

Appunti e Documentazione Verilog per Esercitazioni di Reti Logiche A.A. 2025/26

Raffaele Zippo

15 ottobre 2025

Indice

1	Esercitazioni di Reti Logiche	5
1.1	Chi tiene il corso	5
2	Introduzione	6
2.1	Perché compilare, testare, debuggare	6
2.2	Ambienti utilizzati	6
2.3	Domande e ricevimenti	6
3	Ambienti di sviluppo	7
3.1	Editor	7
3.2	Ambiente assembler	7
3.3	Ambiente Verilog	8
3.4	Versioni dell'ambiente e alternative	8
3.5	Ambiente per Windows 11 + WSL2	8
3.6	Ambiente per Linux nativo o devcontainer	9
3.6.1	Utilizzo nativo	9
3.6.2	Utilizzo tramite devcontainer	10
3.7	Alternative fai da te	10
3.8	Testare gli ambienti	10
3.8.1	Assembler	10
3.8.2	Verilog	10
4	Essere efficienti con VS Code	11
4.1	Le basi elementari	11
4.2	Le basi un po' meno elementari	11
4.3	Editing multi-caret	12
I	Verilog - Esercitazioni	13
5	Esercitazione 1	14
5.1	Da schemi circuitali a codice	14
5.2	Concetto di testbench	15
5.3	Full adder, descrizione e sintesi di reti combinatorie	19
6	Esercitazione 2	22
6.1	Errori comuni: i corto circuiti	22
6.2	Uso efficiente di VS Code	24
6.3	Esercizi d'esame	24
6.4	Esercizio 2.1: parte combinatoria esame 2023-06-27	24
6.5	Esercizio 2.2: parte combinatoria esame 2023-01-31	27
6.5.1	Soluzione 1	27
6.5.2	Soluzione 2	29
7	Esercitazione 3	32
7.1	Reti sincronizzate	32
7.1.1	Testbench e generatore di clock	32
7.1.2	Primo esempio di rete sincronizzata: il contatore	33
7.1.3	Mantenere un segnale per N cicli di clock	35
7.1.4	Esercizio: Handshake e reti combinatorie	37
7.1.5	Testbench con input e output per reti sincronizzate	39

8 Esercitazione 4	41
8.1 Esercizio 4.1: Descrizione	41
8.2 Esercizio 4.1: Sintesi della rete combinatoria	43
8.3 Esercizio 4.1: Sintesi di rete sincronizzata	43
8.3.1 Passo 0: ricopiare su un nuovo file	44
8.3.2 Passo 1: rendere la descrizione omogenea	44
8.3.3 Passo 2: separazione dei blocchi operativi	45
8.3.4 Passo 3: variabili di comando	46
8.3.5 Passo 4: variabili di condizionamento	47
8.3.6 Passo 5: separare le parti	48
8.3.7 Passo 6: la ROM	50
9 Esercitazione 5	52
9.1 Esercizio 5.1: esame 2023-07-18	52
9.2 Esercizio 5.1: esame 2024-01-26	52
10 Esercitazione 6	54
10.1 Esercizio 6.1: esame 2024-07-16	54
10.2 Esercizio 6.2: esame 2024-09-10	56
 II Verilog - Documentazione	 57
11 Introduzione	58
12 Operatori	59
12.1 Valori letterali (<i>literal values</i>)	59
12.1.1 Estensione e troncamento	59
12.2 Operatori aritmetici	59
12.3 Operatori logici e <i>bitwise</i>	59
12.3.1 <i>Reduction operators</i>	60
12.4 Operatore di selezione [...]	60
12.5 Operatore di concatenazione {...}	61
12.5.1 Operatore di replicazione N{...}	61
12.6 Operazioni comuni	61
12.6.1 Estensione di segno	61
12.6.2 Shift a destra e sinistra	61
13 Sintassi per reti combinatorie	62
13.1 module	62
13.1.1 input e output	62
13.2 wire	62
13.3 Usare un module in un altro module	63
13.4 Tabelle di verità	63
13.5 Multiplexer	64
13.6 Reti parametrizzate	64
14 Sintassi per reti sincronizzate	66
14.1 Istanziamento	66
14.2 Collegamento a wire	66
14.3 Struttura generale di un blocco always	67
14.4 Comportamento al reset	67
14.5 Aggiornamento al fronte positivo del clock	67
14.6 Limitazioni della simulazione: temporizzazione, non-trasparenza e operatori di assegnamento	68
15 Simulazione ed uso di GTKWave	69
15.1 Compilazione e simulazione	69
15.1.1 Testbench con `timescale	70
15.2 Waveform e debugging	70
15.2.1 Zoom, ordinamento, formattazione	71
15.2.2 Non specificati e alta impedenza	71
15.2.3 Pulsante <i>Reload</i>	71
15.2.4 Linea di errore	71

III Verilog - Appendice	73
16 Simulatore processore sEP8	74
16.1 Lancio di simulazioni	74
16.2 Caricamento di programmi tramite ROM	75
16.2.1 Riferimenti storici: le cartucce	75
*	

Capitolo 1

Esercitazioni di Reti Logiche

Questa dispensa contiene appunti e materiali per le esercitazioni del corso di Reti Logiche, Laurea Triennale di Ingegneria Informatica dell'Università di Pisa, A.A. 2025/26.

Il contenuto presume conoscenza degli aspetti teorici già discussi nel corso, ricordando alla bisogna solo gli aspetti direttamente collegati con gli esercizi trattati.

Materiale in costruzione

Questa dispensa contiene materiale in via di stesura, è messa a disposizione per essere utile quanto prima. Potrebbero esserci inesattezze o errori. Siete pregati, nel caso, di [segnalarlo](#).

Il contenuto di partenza è quello dell'anno precedente, e verrà sostituito man mano in base a quello che viene svolto a lezione. Aspettativi quindi che cambi *molto* durante il trimestre del corso.

1.1 Chi tiene il corso

Il corso è tenuto dal [Prof. Giovanni Stea](#) . Le esercitazioni sono tenute dal [Dott. Raffaele Zippo](#) .

La pagina ufficiale del corso è http://docenti.ing.unipi.it/~a080368/Teaching/RetiLogiche/index_RL.html.

Capitolo 2

Introduzione

2.1 Perché compilare, testare, debuggare

If debugging is the process of removing bugs, then programming must be the process of putting them in.
Edsger W. Dijkstra

Si parta dal presupposto che fare errori *succede*. Meno è banale il progetto o esercizio, più è facile che da qualche parte si sbagli. La parte importante è riuscire a cogliere e rimuovere questi errori prima che sia troppo tardi, sia che si tratti di rilasciare un software in produzione o di consegnare l'esercizio a un esame. In queste esercitazioni vedremo questo processo in contesti specifici (software scritto in assembler e reti logiche descritte in Verilog) ma la linea si applica in generale in tutti gli altri ambiti dell'ingegneria informatica. Dunque il codice, di qualunque tipo sia, non va solo scritto, va *provato*. Come identificare, trovare e rimuovere gli errori è invece una capacità pratica che va *esercitata*.

2.2 Ambienti utilizzati

Gli strumenti a disposizione per provare e testare il codice, così come la loro praticità d'uso possono cambiare molto in base ad architettura, sistema operativo, e generale potenza delle macchine utilizzate.

Dato che il corso è collegato a un esame, ci si concentrerà sullo stesso ambiente che sarà disponibile all'esame, che è dunque basato su PC desktop con Windows 11 e architettura x86. Il software e le istruzioni a disposizione riguarderanno questa combinazione.

Per altre architetture e sistemi operativi, il supporto è sporadico e *best effort*, con nessuna garanzia da parte dei docenti che funzioni. Dovrete, con molta probabilità, litigare con il vostro computer per far funzionare il tutto.

Una introduzione generale alle opzioni è in [Ambienti software](#).

2.3 Domande e ricevimenti

Siamo a disposizione per rispondere a domande, spiegare esercizi, colmare lacune. Gli orari ufficiali di ricevimento sono comunicati durante il corso e tenuti aggiornati sulle pagine personali. È sempre una buona idea scrivere prima, via email o Teams, per evitare impegni concomitanti o risolvere più rapidamente in via testuale. In caso di dubbi su esercizi, aiuta molto allegare il testo dell'esercizio (foto o pdf) e il codice sorgente (sempre e solo file testuale, non foto o file binari).

Non è raro che gli studenti si sentano in imbarazzo o comunque evitino di fare domande, quindi ci spendo qualche parola in più. Fuori dall'esame, è nostro *compito* insegnare, e questo include rispondere alle domande. È un *diritto* degli studenti chiedere ricevimenti e avere risposte. Avere dubbi o lacune è in questo contesto *positivo*, perché sapere di non sapere qualcosa è un primo passo per imparare.

Capitolo 3

Ambienti di sviluppo

In questo corso, scriveremo codice per programmi assembler e per descrivere reti logiche in Verilog. Per entrambi, utilizziamo un ambiente software che è lo stesso (o estremamente simile) a quello che si troverà all'esame.

3.1 Editor

Nelle esercitazioni e nella documentazione faremo riferimento a [VS Code](#), che è l'unico editor che si potrà utilizzare all'esame.

Non c'è però nessun obbligo a usare VS Code per le esercitazioni personali, qualunque editor di file di testo andrà bene. Anche un editor da terminale come `nano` o `vim`.

3.2 Ambiente assembler

Programmare in assembler vuol dire programmare per una specifica architettura di processori. L'architettura x86 è stata rimpiazzata nel tempo da x64, a 64 bit, che è del tutto retrocompatibile. Altre architetture (in particolare, ARM) hanno istruzioni, registri e funzionamento completamente diversi e non sono compatibili con x86. Usare una macchina con architettura diversa è inevitabilmente fonte di problemi.

L'ambiente fornito funziona con Linux x86 (o x64 o amd64, che significano la stessa cosa). Non funziona invece per processori arm64, come quelli usati da Mac o Windows on ARM.

Da una parte, si potrebbe pensare di esercitarsi scrivendo assembler per la propria architettura, anziché quella usata nel corso. Sorgono diversi problemi:

- dover imparare sintassi, meccanismi, registri completamente diversi;
- dover fare a meno o reingegnerizzarsi la libreria usata per l'input-output a terminale;
- dover comunque imparare l'assembler mostrato nel corso, perché quella sarà richiesta all'esame e supportata dalle macchine in laboratorio.

La seconda opzione è usare strumenti di virtualizzazione capaci di far girare un sistema operativo con architettura diversa. Sorge come principale problema l'ergonomicità ed efficienza di questa soluzione, che dipende molto dagli strumenti che si trovano e dalle caratteristiche hardware della macchina, che potrebbero essere non sufficienti.

Per chi ha una macchina ARM, sarà necessario trovare soluzioni di virtualizzazione o usare un'altra macchina dedicata (va bene qualunque cosa di qualunque potenza, purché x86). In ogni caso, *non offriamo nessun supporto diretto* a tali macchine. Lo ribadisco in rosso, perché chiesto spesso.

Nessun supporto diretto per Mac con ARM

Non testiamo né supportiamo ambienti per Mac con ARM, che non abbiamo a disposizione. *Ci è stato detto* che [UTM](#) può emulare l'architettura x86, affermazione che riportiamo senza alcuna garanzia. Non risponderemo a ulteriori domande a riguardo, soprattutto se parte delle [domande frequenti](#).

Oltre a questioni di architettura, abbiamo anche il sistema operativo, che è rilevante per gestire input e output da terminale. I programmi che scriveremo ed eseguiremo, così come quelli utilizzati per assemblare, gireranno in un terminale Linux. Nei pacchetti forniti e in sede di esame, si usa in particolare Ubuntu 24.04.

Perché Linux?

Perché è molto più facile virtualizzare un ambiente Linux moderno in Windows o Mac che il contrario. In precedenza si usava MS-DOS, un sistema del 1981 facilmente emulabile, ma molto limitante data l'età.

Per assemblare, si usa `gcc`, per debuggare `gdb`. Per usarli però sono necessari comandi *lunghi*, che semplifichiamo usando script Powershell `assemble.ps1` e `debug.ps1`.

Perché Powershell?

Perché Powershell (2006) è object-oriented, e permette di scrivere script leggibili e manutenibili, in modo semplice. Bash (1989) è invece text-oriented, con una [lunga lista di trappole da saper evitare](#).

