagnata da fenomeni di rimbalzo, brevi istanti in cui lo stato del contatto oscilla tra aperto e chiuso. Questa situazione è comune a pulsanti, interruttori, deviatori, tastiere e contatti di relè. La figura seguente riporta un esempio di rimbalzo in un pulsante:





La larghezza degli impulsi, il numero di commutazioni e l'intervallo di tempo necessario affinché il segnale si stabilizzi (che può andare dai pochi microsecondi alle decine di millisecondi), dipendono dal tipo di dispositivo meccanico. È ovvio che un tale segnale non solo è asincrono, ma deve essere anche filtrato affinché il sistema digitale registri una sola commutazione e non un treno di impulsi.

Esistono diversi metodi di filtraggio: impiego di reti RC, integrati appositi (come il MAX6816 che offre anche protezione ESD) e nei microcontrollori l'uso di routine software che verifichino lo stato del contatto dopo un certo tempo dalla prima commutazione. Qui stiamo parlando di FPGA, ci interessa quindi analizzare come risolvere il problema mediante la sintesi HDL.

Per prima cosa il segnale in ingresso alla FPGA proveniente dal pulsante (connesso ad un resistore di pull-up), essendo di natura asincrona, deve passare attraverso un sincronizzatore. Chiamiamo tale segnale di ingresso "*button_in*". Notare che esso è normalmente alto, per cui i flip-flop del sincronizzatore devono essere inizializzati a 1:

```
wire button_in;
reg button_sync, meta;

always @ (posedge clk, posedge reset) begin
        if (reset)
                {button_sync, meta} <= 2'b11;
        else
                {button_sync, meta} <= {meta, button_in};
end
```

Il segnale in uscita dal sincronizzatore (qui chiamato "*button_sync*") deve essere ora filtrato per eliminare i rimbalzi. Ci sono diverse soluzioni per tale compito, ma il modo più semplice è probabilmente quello che vede impiegati un semplice flip-flop di tipo D e un contatore binario. Concettualmente, il flip-flop viene utilizzato per rilevare un rimbalzo nel segnale, mentre il contatore viene utilizzato per tenere traccia del tempo trascorso dall'ultimo rimbalzo: dopo un certo tempo trascorso senza rimbalzi si può asserire che il segnale si sia stabilizzato. In pratica:

- ad ogni fronte positivo del clock il segnale "button_sync" viene memorizzato nel flip-flop;
- sullo stesso fronte positivo di clock, il valore di "button_sync" immagazzinato nel ciclo precedente (qui chiamato "button_sync_old"), viene confrontato con il valore corrente di "button_sync":
    - se i valori di "button_sync" e "button_sync_old" sono diversi, si può asserire con certezza che si è verificato un rimbalzo: in tal caso il contatore deve essere resettato (per ricominciare da zero nel tenere traccia del trascorrere del tempo);

- se i valori di “button_sync” e “button_sync_old” sono uguali, si può assumere che non si sono verificati rimbalzi (essi sono di natura meccanica, quindi molto lenti: è assai improbabile che si verifichi un doppio rimbalzo durante un ciclo di clock); in tal caso il contatore viene incrementato per tener conto del trascorrere del tempo;

- quando il contatore raggiunge il valore massimo prestabilito, vuol dire che non si sono verificati rimbalzi per l’intervallo di tempo considerato: in tal caso si può ritenere il segnale stabile e considerare il valore “button_sync_old” immagazzinato come segnale filtrato (“button_debounced”); inoltre l’incremento del contatore in questo caso non è necessario.

Per un classico pulsante azionato manualmente possiamo considerare stabile la sua uscita dopo 15ms dall’ultimo rimbalzo. Sulla TinyFPGA BX abbiamo un clock a 16MHz e per attendere 15ms occorre contare 240.000 impulsi: sono pertanto necessari 18 bit per il contatore (2^18 => 262.144).

Abbiamo quindi il seguente codice Verilog per la sezione antirimbalzo:

```
reg button_sync_old, button_debounced;
reg [17:0] counter;

// FF.
always @ (posedge clk, posedge reset) begin
	if (reset)
		button_sync_old <= 1'b1;
	else
		button_sync_old <= button_sync;
end

// Counter.
always @ (posedge clk, posedge reset) begin
	if (reset) begin
		button_debounced <= 1'b1;
		counter <= 18'b0;
	end else if (button_sync_old != button_sync)
		counter <= 18'b0;
	else if (counter == 240000)
		button_debounced <= button_sync_old;
	else
		counter <= counter + 1'b1;
end
```

Notare che poiché il segnale è attivo basso, sia “button_sync_old”, sia “button_debounced” devono essere inizializzati a 1.