32 vs 64 bit

In realtà, i processori x86 a soli 32 bit non sono più in commercio da vent'anni. I processori che si trovano oggi sono x64, a 64 bit, e sono in grado di eseguire codice a 32 bit per retrocompatibilità. Nel corso, continuiamo ad usare l'istruzione set a 32 bit perché

2. è di complessità ridotta e sufficiente per i nostri scopi didattici,
2. il vecchio ambiente DOS, che qualcuno può trovare ancora utile, supporta *solo* x86.

3.3 Ambiente Verilog

L'ambiente Verilog non ha i problemi di quello assembler, perché quel che compiliamo (una rete simulabile) non è legato sistema operativo o all'architettura della CPU. Basta che si riescano ad installare

- `iverilog` e `vvp`
- GTKWave

3.4 Versioni dell'ambiente e alternative

L'ambiente dell'A.A. 2025/26 è leggermente diverso da quello degli anni precedenti. Le differenze riguardano solo aspetti di installazione e configurazione, il modo di utilizzo rimane pressoché invariato.

Se si ha già un ambiente funzionante, non c'è bisogno di fare nulla.

L'ambiente è fornito in due versioni:

- Windows 11 + WSL2
- Linux nativo o devcontainer

Questi contengono sia istruzioni per installazione e configurazione, sia le cartelle `assembler` e `verilog` con i file necessari per scrivere codice.

Tenere presente che non c'è bisogno di utilizzare lo stesso tipo di pacchetto o macchina per assembler e Verilog, le due scelte sono indipendenti.

3.5 Ambiente per Windows 11 + WSL2

Download

Questo pacchetto supporta macchine Windows 11 x64, utilizza WSL2 per virtualizzare un sistema Ubuntu 24.04 per assembler, e applicazioni native per Verilog.

WSL2 è un sottosistema di Windows che permette di virtualizzare macchine Linux in modo semplice, e l'integrazione con VS Code tramite [l'estensione WSL](#) permette di scrivere codice *fuori* dalla macchina virtuale ed assemblare ed eseguire *dentro* la macchina virtuale. Questo ci permette di mantenere un ambiente grafico moderno mentre si lavora con un terminale Linux virtualizzato.

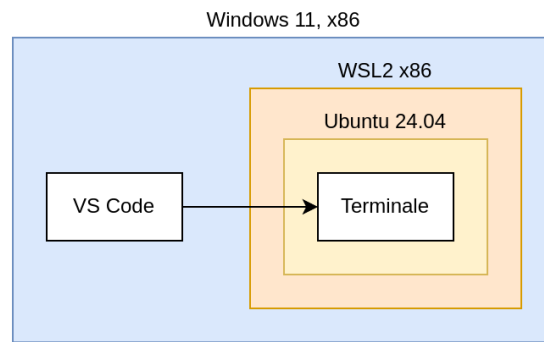


Figura 3.1: Schema dell'ambiente usato all'esame.

Il pacchetto dell'ambiente contiene le istruzioni passo passo per installare e configurare la macchina virtuale su una macchina Windows 11 con architettura x86.

Per l'ambiente Verilog, invece, ci sono sia installer precompilati [qui](#) che codice sorgente [qui](#) e [qui](#).

3.6 Ambiente per Linux nativo o devcontainer

Download

Questo pacchetto supporta due scenari: una macchina con Linux x64, oppure [devcontainers](#) tramite Docker. Il pacchetto contiene le cartelle `assembler` e `verilog` con i file necessari per scrivere codice.

3.6.1 Utilizzo nativo

Per assembler, l'ambiente Linux deve essere in grado di

- Eseguire gli script powershell dell'ambiente
- Assemblare, usando `gcc`, programmi x86 scritti con sintassi GAS
- Eseguire programmi x86
- Debuggarli usando `gdb`

Per far questo su Ubuntu 24.04, i pacchetti da installare sono

- `build-essential`
- `gcc-multilib`
- `gdb`
- powershell ([guida](#))

Per Verilog, l'ambiente Linux deve essere in grado di

- Compilare simulazioni con `iverilog`
- Eseguire simulazioni con `vvp`
- Visualizzare waveform con `gtkwave`

Per far questo su Ubuntu 24.04, i pacchetti da installare sono

- `iverilog`
- `gtkwave`

Altro software per installazioni minime

Script e istruzioni si basano anche su due altri programmi: `wget` e `file`. Di solito sono inclusi di default per installazioni Desktop, ma su installazioni minime (come l'immagine Docker di Ubuntu 24.04) vanno installati manualmente.

Una volta installato il software richiesto, per sviluppare basterà aprire le cartelle con VS Code.

3.6.2 Utilizzo tramite devcontainer

I **devcontainer** sono un'altra forma di virtualizzazione integrata in VS Code, basata su Docker anziché WSL. Il pacchetto include, nelle cartelle `.devcontainer`, i Dockerfile che installano il software necessario su immagini Ubuntu 24.04.

Una volta aperta la cartella con VS Code, usare il comando "Riapri in devcontainer".

3.7 Alternative fai da te

Un'altra opzione molto utile di VS Code è lo sviluppo remoto tramite **SSH** usando [questa estensione](#). In questo caso, invece di collegarsi a un ambiente di sviluppo virtualizzato, questo risiede su un'altra macchina a cui ci si collega aprendo un terminale SSH.

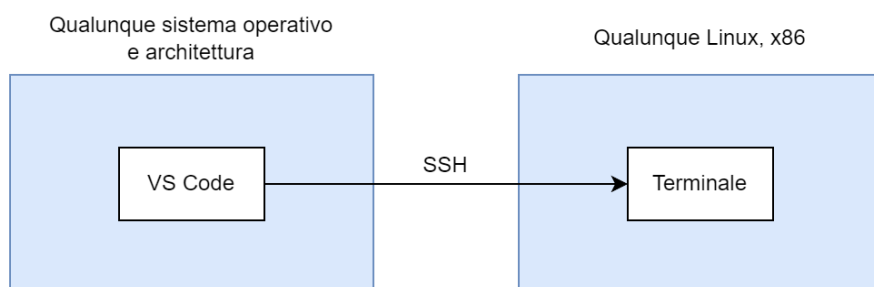


Figura 3.2: Schema di un ambiente che usa SSH.

Da notare che le macchine sono distinte "concettualmente": niente ci vieta di avere una macchina virtuale (e.g. VirtualBox) al posto di una macchina fisicamente distinta.

3.8 Testare gli ambienti

I pacchetti includono dei file per *testare* che l'ambiente sia utilizzabile.

3.8.1 Assembler

Il file `test-ambiente.s` contiene il codice di un semplice programma che si limita a stampare `Ok..`. Provare ad assemblarlo, eseguirlo e debuggarlo.

```
PS /workspaces/assembler> ./assemble.ps1 ./test-ambiente.s
PS /workspaces/assembler> ./test-ambiente
Ok.
PS /workspaces/assembler> ./debug.ps1 ./test-ambiente
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
[output di poca utilità]
Breakpoint 1, _main () at /workspaces/assembler/test-ambiente.s:7
7      _main:  nop
(gdb) qq
PS /workspaces/assembler>
```

3.8.2 Verilog

Il file `test-ambiente.v` contiene il codice di una semplice testbench con registro da 1 bit che cambia valore, e stampa `Ok.` a terminale prima di terminare. Provare a compilare ed eseguire la simulazione, e poi osservarne la waveform.

```
PS /workspaces/verilog> iverilog -o sim ./test-ambiente.v
PS /workspaces/verilog> vvp ./sim
VCD info: dumpfile waveform.vcd opened for output.
Ok.
./test-ambiente.v:10: $finish called at 20 (1s)
PS /workspaces/verilog> gtkwave ./waveform.vcd
[output di poca utilità]
PS /workspaces/verilog>
```

Capitolo 4

Essere efficienti con VS Code

VS Code è l'editor disponibile in sede d'esame e mostrato a lezione. Come ogni strumento di lavoro, è una buona idea imparare ad usarlo bene per essere più rapidi ed efficaci. Questo si traduce, in genere, nel prendere l'abitudine di usare meno il mouse e più la tastiera, usando le dovute scorciatoie e combinazioni di tasti.

In questa documentazione ci focalizziamo sulle combinazioni per Windows, che sono quelle che troverete all'esame. Evidenzierò con una ☆ le combinazioni più importanti e probabilmente meno note.

Salvare i file

Fra le cause dei vari errori per cui riceviamo richieste d'aiuto, una delle più frequenti è che i file modificati non sono stati salvati. Un file modificato ma non salvato è indicato da un pallino nero nella tab in alto, e le modifiche non saranno visibili a altri programmi come gcc e iverilog.

Si consiglia di salvare spesso e abitualmente, usando `ctrl + s`.

4.1 Le basi elementari


Quando si scrive in un editor, il testo finisce dove sta il cursore (in inglese *caret*). È la barra verticale che indica dove stiamo scrivendo. Si può spostare usando le frecce, non solo destra e sinistra ma anche su e giù. Usando font monospace, infatti, il testo è una matrice di celle delle stesse dimensioni, ed è facile prevedere dove andrà il caret anche mentre ci si sposta tra le righe.

Vediamo quindi le combinazioni più comuni.

	Tasti	Cosa fa
	Tenere premuto shift	Seleziona il testo seguendo il movimento del cursore.
	<code>ctrl + c</code>	Copia il testo selezionato.
	<code>ctrl + v</code>	Incolla il testo selezionato.
	<code>ctrl + x</code>	Taglia (cioè copia e cancella) il testo selezionato.
	<code>ctrl + f</code>	Cerca all'interno del file.
	<code>ctrl + h</code>	Cerca e sostituisce all'interno del file.
☆	<code>ctrl + s</code>	Salva il file corrente.
	<code>ctrl + shift + p</code>	Apri la <i>Command Palette</i> di VS Code.

4.2 Le basi un po' meno elementari

Si può spostare il cursore in modo ben più rapido che un carattere alla volta.

	Tasti	Cosa fa
☆	<code>ctrl + freccia sx o dx</code>	Sposta il cursore di un <i>token</i> (in genere una parola, ma dipende dal contesto).
	<code>home</code> (inizio in italiano, più spesso )	Sposta il cursore all'inizio della riga.
	<code>end</code> (fine in italiano)	Sposta il cursore alla fine della riga.
	<code>ctrl + shift + f</code>	Cerca all'interno della cartella/progetto/...
	<code>ctrl + shift + h</code>	Cerca e sostituisce all'interno della cartella/progetto/...
	<code>alt + freccia su/giù</code>	Sposta la riga corrente (o le righe selezionate) verso l'alto/basso.
☆	<code>ctrl + alt + freccia su/giù</code>	Copia la riga corrente (o le righe selezionate) verso l'alto/basso.

4.3 Editing multi-caret

Normalmente c'è *un* cursore, e ogni modifica fatta viene applicata dov'è quel *singolo* cursore.

Negli esempi che seguono, userò `|` per indicare un cursore, e coppie di `_` come delimitatori del testo selezionato.

```
Contenu|to dell'editor
```

Premendo A

```
ContenuA|to dell'editor
```

L'idea del multi-caret è di avere più di un cursore, per modificare più punti del testo allo stesso tempo. Questo è utile se abbiamo più punti del testo con uno stesso *pattern*.

	Tasti	Cosa fa
☆	ctrl + d	Aggiunge un cursore alla fine della prossima occorrenza del testo selezionato.
	esc	Ritorno alla modalità con singolo cursore.

Vediamo un esempio.

```
Prima |riga dell'editor
Seconda riga dell'editor
Terza riga dell'editor
```

Si comincia selezionando del testo.

```
Prima _riga_| dell'editor
Seconda riga dell'editor
Terza riga dell'editor
```

Usiamo ora `ctrl + d` per mettere un nuovo caret dopo la prossima occorrenza di "riga".

```
Prima _riga_| dell'editor
Seconda _riga_| dell'editor
Terza riga dell'editor
```

Abbiamo ora due caret e se facciamo una modifica verrà fatta in tutti e due i punti. Premendo per esempio `e`, andremo a sovrascrivere la parola "riga" in entrambi i punti.

```
Prima e| dell'editor
Seconda e| dell'editor
Terza riga dell'editor
```

Entrambi i cursori seguiranno indipendentemente anche gli altri comandi: movimento per caratteri, movimento per token, selezione, copia e incolla.

Per sfruttare questo, conviene scrivere codice secondo pattern in modo da facilitare questo tipo di modifiche. Per esempio, è utile avere cose che vorremmo poi modificare contemporaneamente su righe diverse, in modo da sfruttare `home` e `end` in modalità multi-cursore.

Vedremo in particolare come la sintesi di reti sincronizzate diventa molto più semplice se si sfrutta appieno l'editor.

Parte I

Verilog - Esercitazioni

Capitolo 5

Esercitazione 1

Per capire bene cos'è il Verilog è bene partire dal capire per cosa si usa. È un *Hardware Description Language*, cioè un linguaggio formalizzato per la progettazione e realizzazione di componenti hardware: da reti combinatorie a CPU, architetture avanzate e componenti dedicati a scopi specifici.

Lo scopo non è quindi *solo* descrivere dell'hardware con del codice anziché disegni, ma in generale supportare con strumenti utili l'ingegnere in tutte le fasi di progettazione di sistemi elettronici digitali, a partire dalla semplice prototipazione dell'interfaccia (dove poco importa la realizzazione interna, ma solo l'algoritmo implementato), passando per la simulazione in testbench software, alla realizzazione fisica su FPGA e test in hardware.

Tutti questi scopi hanno richieste diverse, e *semantiche* relative diverse. Per questo non dovrebbe stupire il fatto che Verilog include molte diverse funzionalità e sintassi che hanno senso solo in specifici contesti e non altri, che spazia dalle porte logiche elementari a strutture di programmazione stile-C e funzionalità di stampa a terminale.

Questo è spesso fonte di confusione, visto che il compilatore Verilog non aiuta a fare queste distinzioni, anzi, supporta intenzionalmente diversi modi di usare le stesse keyword, come `reg` che può essere utilizzata sia come variabile di un programma che come un registro in una rete sincronizzata. Come vedremo, è importante tenere presente *cosa* si sta facendo e *perché* per poter capire quale forma e sintassi ha senso usare e quale no.

Noi vedremo 3 usi diversi, in particolare:

- descrizione e sintesi di reti combinatorie
- descrizione e sintesi di reti sincronizzate
- verifica con testbench simulativa

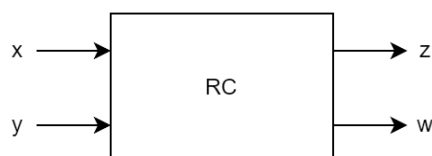
Argomenti d'esame

Saper leggere o scrivere testbench *non* è parte degli argomenti d'esame. È tuttavia estremamente utile per esercitarsi provando con mano l'hardware descritto e capire come si comporta.

Per ogni esercizio, così come in sede d'esame, viene *fornita* una testbench adatta.

5.1 Da schemi circuitali a codice

La bussola fondamentale per scrivere Verilog è tenere sempre presente l'hardware che si vuole realizzare. Partiamo dall'idea di hardware che abbiamo tramite schemi, come nell'esempio in figura.



Questo schema mostra una generica rete combinatoria RC con ingressi `x` e `y`, e uscite `z` e `w`. Questa rete logica sarà implementata poi con componenti elettronici. Sappiamo che questi, in quanto componenti fisici reali, non hanno un concetto di ordine tra di loro, o sincronizzazione, o attesa: gli ingressi `x` e `y` variano indipendentemente, possono avere cambiamenti anche contemporanei e fluttuanti, e la rete RC risponde sempre a questi cambiamenti tramite le uscite `z` e `w`, anche *durante* i transitori dove gli ingressi variano da uno stato a un altro. Questa può sembrare una ripetizione banale se si pensa ai segnali elettrici che si propagano in un circuito, ma è facile dimenticarsene quando si guarda al codice Verilog. Vediamo come questo schema si può tradurre in codice.

```

module RC(x, y, z, w);

input x, y;
output z, w;

assign #1 z = x | y;
assign #2 w = x & y;

endmodule

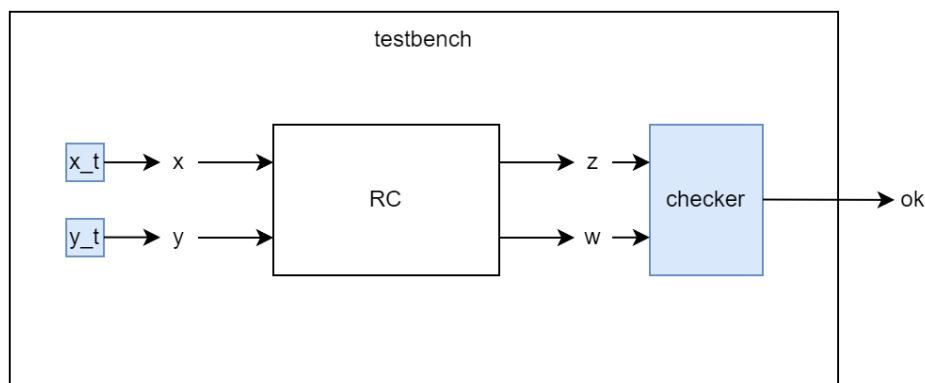
```

In Verilog si dichiarano moduli in modo simile alle classi in linguaggi di programmazione: un modulo è un *tipo* di componente che altri moduli potranno poi usare. La riga 1 inizia la dichiarazione del modulo, che è composta dal nome del modulo (`RC`) e dalla lista di porte di questo modulo, anch'esse con nome (`x, y, z, w`). Queste porte possono essere di input e/o output, a uno o a più bit. Specifichiamo questo alle righe 3 e 4. Mancando indicazioni di dimensione, saranno tutte da 1 bit. Alle righe 6 e 7 specifichiamo il comportamento dei fili di uscita `z` e `w`. Lo statement `assign` indica che l'elemento a sinistra assume continuamente il valore indicato dall'espressione a destra. Con `#1` si indica un fattore di ritardo nell'aggiornamento, di 1 unità di tempo. Ogni rete combinatoria che non sia un semplice filo ha un certo tempo di attraversamento non trascurabile, ed è importante rappresentarlo con un elemento di ritardo.

Nel codice, vediamo che l' `assign` di `z` precede quello di `w`. Questo però non ha nulla a che vedere con le proprietà temporali che li legano: con queste linee di codice rappresentiamo componenti hardware distinti che si evolvono continuamente, indipendentemente e contemporaneamente. L'ordine degli statement di un `module` ha lo stesso valore dell'ordine con cui si disegnano le linee di uno schema circuitale: completamente irrilevante ai fini del risultato finale. Questo rimarrà vero quando vedremo reti più complesse, dove dimenticarsi di questo porta a errori gravi.

5.2 Concetto di testbench

Abbiamo progettato il nostro hardware, la rete `RC` di cui sopra. Vogliamo sapere però come si comporta, e in particolare se fa quello che ci aspettiamo dalle specifiche. Per far questo, ho bisogno di mettere `RC` in un contesto in cui ne manipolo gli ingressi in un modo noto, così da conoscere quali output aspettarsi, e con della logica apposita misuro le uscite e verifico che corrispondano a quelle attese. Tale ambiente è quello che chiamiamo *testbench*. Nell'esempio in figura, una rete *checker* controlla le uscite e con l'uscita `ok` indica se il test è andato a buon fine o no.



Il corrispettivo nel mondo software è un programma di test che prova i metodi e strutture dati di una libreria. Anche noto come *unit test*.

Una opzione è progettare questa testbench come un ulteriore componente hardware, e seguire tutti i passaggi necessari a realizzare *con hardware vero* la testbench con dentro la rete sopra descritta, per esempio con FPGA. Questo è sicuramente corretto, ma molto costoso, quantomeno nel tempo necessario a fare la verifica. Una opzione più interessante è usare la *simulazione*: si *compila* un programma eseguibile che simula il comportamento dell'hardware, almeno fino a un certo livello di dettaglio. Questo ci dà un responso in modo molto più efficiente, visto che si può modificare, ricompilare e rieseguire in pochi secondi vedendo il risultato direttamente a terminale. Si può fare un passo in più: anziché progettare la testbench come dell'altro hardware con semplice uscita `ok`, si sfrutta appieno la natura software della simulazione per scrivere qualcosa che è più simile a un *programma di test*, dove abbiamo effettivamente ordine e temporizzazione tra gli statement, insieme ad altri concetti che sarebbero privi di senso al di fuori della simulazione. Questo ci fornisce un modo per fare *debugging* su descrizioni di hardware.

```

module testbench();
    reg x_t, y_t; // "variabili"

```

```

wire z_t, w_t;

RC rc (
    .x(x_t), .y(y_t),
    .z(z_t), .w(w_t)
);

initial begin
    $dumpfile("waveform.vcd");
    $dumpvars;

    x_t = 0;
    y_t = 0;
    #10;
    if (z_t == 0 && w_t == 0)
        $display("0 0 -> 0 0 success");
    else
        $display("0 0 -> 0 0 fail");

    x_t = 0;
    y_t = 1;
    #10;
    if (z_t == 0 && w_t == 1)
        $display("0 1 -> 0 1 success");
    else
        $display("0 1 -> 0 1 fail");

    x_t = 1;
    y_t = 0;
    #10;
    if (z_t == 0 && w_t == 1)
        $display("1 0 -> 0 1 success");
    else
        $display("1 0 -> 0 1 fail");

    x_t = 1;
    y_t = 1;
    #10;
    if (z_t == 1 && w_t == 1)
        $display("1 1 -> 1 1 success");
    else
        $display("1 1 -> 1 1 fail");

end
endmodule

```

Le righe da 2 a 8 sono molto vicine a quello che vediamo nel disegno. Dichiariamo dei `reg` che useremo per pilotare gli ingressi della rete combinatoria, e dei `wire` che useremo per monitorarne le uscite. Dichiariamo poi la nostra rete combinatoria: lo statement a righe 5-8 è nella forma `tipo_modulo nome_istanza([lista porte]);`. Possiamo immaginare questo statement come equivalente dell'atto fisico di prendere un chip di tipo RC, che chiameremo con un nome d'istanza `rc` per distinguerlo dagli altri, e posizionarlo nella nostra rete collegandone i vari piedini con altri elementi: l'ingresso `x` al `reg x_t`, l'uscita `z` al `wire z_t`, e così via.

La notazione mostrata a righe 6-7 è con parametri nominati (*named parameters*), dove si indicano esplicitamente gli assegnamenti tra parametro del componente e componente esterno. Si può sempre utilizzare l'alternativa più nota - perché unica scelta in molti linguaggi, come C - ossia la notazione con parametri posizionali (*positional parameters*), dove l'associazione è data dalla corrispondenza con l'ordine di dichiarazione dei parametri.

Evitare parametri posizionali

La notazione con parametri posizionali può sembrare meno prolissa, ma è anche più pericolosa. In primo luogo, si basa sul fatto di ricordarsi *esattamente* l'ordine dei parametri, quando è invece facile distrarsi e scambiarsi di posto. In secondo luogo, non permette di *saltare* una posizione, mentre vedremo esempi dove collegare qualcosa a una o più uscite di una rete è del tutto opzionale.

Queste limitazioni possono sembrare semplici da aggirare, ma il vero problema è che a una semplice svista su un assegnamento di parametri posizionali corrisponde una lunga e faticosa fase di debug in cui tutto sembra comportarsi in modo completamente casuale.

Guardando le righe successive, ci sono diversi concetti che hanno un senso *in questo contesto* mentre altrove o hanno un senso *diverso* o sono del tutto privi di senso. Iniziamo dall'uso di `reg` come variabili, assegnando valori in serie come in un programma C. Nelle reti sincronizzate, vedremo che `reg` viene usato con significato e comportamento completamente diverso. Vediamo poi che usiamo un blocco `initial begin ... end`: questo contiene degli statement, eseguiti come un programma uno alla volta, separati talvolta da delle attese esplicite come `#10` che attende 10 unità di tempo. Il termine `initial` significa che il programma è eseguito "all'inizio della simulazione":

questo è un esempio di concetto completamente insensato per dell'hardware, dove non esiste un tempo 0. Altri statement che hanno senso solo in una simulazione sono `$display`, che stampa a terminale, e `$dumpfile` e `$dumpvars`, che producono invece un file `waveform.vcd` che possiamo studiare con GTKWave.

Leggendo il codice come un programma, vediamo che questa testbench altro non fa che testare tutti e 4 i possibili stati di `x` e `y`, confrontando le uscite `z` e `w` con i valori attesi.