Per quanto riguarda il resistore di pull-up abbiamo due possibilità: connettere fisicamente un resistore tra il pulsante e la tensione di +3.3V in uscita dal relativo pin della TinyFPGA BX (attenzione a non connetterlo ai +5V), oppure utilizzare il resistore di pull-up interno che la iCE40 fornisce sui pin di I/O dei banchi 0, 1 e 2 (ricordare che il banco 3 non ha resistori di pull-up).

La seconda opzione ci permette di semplificare al massimo il circuito ed elimina il rischio di utilizzare per errore i +5V (che danneggerebbero la FPGA). Come visto in precedenza, per abilitare un resistore di pull-up su un pin di I/O si possono utilizzare due diverse strategie:

- modificare il file .pcf (aggiungendo "*-pullup yes*" a "*set_io*");
- usare la direttiva "*SB_IO*".

La prima è banale. Per la seconda, supponendo che il pulsante venga connesso al pin serigrafato "PIN_1" della TinyFPGA BX (che dispone di pull-up, in quanto connesso al pin A2 della FPGA che appartiene al banco 0), basta aggiungere al modulo Verilog principale il seguente codice:

```
`ifndef VERILATOR
        SB_IO #(
                .PIN_TYPE( 6'b0000_01 ),        // Configure pin as simple input.
                .PULLUP( 1'b1 )                 // Enable pull-up resistor.
        ) enable_button_in_pullup (
                .PACKAGE_PIN( PIN_1 ),          // TinyFPGA BX pin name.
                .D_IN_0( button_in )            // Input signal name.
        );
`else
        assign button_in = PIN_1;               // Bypass SB_IO on Verilator.
`endif
```

Notare che Verilator (che viene eseguito da "apio lint") deve essere bypassato poiché riporta errori nella direttiva SB_IO, pur essendo questa corretta (riscontrato con le versioni 3.992 e 4.102).

Notare inoltre che, nella FPGA iCE40 della famiglia LP, il valore del resistore di pull-up è fisso a 100Kohm. Per alcune altre famiglie, ad esempio la iCE40 UL, è invece possibile modificare tale valore a 10Kohm utilizzando l'attributo Verilog "(* PULLUP_RESISTOR = "10K" *)".

Suggerimenti utili

Analogamente allo sviluppo software, la sintesi hardware è profondamente influenzata dalla qualità del codice. Ad esempio, il numero di elementi logici istanziati e le modalità con cui essi sono connessi, determinano i tempi di propagazione dei segnali attraverso le componenti del design e quindi le performance del circuito complessivo. Per scrivere del buon codice Verilog (o VHDL) sono necessari una buona preparazione nel campo delle reti logiche e anni di esperienza che non possiamo riassumere in queste poche pagine. Comunque ecco alcuni suggerimenti utili:

- Studiare la FPGA a disposizione ed effettuare dei semplici test per verificare che il datasheet sia corretto (ciò permette anche di prendere confidenza con la FPGA prima di affrontare un progetto). Se possibile, provare più tool e verificare quale sintetizza la logica migliore.

- Costruire un progetto un piccolo pezzo per volta. Nello sviluppo del codice, non cambiare troppe cose contemporaneamente e fare simulazioni continue (e test sulla FPGA reale). Se si introducono troppe modifiche insieme, e quanto realizzato finora smette di funzionare, diventa difficile determinare quale modifica è responsabile del fallimento.

- Preferire sempre la logica sincrona a quella asincrona: quest'ultima è meno affidabile e più difficile da progettare correttamente. Per il reset vale l'opposto.

- Mantenere lo stesso stile di reset per tutto il design (cioè non mescolare reset sincrono con reset asincrono).

- Usare possibilmente un unico segnale di clock in tutto il progetto: l'uso di divisori o altra logica per generare altri domini di clock è una pessima pratica che causa la commutazione non omogenea della logica sequenziale, jitter, problemi di fan-out (in quanto le linee global buffer sono limitate) e tempi di propagazione elevati[43]. La metodologia corretta è quella di utilizzare un unico clock per tutti i flip-flop e registri della FPGA, e sfruttare gli ingressi "clock enable" (che tali componenti possiedono) per ottenere quanto desiderato, racchiudendo il codice HDL in istruzioni "if" condizionate da logica opportuna[44].