Unità di tempo

Le unità temporali (sia di default che di volta in volta) si possono specificare, ma noi per semplicità non lo facciamo. Come vedremo dalle waveform, di conseguenza ogni valore viene interpretato di default come *secondi*, cosa decisamente poco realistica, ma comunque di nessun impatto per i nostri usi.

Per eseguire il test useremo tre programmi: `iverilog` e `vvp`, dalla suite [Icarus Verilog](#), e GTKWave. A differenza dell'ambiente per Assembler, questi sono facilmente reperibili per ogni piattaforma, o compilabili dal sorgente. [Qui](#) si trovano installer per Windows.

`iverilog` è il programma che *compila* la nostra simulazione. La sintassi è la seguente:

```
iverilog -o nome_simulazione testbench.v mia_rete.v [altri file .v]
```

I file per questo test sono scaricabili [qui](#) e [qui](#). Il file prodotto da `iverilog` non è direttamente eseguibile, ma va eseguito usando `vvp`:

```
vvp nome_simulazione
```

Otteniamo un output come il seguente:

```
VCD info: dumpfile waveform.vcd opened for output.
0 0 -> 0 0 success
0 1 -> 0 1 fail
1 0 -> 0 1 fail
1 1 -> 1 1 success
```

La prima riga è relativa ai comandi `$dumpfile` e `$dumpvars`, ci informa semplicemente che la simulazione sta effettivamente salvando i dati su `waveform.vcd`. Le righe successive sono invece quelle stampate dai nostri `$display`: vediamo che alcuni test sono falliti.

Stampe a fine simulazione

Alcune versioni di `iverilog` aggiungono *di default* una stampa del tipo “\$finish called at ...” al termine della simulazione, altre no.

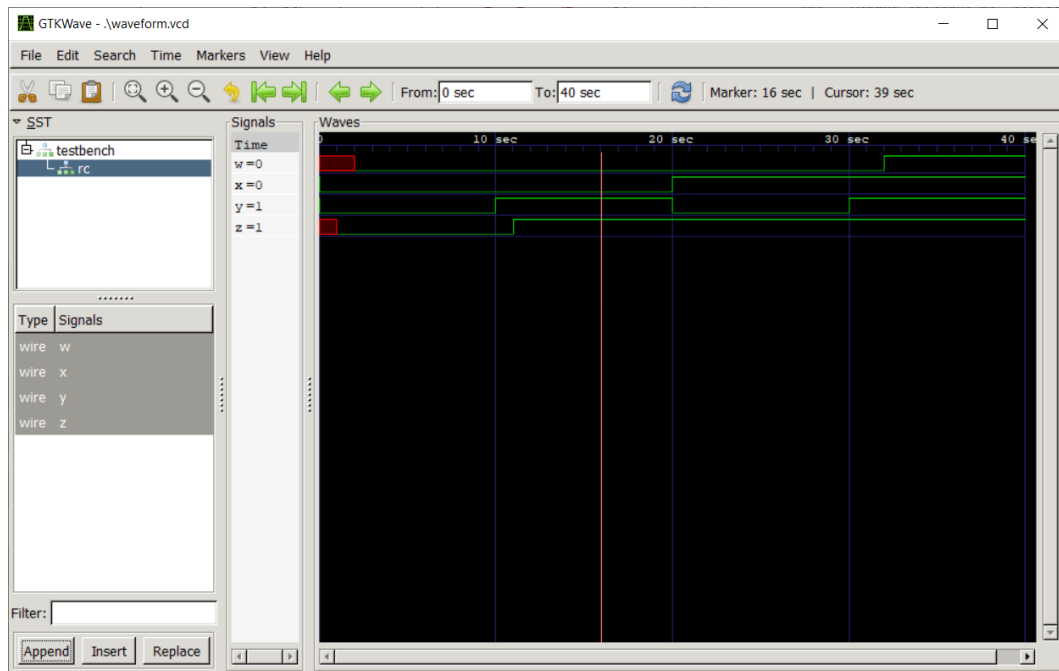
Chi ha ragione?

Un test che fallisce indica soltanto che il codice di test e il codice testato sono in disaccordo. La maggior parte delle volte, se fatto bene, il test rappresenta la specifica desiderata, mentre ciò che è testato ne indica solo l'implementazione. Per questo, di solito, ha ragione il test e va cambiato ciò che è testato.

Cerchiamo di capire perché il test fallisce, e quindi in cosa la rete RC non segue la specifica. Le stampe ci indicano i valori attesi e il fatto che non corrispondono con quelli prodotti da RC, non quali valori sono stati trovati in `z` e `w`. Potremmo cambiare le stampe per includerlo, ma è facile intuire che questo approccio non scala bene: non possiamo stampare a schermo tutte le variabili in tutte le situazioni. È per questo che si usa la waveform. Lanciamo GTKWave con il comando

```
gtkwave waveform.vcd
```

Si dovrebbe aprire quindi una finestra dal quale possiamo analizzare l'evoluzione della rete, filo per filo, nel tempo. Espandiamo le reti nel menu a sinistra, selezioniamo la rete `rc` e quindi gli input `x` e `y` e gli output `z` e `w`, clicchiamo poi *Append*. Otteniamo una schermata come quella in figura.



La schermata mostra l'evoluzione nel tempo dei fili selezionati, in particolare nel momento selezionato (la linea verticale rossa).

Significato delle waveform

GTKWave usa linee verdi con valore alto o basso per elementi da un singolo bit che hanno valore *logico* 0 o 1. In caso di elemento da più bit, utilizza linee verdi sopra e sotto il valore corrente dell'elemento (si può cambiare come sono interpretati i bit usando il menu contestuale).

Le aree di colore rosso indicano punti in cui il valore logico è *non specificato*, 'bx, tipicamente perché uno o più bit dell'elemento non sono unicamente determinabili. Una linea in mezzo di colore giallo vuol dire invece *alta impedenza*, 'bz, che non è un valore logico e vuol dire che, elettricamente, il filo non è connesso. Sia 'bx che 'bz hanno contesti ed usi utili in cui è normale che compaiano, ma *molto spesso* sono sintomo di un errore e un buon punto di partenza per il debug.

Vediamo dalla waveform i valori di *w* e *z* in corrispondenza dei test falliti: in entrambi i casi il test richiede *z* a 0 e lo trova a 1, *w* a 1 e lo trova a 0. Notiamo quindi che il test si aspetta che *z* si comporti come un AND e *w* come un OR, mentre vediamo che succede il contrario. Dobbiamo quindi scambiare gli assign delle due uscite.

```
module RC(x, y, z, w);

input x, y;
output z, w;

assign #1 z = x & y;
assign #2 w = x | y;

endmodule
```

Usare il reload in GTKWave

Una volta cambiato il codice, vorremmo ricompilare e rieseguire la simulazione. Ma il comando `gtkwave waveform.vcd` blocca il terminale finché non chiudiamo la finestra. Potremmo chiudere GTKWave e riavviarlo dopo, ma questo significa rifare daccapo tutto il setup per analizzare le waveform.

È per questo una buona idea utilizzare una delle seguenti strategie:

- usare due terminali, uno dedicato a `iverilog` e `vvp`, l'altro a `gtkwave`
- lanciare il comando in background. Nell'ambiente Windows all'esame, questo si può fare con un `&` in fondo: `gtkwave waveform.vcd &`

In entrambi i casi, otteniamo di poter rieseguire la simulazione mentre GTKWave è aperto. Possiamo quindi sfruttare il pulsante Reload, che caricherà le nuove waveform dall'ultima simulazione senza dover reimpostare l'interfaccia.

Se l'operatore & non funziona

In alcune installazioni di Powershell l'operatore & non funziona. L'operatore è un semplice alias per Start-Job, e si può ovviare al problema usando questo comando per esteso:

```
Start-Job gtwave waveform.vcd
```

L'operatore è documentato [qui](#).

5.3 Full adder, descrizione e sintesi di reti combinatorie

In generale, la differenza tra *descrizione* e *sintesi* è la seguente: una descrizione si limita a dire cosa una rete *fa*, senza scendere oltre nei dettagli implementativi; una sintesi mostra invece *come si implementa* questo comportamento. Una sintesi è un modo di realizzare una rete che si comporta come indicato dalla descrizione, e ci possono essere diversi modi (seguendo diversi modelli, algoritmi, criteri di costo) per sintetizzare una descrizione.

Per il caso delle reti combinatorie, vediamo l'esempio del *full adder*, partendo dal caso a 1 bit ([testbench](#), [descrizione](#), [sintesi](#)).

```
module full_adder(
    x, y, c_in,
    s, c_out
);
    input x, y;
    input c_in;
    output s;
    output c_out;

    assign #5 c_out, s = x + y + c_in;
endmodule
```

Sintassi: raggruppamento

Le parentesi graffe, come in `c_out, s`, si può usare per raggruppare elementi sia a destra che a sinistra di un assegnamento. Bisogna stare però attenti alle dimensioni in bit, e cosa viene assegnato a cosa.

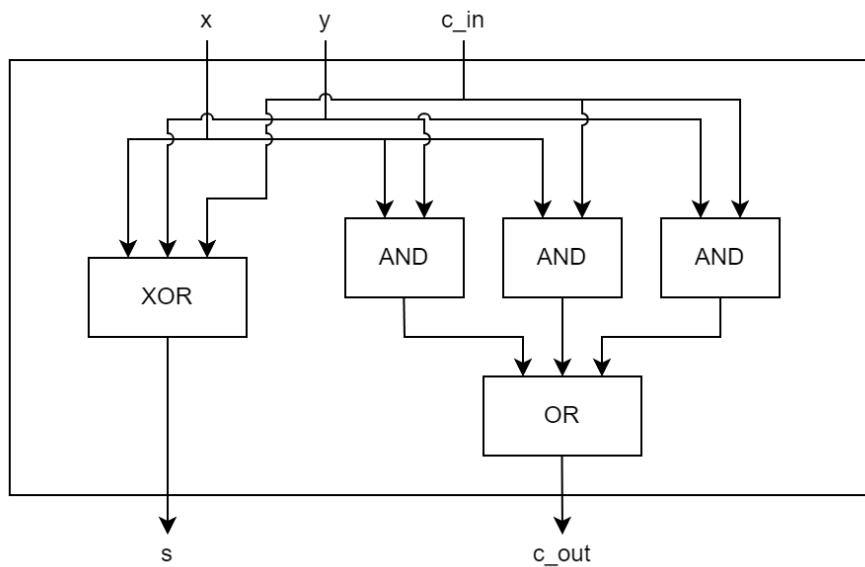
Questa è una *descrizione* del full adder: ci spiega cosa fa questo modulo, indicando le porte e la relazione tra ingressi e uscite, ma non ci dice nulla su come è implementata questa relazione. Infatti, la riga 10 utilizza l'operatore + del linguaggio Verilog, non ci spiega *come si fa* la somma. Quando si usano espressioni in questo modo, il compilatore Verilog non le traduce in hardware, ma ne calcola direttamente il risultato usando la nostra CPU a tempo di simulazione.

```
module full_adder(
    x, y, c_in,
    s, c_out
);
    input x, y;
    input c_in;
    output s;
    output c_out;

    assign #5 s = x ^ y ^ c_in;
    assign #5 c_out = ( x & y ) | ( y & c_in ) | ( x & c_in );
endmodule
```

Questa invece è una *sintesi* : ci mostra come realizzare il sommatore usando operatori logici elementari.

Un altro modo per definire *sintesi* è il fatto che siamo in grado, a partire dalla sintesi, di produrre lo schema circuitale corrispondente. Infatti, dal codice sopra possiamo ricavare il seguente schema.



Vediamo ora il caso di un full adder a 3 bit ([testbench](#), [descrizione](#), [sintesi](#)).

Per una descrizione, ci basta seguire l'esempio del caso a 1 bit, con l'aggiunta delle diverse dimensioni dei fili.

```

module full_adder_3(
    x, y, c_in,
    s, c_out
);
    input [2:0] x, y;
    input c_in;
    output [2:0] s;
    output c_out;

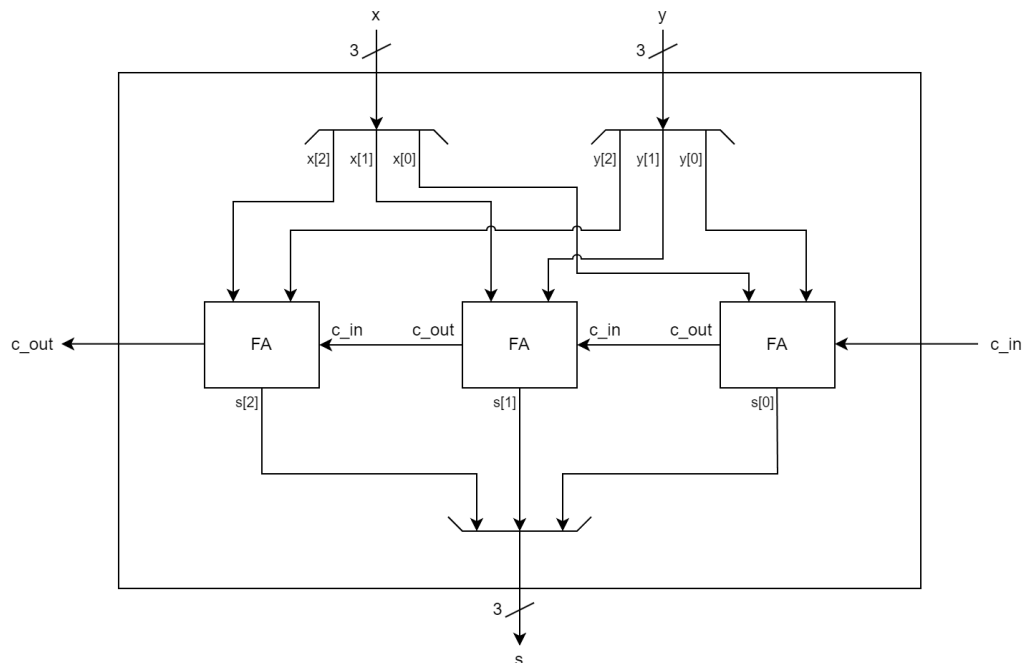
    assign #5 c_out, s = x + y + c_in;
endmodule
  
```

Sintassi: elementi di più bit

La dichiarazione con `[2:0]` indica che l'elemento è composto da 3 bit, indicizzati da 0 a 2. Questi indici possono poi essere utilizzati per selezionare uno più di componenti. Per esempio, con `x[2:1]` si selezionano i bit 2 e 1 di `x`, con `x[1]` solo il bit 1.

Come prima, questa è una descrizione perché non ci dice affatto come realizzare il sommatore, e non possiamo disegnare uno schema circuitale a partire da questo codice senza sapere già, da un'altra fonte, come realizzare un full adder a 3 bit.

Passiamo invece alla sintesi. Sappiamo che il full adder è un esempio di rete componibile, nel senso che possiamo realizzare un full adder a N bit usando N full adder a 1 bit. Vediamo come partendo, questa volta, dallo schema circuitale.



Da questo schema, si evince che sappiamo realizzare un full adder a 3 bit se sappiamo già realizzare un full adder a 1 bit. Questa relazione si conserva anche nel codice Verilog: nella sintesi di una rete combinatoria si possono utilizzare altre reti combinatorie di cui, a loro volta, si conosce la sintesi.

```
module full_adder_3(
    x, y, c_in,
    s, c_out
);
    input [2:0] x, y;
    input c_in;
    output [2:0] s;
    output c_out;

    wire c_in_1;
    full_adder fa_0 (
        .x(x[0]), .y(y[0]), .c_in(c_in),
        .s(s[0]), .c_out(c_in_1)
    );

    wire c_in_2;
    full_adder fa_1 (
        .x(x[1]), .y(y[1]), .c_in(c_in_1),
        .s(s[1]), .c_out(c_in_2)
    );

    full_adder fa_2 (
        .x(x[2]), .y(y[2]), .c_in(c_in_2),
        .s(s[2]), .c_out(c_out)
    );

endmodule
```

In questo codice riutilizziamo la rete `full_adder` che abbiamo sintetizzato prima. Notiamo come per farlo dobbiamo istanziare la rete tre volte, dandogli nomi diversi (`fa_0`, `fa_1`, `fa_2`), e dichiarare dei nuovi `wire` per collegarli, `c_in_1` e `c_in_2`. Infine, utilizziamo indici per indicare le componenti di `x` e `y` da collegare a ciascun `full_adder`, così come quale componente di `s` è collegata a quale uscita.

Di nuovo, possiamo vedere la corrispondenza tra il codice Verilog e lo schema circuitale: questo non è un caso, anzi è fondamentale. Tolto il caso limite delle testbench simulative, ogni cosa che scriviamo in Verilog ha senso solo se ci è chiaro che tipo di hardware corrisponde a ciò che scriviamo e come si può realizzare.

Questo vale anche quando si fa una descrizione. Per esempio, abbiamo prima visto come si può descrivere un full adder scrivendo `c_out, s = x + y + c_in`. Questo ci permette di essere meno prolissi, a patto che si sa come si fa un sommatore. Vedremo più avanti, nelle reti sincronizzate, esempi di cose che sono semplici da scrivere in descrizione, affidandosi al simulatore per eseguirne la logica, ma che si rivelano poi *molto difficili* da sintetizzare.

Capitolo 6

Esercitazione 2

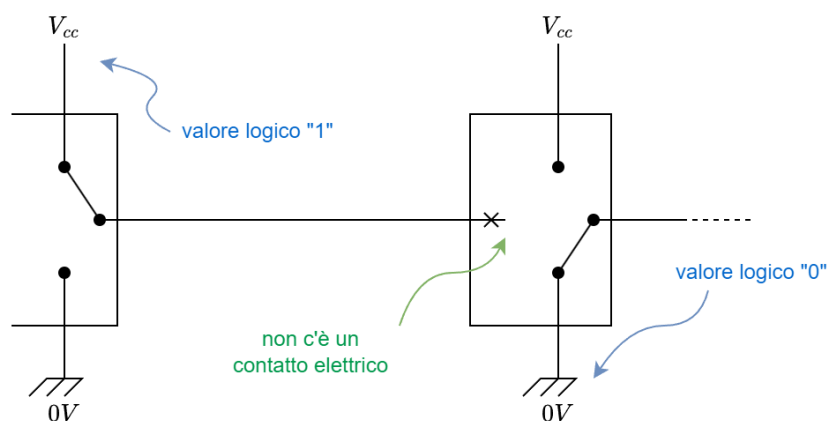
6.1 Errori comuni: i corto circuiti

Vediamo ora un esempio di come *non* tenere presente la corrispondenza tra Verilog e schemi circuitali porta a grossi guai.

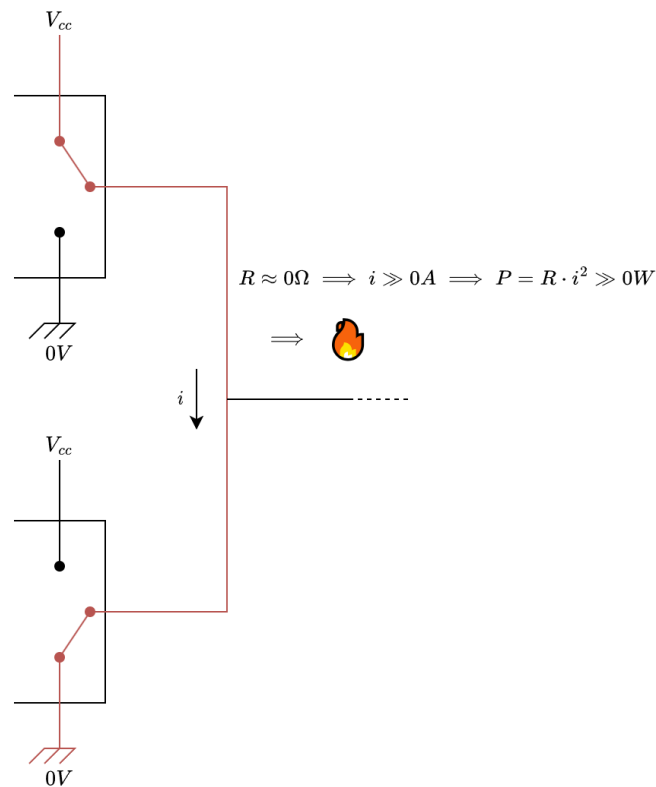
La maggior parte di quello di cui discutiamo in questo corso si applica per *qualunque* tecnologia si utilizzi per implementarle. Infatti, una volta ottenuti gli operatori logici elementari, sono identici i passaggi necessari per arrivare a costruire un processore in grado di eseguire programmi. Per esempio, c'è chi ha realizzato un processore funzionante usando la [redstone di Minecraft](#), così come ricerca sull'uso della luce, detta [photonic computing](#).

Attualmente, usiamo elettronica digitale basata su semiconduttori (studiata nel corso di Elettronica Digitale). Questo implica che dobbiamo stare attenti ai limiti imposti dall'elettronica quando realizziamo reti logiche, in particolare il fatto che non si possono collegare due o più fonti di tensione allo stesso filo. Vediamo perché.

Una porta logica agisce fondamentalmente come un interruttore che collega la propria uscita a *terra*, 0V, o alla tensione di alimentazione V_{cc} , per esempio 5V. All'ingresso di una porta logica, invece, viene rilevata la tensione senza contatti elettrici con l'uscita.



Cosa succede invece se colleghiamo le uscite di due porte logiche, in particolare se una produce 1 e l'altra 0? Si chiude un circuito che collega V_{cc} a *terra*. Data la differenza di potenziale, scorre corrente. Data la bassissima resistenza di un semplice filo, scorre tanta corrente. Data la relazione tra la potenza dissipata in calore e la corrente che attraversa il circuito, $P = R \cdot i^2$, viene dissipato tanto calore. Il circuito [prende fuoco](#) 🔥.



Arriviamo quindi al perché questo è un grosso problema all'esame: in Verilog, questa regola si traduce in non si possono fare due o più assign allo stesso wire. Questo è un errore tanto grave quanto è facile da fare, soprattutto se non si tiene a mente la corrispondenza con schemi circuitali come discusso sopra. Per dar fuoco al proprio circuito basta infatti scrivere:

```
wire filo;
...
assign #1 filo = ...;
...
assign #1 filo = ...;
```

Una forma (purtroppo) comune di questo errore è quello in cui si tenta di usare un wire come variabile accumulatore.

```
wire [7:0] filo;
...
rete_combinatoria rc_1 (
    .ingresso1(...), .ingresso2(...),
    .uscita(filo)
)

rete_combinatoria rc_2 (
    .ingresso1(...), .ingresso2(...),
    .uscita(filo)
)
...
```

Ci sono qui due errori in tandem: si parte dall'idea che `rc_1` e `rc_2` lavorino in sequenza anziché in parallelo, come due righe distinte di un programma, e si arriva a collegare sia l'uscita di `rc_1` che `rc_2` allo stesso filo, creando il corto circuito.

Dall'uso del simulatore Verilog questo problema non è sempre evidente: se due valori assegnati sono gli stessi, il simulatore "lascia fare" assegnando quel valore al filo, se invece i valori sono distinti il filo avrà valore logico indeterminato `1'bx`.

6.2 Uso efficiente di VS Code

Uso efficiente di VS Code

Questa parte della lezione copre l'uso efficiente di VS codice. Il materiale relativo si trova [qui](#).

6.3 Esercizi d'esame

Negli esercizi d'esame dove compare la sintesi di reti combinatorie, questa è *parte* di un esercizio più ampio: si chiede di realizzare una rete sincronizzata che interagisce con l'esterno per raccogliere input, svolgere un calcolo, e inviare un risultato. Viene chiesto di implementare tale calcolo con una rete combinatoria, da sintetizzare come modulo a parte utilizzato dalla rete sincronizzata.

Per esercitarsi, è possibile utilizzare tutti i test d'esame in questa forma prendendo in considerazione solo la parte relativa alla rete combinatoria e ignorando, per ora il resto. Uno svantaggio è il fatto che le testbench fornite sono relative all'esercizio per intero, interfacciandosi solo con la rete sincronizzata, e si dovrà realizzare da sé una testbench apposita per testare la sola rete combinatoria.

Materiale in costruzione

Non è ancora pronta, ma prevista, una guida adeguata alle testbench preparate per gli esercizi d'esame e come riadattarle per altri usi, per esempio per testare solo la parte combinatoria.

Vediamo alcuni esercizi di reti combinatorie prese da test d'esame.

6.4 Esercizio 2.1: parte combinatoria esame 2023-06-27

[Qui](#) il testo completo.