- Quando si ha della logica combinatoria complessa valutare sempre il numero di livelli di lookup table (LUT) richiesti: se il tempo di propagazione attraverso tale logica diventa comparabile con il periodo del clock (o peggio, se risulta maggiore) spezzare il percorso logico in due o più parti, interponendo uno o più registri in mezzo (pipelining). Ripetere l'analisi dei tempi di propagazione man mano che si costruisce il circuito per non scoprire troppo tardi eventuali problemi di timing.

- Se possibile, utilizzare sempre uno stesso fronte del clock in tutto il design (ad esempio sempre quello positivo). L'impiego di entrambi i fronti potrebbe dimezzare il tempo massimo accettabile per la propagazione dei segnali.

- Preferire i flip-flop D agli altri tipi (che comportano l'uso di logica aggiuntiva).

- Sincronizzare sempre i segnali asincroni al clock mediante gli opportuni sincronizzatori.

- Se si porta il design su architetture FPGA diverse, considerare le caratteristiche di ciascuna famiglia per verificare che soddisfino i vincoli temporali del progetto.

- Utilizzare dei nomi significativi nel codice ed essere coerenti nello stile impiegato.

- Non utilizzare ciecamente le componenti di libreria a disposizione, comprendere come si integrano nel progetto e se devono essere o meno modificate.

- Sfruttare componenti evolute che la particolare FPGA mette a disposizione (moltiplicatori, unità DSP, ecc) ma mantenere anche una corrispondente versione generica: questa tornerà utile per la simulazione ed eventualmente per portare il design su quelle FPGA che non dispongono di tali componenti.

Conclusioni

Questo documento termina qui, il suo scopo è solo quello di facilitare chi si avvicina per la prima volta all'affascinante mondo delle FPGA. Per cui si rimanda l'approfondimento agli appositi testi e agli innumerevoli esempi presenti su Internet.

Buon divertimento!

43 https://electronics.stackexchange.com/questions/222972/advantage-of-clock-enable-over-clock-division
44 https://www.fpga4student.com/2017/08/how-to-generate-clock-enable-signal.html

7 Riferimenti

Guida di riferimento
https://tinyfpga.com/bx/guide.html

Schematici/PCB KiCad, bootloader, esempi
https://github.com/tinyfpga/TinyFPGA-BX
https://github.com/mattvenn/TinyFPGA-BX
NOTA: i due repository differiscono solo per alcuni file nella cartella examples/picosoc.

Forum:
https://discourse.tinyfpga.com/

FPGA iCE40LP8K-CM81:
http://www.latticesemi.com/Products/FPGAandCPLD/iCE40.aspx

Librerie logiche:
http://www.latticesemi.com/solutionsearch?qprod=9e00ea3e2f6e43b2bb3f9430fc476b47&qiptype=3614c818569f4eecb0602ba20a521a41,6da9534f318a4969a6b5e7dc9081bdba

Guida di Apio
https://apiodoc.readthedocs.io/en/stable/source/user_guide/index.html

Verilog Testbench
https://www.fpgatutorial.com/how-to-write-a-basic-verilog-testbench/

Altri tool di sviluppo:
https://f4pga.org/
https://github.com/FPGAwars/icestudio
https://github.com/rochus-keller/QtcVerilog
https://github.com/YosysHQ/oss-cad-suite-build

RISC-V PicoRV32
https://github.com/YosysHQ/picorv32
https://libraries.io/github/kionix/picorv32

Indice

1 Aggiornamento del bootloader....3
2 TinyFPGA BX Quick Start....5
3 Architettura della FPGA iCE40....7
Struttura delle celle....8
Global Buffer....9
Programmazione....10
Conclusioni....11
4 Scheda TinyFPGA BX in dettaglio....12
Comportamento del bootloader....14
Pull-up....14
5 Simulazione e programmazione della TinyFPGA BX....15
Note sulla simulazione....16
6 Brevi note sulla sintesi hardware in Verilog....17
Modelli “Behavioural”, “RTL” e “Gate Level”....17
Blocchi “always” e “initial”....18
Tipi “reg” e “wire”....18
Assegnazioni ed eventi....19
Dichiarazione “assign”....21
Qualche regola fondamentale....21
Inizializzazione dei registri....22
Reset sincrono e asincrono....23
Cross Domain Clock (CDC)....27
Debouncing....28
Suggerimenti utili....31
Conclusioni....32
7 Riferimenti....33