L'esercizio parla di una rete sincronizzata, che preleva due numeri naturali x e y , su 8 bit, e ha bisogno di calcolare $z = \max(x, y)$. Per ora, ci interessa soltanto la parte dove ci viene chiesto di sintetizzare la rete MAX che svolge questo calcolo.

Per testare tale rete, possiamo ricavarci una testbench come la seguente, scaricabile [qui](#).

```
module testbench();
    reg [7:0] x, y;
    wire [7:0] z;

    MAX m (
        .x(x), .y(y), .max(z)
    );

    initial begin
        x = 10; y = 5;
        #10;
        if(z != 10)
            $display("Test failed!");

        x = 5; y = 10;
        #10;
        if(z != 10)
            $display("Test failed!");

        x = 10; y = 10;
        #10;
        if(z != 10)
            $display("Test failed!");

        x = 100; y = 50;
        #10;
        if(z != 100)
            $display("Test failed!");

        x = 50; y = 100;
        #10;
        if(z != 100)
            $display("Test failed!");
    end
endmodule
```


Una versione più completa, ottenuta dalla testbench originale dell'esercizio prendendo lo schema del blocco consumer e i casi di test della funzione `get_testcase`, è scaricabile [qui](#).

Notiamo che, come per le testbench d'esame, questa emette output solo in caso di errore. Questo significa che quando lanciamo la simulazione, se vediamo a terminale *solo* le righe riguardo il file VCD e la `$finish` di fine simulazione, possiamo dire che la testbench *non ha trovato* errori. Questo non vuol dire che *non ci sono*, ed è sempre indicato di verificare da sé il corretto comportamento per tutti gli aspetti.

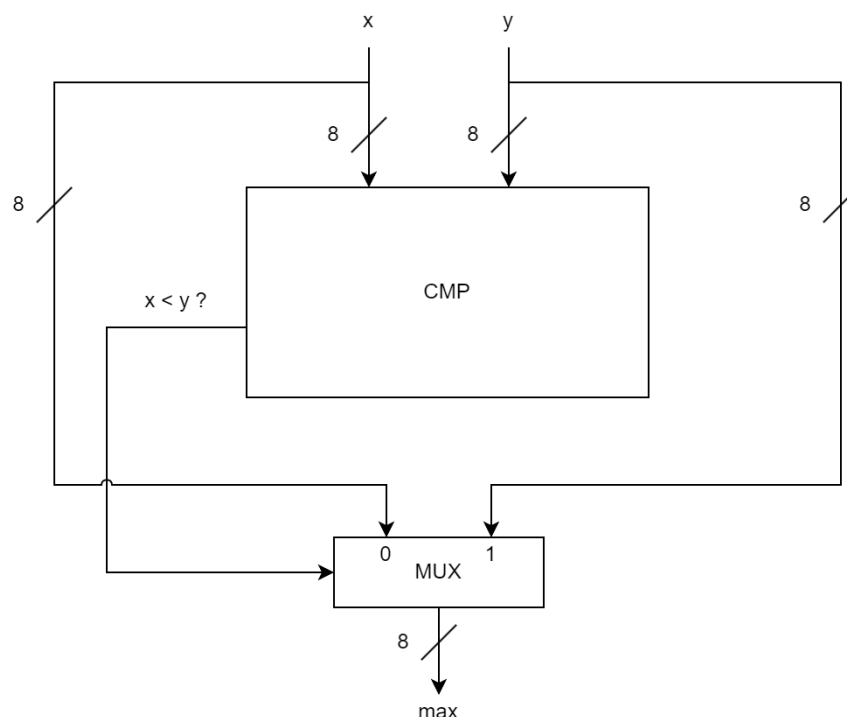
Vediamo ora il file `reti_standard.v`, anche questo fornito con l'esercizio. Questo file contiene delle reti combinatorie che si assume *note* e *sintetizzabili*. Ciò vuol dire che possiamo liberamente usarle come componenti nelle nostre sintesi di reti combinatorie - assieme alle porte logiche elementari e eventuali altre reti sintetizzate da noi nello stesso esercizio.

Controllare sempre `reti_standard.v`

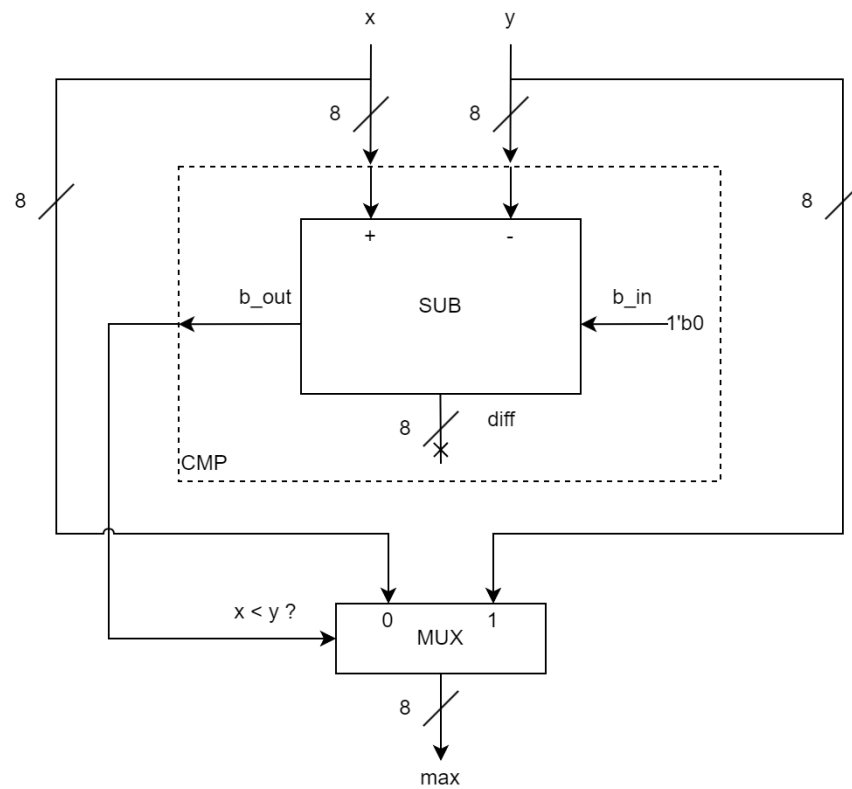
Il contenuto di `reti_standard.v` varia da esercizio ad esercizio. Questo sia in termini di reti fornite sia per la presenza o meno di parametri configurabili. Ciò è intenzionale, e la difficoltà di un esercizio è data anche da ciò che si è fornito come partenza.

In questo caso abbiamo a disposizione una sola rete combinatoria, il sommatore. Questo sommatore ha però un parametro, `N`. I parametri sono simili ai *generics* nei linguaggi di programmazione: un modo per scrivere un modulo configurabile che si adatta a più situazioni, che in questo caso vuol dire a un diverso numero di bit. Questo vuol dire che possiamo collocare nella nostra rete sommatore di qualunque numero di bit vogliamo, anzi *dobbiamo* trovare il numero giusto di bit da usare.

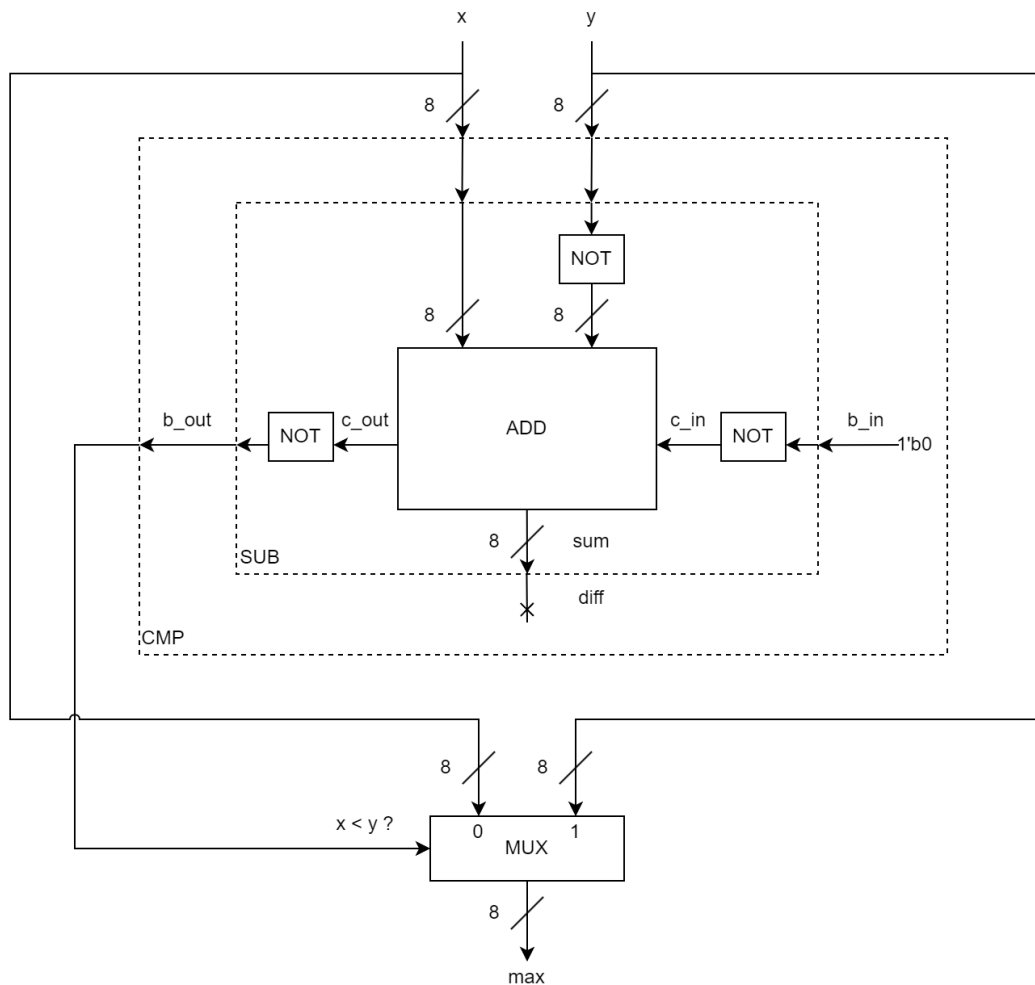
Prima di vedere la sintassi per usare queste reti parametriche, capiamo prima *come* lo vogliamo usare, ragionando sul problema con schemi circuitali. Una rete che determini il massimo tra due numeri dovrà necessariamente passare da una comparazione tra i due. Partiamo dall'idea di avere un comparatore il cui risultato fa da selettore per un multiplexer.



$x < y$ equivale a $x - y < 0$. Dato che x e y sono numeri naturali, questo equivale a chiedersi se la loro sottrazione genera un prestito uscente. Posso quindi realizzare questo comparatore usando un sottrattore.



Arriviamo quindi a come fare il sottrattore: sappiamo dal modulo di aritmetica che si può fare a partire da un sommatore: basta negare il sottraendo e i riporti in ingresso e uscita.



Abbiamo quindi una rete sintetizzabile: usiamo solo dei not, un multiplexer e un sommatore a 8 bit, quest'ultimo sintetizzabile perché parte della libreria `reti_standard.v`. Possiamo ora scrivere l'equivalente in Verilog, specificando per il sommatore $N = 8$. Questo parametro viene impostato all'istanziamento del sommatore, e deve essere una costante: determina infatti la quantità di hardware utilizzata, e non si può cambiare l'hardware a runtime.

```
module MAX(
    x, y,
    max
);
    input [7:0] x, y;
    output [7:0] max;

    wire [7:0] y_neg;
    assign #1 y_neg = ~y;

    wire c_out;
    add #(.N(8)) s (
        .x(x), .y(y_neg), .c_in(1'b1),
        .c_out(c_out)
    );

    wire b_out;
    assign #1 b_out = ~c_out;

    assign #1 max = b_out ? y : x;
endmodule
```

Uso di wire vs. assegnamento diretto

Esiste una certa flessibilità, soprattutto quando le reti si fanno più complesse, attorno alla sintesi esplicita con ritardi di operazioni come la negazione \sim e l'incremento $+1$.

Per esempio, in questo esercizio abbiamo dichiarato separatamente i wire `y_neg` e `b_out`, con dei ritardi negli assign relativi. È lecito però anche evitare questi wire e scrivere più compattamente `.y(~y)` a riga 13 e `~c_out ? y : x` a riga 20. Uno svantaggio di questo approccio è che, rimuovendo dei punti di ritardo, può rendere più difficile il debugging via waveform.

6.5 Esercizio 2.2: parte combinatoria esame 2023-01-31

[Qui](#) il testo completo.

Anche in questo caso, l'esercizio parla di una rete sincronizzata, che per ora ignoreremo. Per la rete sincronizzata, avremo da calcolare un prodotto di numeri naturali, ma abbiamo a disposizione solo `mul+add` da 4 bit (non parametrizzati). [Qui](#) la testbench riadattata per la sola rete combinatoria.

Questo esercizio segue in realtà lo stesso schema dell'equivalente già visto in *Assembler* ([qui](#)), cambia solo la base, che passa da $\beta_{asm} = 2^8$ a $\beta_{vrl} = 2^4$. Anche se matematicamente è lo stesso problema, cambia abbastanza come dovremmo risolverlo proprio perché stiamo descrivendo hardware e non programmando software.

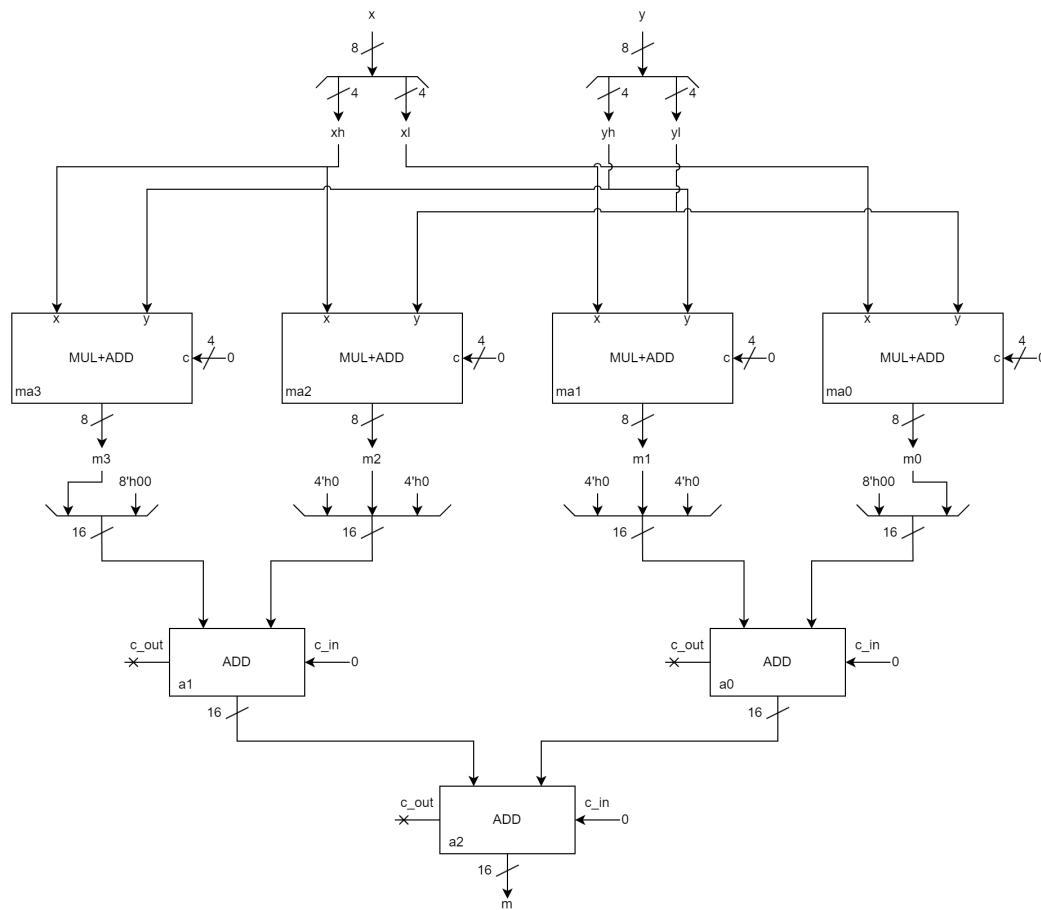
6.5.1 Soluzione 1

Una buona strategia, soprattutto quando si ha tempo limitato, è partire da soluzioni semplici ma funzionali, e passare poi a migliorarle.

Seguendo lo schema già visto, dovremo calcolare quattro sottoprodotti tra due cifre. Ciascun sottoprodotto può essere calcolato indipendentemente e produce un risultato su 2 cifre (8 bit). Questi sottoprodotti venivano shiftati a sinistra di 0, 1 o 2 cifre.

In *Assembler*, dopo questo passaggio si era già pronti a fare la somma su 4 cifre: questo perché i registri del processore hanno numeri di bit fissi, e le istruzioni a disposizione operano su questi numeri di bit. In Verilog, dove descriviamo hardware, i numeri di bit sono decisi da noi, ed è normale avere valori su numeri di bit diversi che non possiamo passare a un sommatore senza prima *estenderli*.

In altre parole, dobbiamo occuparci tanto degli zeri aggiunti a destra (per shift) quanto di quelli aggiunti a sinistra (per estensione) prima di poter sommare i sottoprodotti tra di loro. Una volta ottenuti i quattro sottoprodotti, tutti su 4 cifre, possiamo sommarli tra loro. Possiamo usare tre sommatore a 4 cifre (16 bit) per farlo. Otteniamo quindi lo schema seguente.



In Verilog, questo diventa quanto segue (scaricabile [qui](#)).

```
// x naturale su 8 bit
// y naturale su 8 bit
// m = x * y, su 16 bit
module MUL8(x, y, m);
  input [7:0] x;
  input [7:0] y;
  output [15:0] m;

  wire [3:0] xl, xh;
  assign xh, xl = x;
  wire [3:0] yl, yh;
  assign yh, yl = y;

  wire [7:0] m0;
  mul_add_nat ma0(
    .x(xl), .y(yl), .c(4'h0),
    .m(m0)
  );
  wire [15:0] m0e = 8'h00, m0;

  wire [7:0] m1;
  mul_add_nat ma1(
    .x(xl), .y(yh), .c(4'h0),
    .m(m1)
  );
  wire [15:0] m1e = 4'h0, m1, 4'h0;

  wire [7:0] m2;
  mul_add_nat ma2(
    .x(xh), .y(yl), .c(4'h0),
    .m(m2)
  );
  wire [15:0] m2e = 4'h0, m2, 4'h0;

  wire [7:0] m3;
  mul_add_nat ma3(
    .x(xh), .y(yh), .c(4'h0),
    .m(m3)
  );
```

```

);
wire [15:0] m3e = m3, 8'h00;

wire [15:0] s0;
add #(.N(16)) a0 (
    .x(m0e), .y(m1e), .c_in(1'b0),
    .s(s0)
);

wire [15:0] s1;
add #(.N(16)) a1 (
    .x(m2e), .y(m3e), .c_in(1'b0),
    .s(s1)
);

add #(.N(16)) a2 (
    .x(s0), .y(s1), .c_in(1'b0),
    .s(m)
);

```

endmodule

Si diceva dei corto circuiti...

Questo è un esempio del tipo di esercizi dove è comune vedere confusione: dovendo sommare quattro valori, usando tre sommatore, si può pensare di poter utilizzare un `wire` come una variabile accumulatore. Come spiegato, ciò non può funzionare in una rete combinatoria ed è un errore grave.

6.5.2 Soluzione 2

Nella soluzione precedente si può notare una inefficienza: abbiamo a disposizione solo reti `mul+add`, che hanno anche un ingresso `c` e calcolano $(x \cdot y) + c$, ma stiamo ignorando questa possibilità impostando tutti gli `c` a 0. Per ottimizzare, possiamo quindi cercare un modo di sfruttare questi ingressi per ridurre il numero di sommatore. Ritorniamo alla somma in colonna dei sottoprodotti.

```

0  0  [ m0 ]  +
0  [ m1 ] 0  +
0  [ m2 ] 0  +
[ m3 ] 0  0  +

```

Se guardiamo a `m0`, notiamo che la sua parte bassa è destinata a finire nella somma finale senza alcuna modifica, e dunque i sommatore sono, per quella parte, completamente superflui. La stessa cosa accade per i bit più significativi, dove troviamo tanti zeri sommati tra di loro. Per vedere meglio come procedere, scomponiamo anche i sottoprodotti in parte alta e parte bassa, per esempio `m0h`, `m0l` = `m0`.

```

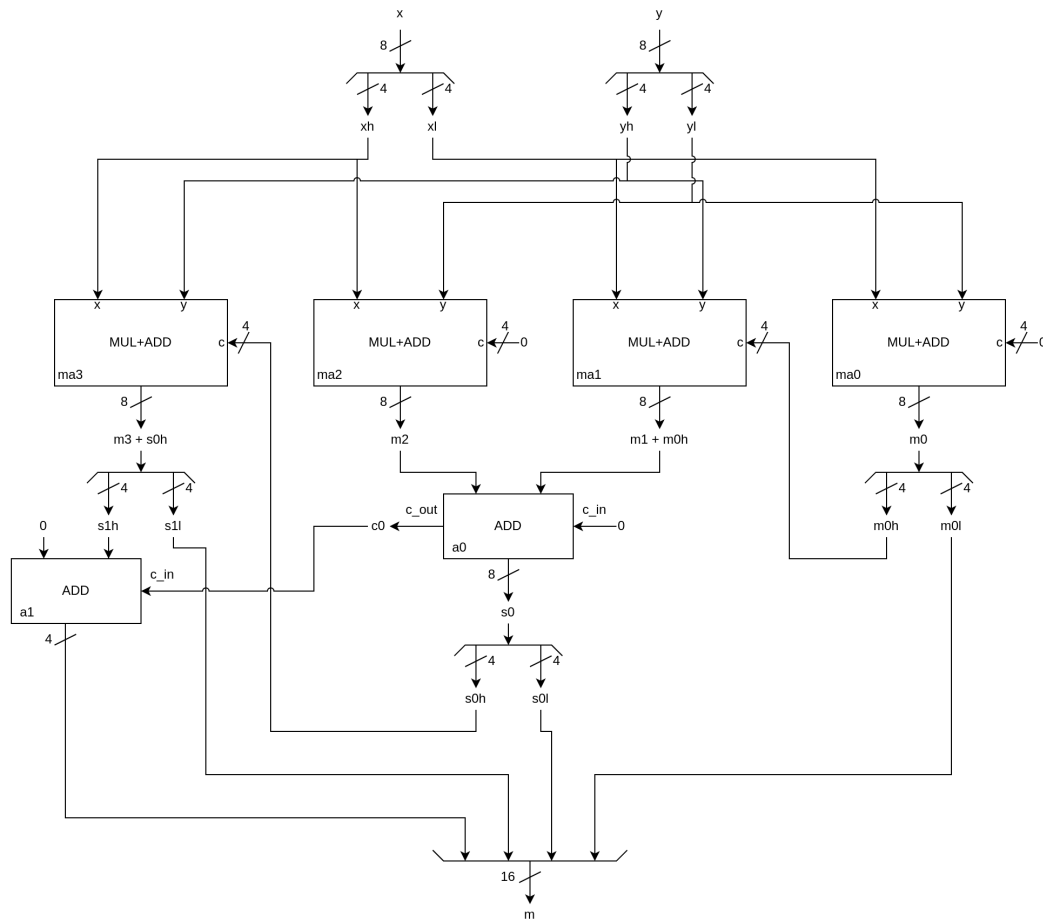
0    0    m0h  m0l  +
0    m1h  m1l  0    +
0    m2h  m2l  0    +
m3h  m3l  0    0    +

```

Ecco quindi le ottimizzazioni che si possono eseguire:

- `m0l` non viene sommato ad alcunché, dunque `m[3:0] = m0l`
- `m0h` può essere collegato all'input `c` di `ma1` o `ma2`
- sia `s0` la somma `m1 + m2 + m0h`, scomposto in `s0h`, `s0l` = `s0`, e `c0` il suo eventuale riporto uscente. Allora `m[7:4] = s0l`, e `s0h` può essere collegato all'input `c` di `ma3`.
- sia `s1` la somma `m3 + s0h`, scomposto in `s1h`, `s1l` = `s1`. Allora `m[11:8] = s1l`, mentre `m[15:12] = s1h + c0`.

Per realizzare questo ci serve, oltre ai moltiplicatori, un sommatore a 8 bit per `s0`, e un incrementatore a 4 bit per `s1h + c0`. Lo schema che lo rappresenta è il seguente.



In Verilog, questo diventa quanto segue (scaricabile [qui](#)).

```
// x naturale su 8 bit
// y naturale su 8 bit
// m = x * y, su 16 bit
module MUL8(x, y, m);
  input [7:0] x;
  input [7:0] y;
  output [15:0] m;

  wire [3:0] xh, xl;
  assign xh, xl = x;
  wire [3:0] yh, yl;
  assign yh, yl = y;

  wire [7:0] m0;
  mul_add_nat ma0(
    .x(xl), .y(yl), .c(4'b0000),
    .m(m0)
  );
  wire [3:0] m0h, m0l;
  assign m0h, m0l = m0;

  wire [7:0] m1_m0h;
  mul_add_nat ma1(
    .x(xl), .y(yh), .c(m0h),
    .m(m1_m0h)
  );

  wire [7:0] m2;
  mul_add_nat ma2(
    .x(xh), .y(yl), .c(4'b0000),
    .m(m2)
  );

  wire [7:0] s0;
  wire c0;
  add #( .N(8) ) a0 (
    .x(m2), .y(m1_m0h), .c_in(1'b0),
    .c_out(c0),
    .s(s0)
  );
  wire [3:0] s0h, s0l;
  assign s0h, s0l = s0;

  wire [7:0] m3_s0h;
  mul_add_nat ma3(
    .x(xh), .y(yh), .c(c0),
    .m(m3_s0h)
  );
  wire [3:0] m3h, m3l;
  assign m3h, m3l = m3_s0h;

  wire [7:0] s1;
  add #( .N(8) ) a1 (
    .x(m3h), .y(m3l), .c_in(c0),
    .c_out(c0),
    .s(s1)
  );
  wire [3:0] s1h, s1l;
  assign s1h, s1l = s1;

  wire [15:0] m;
  assign m = {s1h, s1l, m3l, m3h};
endmodule
```

```
        .s(s0), .c_out(c0)
    );
    wire [3:0] s0h, s0l;
    assign s0h, s0l = s0;

    wire [7:0] m3_s0h;
    mul_add_nat ma3(
        .x(xh), .y(yh), .c(s0h),
        .m(m3_s0h)
    );
    wire [3:0] s1h, s1l;
    assign s1h, s1l = m3_s0h;

    wire [3:0] s1h_c0;
    add #(.N(4)) a1(
        .x(s1h), .y(4'b0), .c_in(c0),
        .s(s1h_c0)
    );

    assign m = s1h_c0, s1l, s0l, m0l;
endmodule
```

Capitolo 7

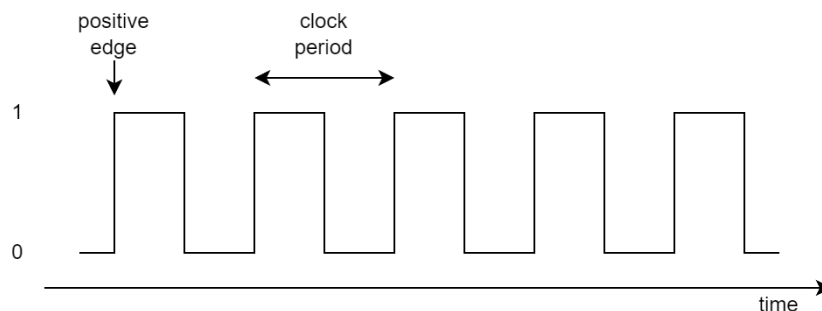
Esercitazione 3

7.1 Reti sincronizzate

Le reti sincronizzate sono reti logiche con uno stato interno, mantenuto usando registri, che si evolvono a istanti discreti dati da un segnale di clock. In questa esercitazione vedremo come realizzarle, simularle e studiarle nell'ambiente d'esame.

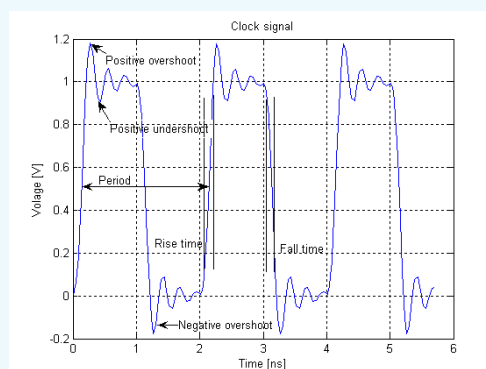
7.1.1 Testbench e generatore di clock

Per poter simulare una rete sincronizzata dobbiamo innanzitutto avere un generatore di clock. Il segnale di clock è segnale oscillante, che dal punto di vista logico appare come in figura.



Clock in realtà

In realtà i generatori di clock sono basati su cristalli di quarzo, un materiale *piezoelettrico* con il quale si possono realizzare circuiti oscillanti. Questi circuiti emettono una tensione oscillante come mostrato in figura (da Wikimedia), notare come l'onda sia molto meno squadrata di quanto presentato a livello logico.



Per i nostri usi, ci basterà descrivere una rete asincrona che cambia il proprio segnale da 0 a 1, e viceversa, ad intervalli regolari. Una qualunque descrizione realistica, e dunque *sintetizzabile*, dovrebbe avere a che fare con un segnale di reset che indichi a questa rete da che punto cominciare. Dato però che vogliamo usare questo generatore in una testbench simulativa, possiamo utilizzare direttamente i concetti relativi, approfittando della keyword `initial` per mantenere il codice *semplice*.


```
// generatore del segnale di clock
module clock_generator(
    clock
);
    output clock;

    parameter HALF_PERIOD = 5;

    reg CLOCK;
    assign clock = CLOCK;

    initial CLOCK <= 0;
    always #HALF_PERIOD CLOCK <= ~CLOCK;

endmodule
```

Notiamo che questa rete *non è sintetizzabile*. Infatti, utilizza la keyword `initial`, che è priva di senso in hardware, e un `reg` non come registro ma come variabile, come già visto nelle testbench. Questo si nota dal fatto che il `reg` non viene aggiornato in risposta a un altro segnale, come il positive edge del clock, come invece accade per registri.

Periodo del clock

Il parametro `HALF_PERIOD` rende il periodo di questo generatore di clock configurabile. Tipicamente all'esame viene utilizzato il valore default di 5, che implica periodi di clock di 10 unità di tempo. Qualora questo cambiasse (per esempio, per permettere reti combinatorie con maggior tempo di attraversamento) sarà segnalato nel testo.

Come ogni altra rete, questa viene inclusa nella testbench con una istanziazione.

```
module testbench();
    wire clock;
    clock_generator clk(
        .clock(clock)
    );
    ...
    mia_rete dut(
        ...
        .clock(clock)
    );
    ...
endmodule
```

Oltre al segnale di clock, una rete sincronizzata avrà bisogno anche del segnale di reset. Questo viene aggiunto come un `reg` pilotato all'inizio del blocco `initial` della testbench.

```
module testbench();
    ...
    reg reset_;
    ...
    mia_rete dut(
        ...
        .clock(clock), .reset_(reset_)
    );
    ...
    initial begin
        reset_ = 0;
        #(clk.HALF_PERIOD);
        reset_ = 1;
        ...
    end
endmodule
```

Con la sintassi `#(clk.HALF_PERIOD);` si attendono unità di tempo proporzionali al periodo di clock configurato. Questo è utile ad evitare di modificare manualmente tutte le attese in caso di cambio di clock.

7.1.2 Primo esempio di rete sincronizzata: il contatore

Vediamo ora un semplice esempio di rete sincronizzata, un contatore. Questa rete ha un registro di 3 bit inizializzato a 0, che viene incrementato ad ogni ciclo di clock, facendo infine wrap-around da 7 a 0. Il codice è scaricabile [qui](#).

```
module contatore (
    out,
    clock, reset_
);
    output [2:0] out;
    input clock, reset_;
```

```

reg [2:0] OUT;
assign out = OUT;

always @(reset_ == 0) begin
    OUT <= 0;
end

always @(posedge clock) if (reset_ == 1) #3 begin
    OUT <= OUT + 1;
end
endmodule

```

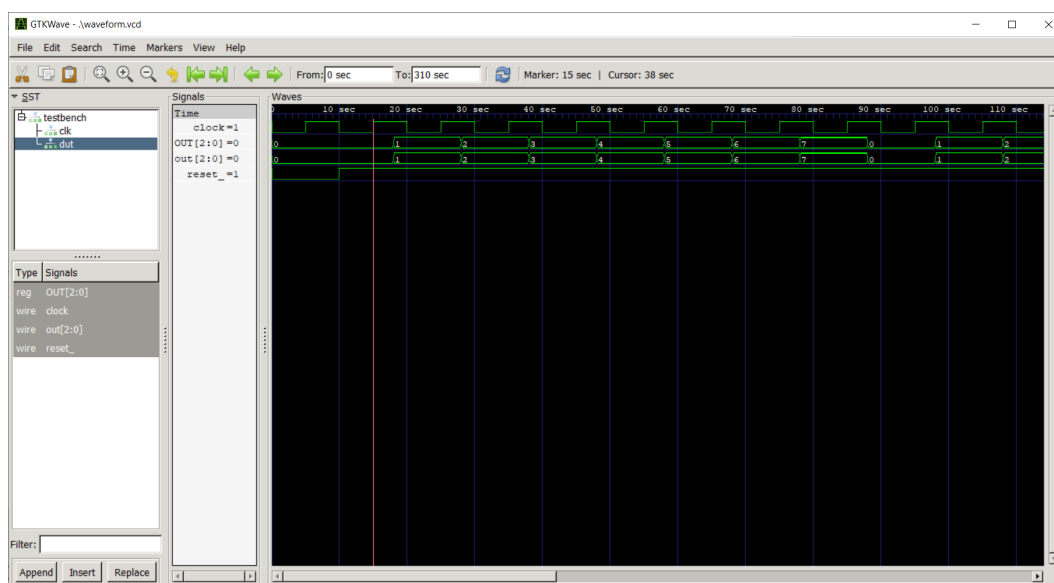
Vediamo quindi come questo codice modella il comportamento di una vera rete sincronizzata. Il reg OUT viene collegato direttamente all'uscita out, viene inizializzato a 0 solo in corrispondenza del segnale di reset (righe 11-13) e, durante la normale operazione, viene aggiornato con il valore $OUT + 1$ in corrispondenza di un posedge del clock. Il ritardo #3 modella il tempo $T_{propagation}$ del registro.

Assegnamento con <=

È importante, nelle reti sincronizzate, utilizzare <= per assegnamenti a registri sul fronte positivo del clock. Questo perché gli statement con <= sono intesi come eseguiti *in parallelo*, non *sequenzialmente*. Infatti, i registri non sono variabili e i loro cambiamenti non sono visibili agli altri registri fino al successivo posedge del clock.

Possiamo vedere come si evolve questa rete, simulandola in una testbench con segnale di clock e reset_ (scaricabile [qui](#)). Per il resto, la testbench non fa altro che attendere diversi cicli di clock, visto che questa rete non ha alcun input e si evolve in autonomia.

Vediamo l'evoluzione della rete usando GTKWave.



Osserviamo, in particolare, il registro OUT e il segnale del clock. Notiamo che a ogni fronte positivo del clock, OUT non cambia immediatamente, ma solo dopo 3 unità di tempo.

Dalla teoria sui registri, ricordiamo anche che il nuovo valore assunto dal registro deve essergli dato in input da T_{setup} prima del posedge e fino a T_{hold} dopo il posedge. Questo però non si nota da questa waveform: non c'è nulla che rappresenta il valore in ingresso al registro prima del posedge. Torniamo al codice: alla riga 16 usiamo una espressione combinatoria a sinistra dell'assegnamento. Questa espressione viene calcolata dal simulatore Verilog al momento dell'assegnamento, cioè $t_{posedge} + 3$, e non prima. Ciò significa che il simulatore non sta simulando né il risultato combinatorio in ingresso al registro, né il fatto che sia settato e mantenuto per i giusti tempi.

Possiamo ovviare al primo di questi problemi introducendo un wire, che ci rappresenti la rete combinatoria che calcola il successivo valore di OUT (scaricabile [qui](#)).

```

module contatore (
    out,
    clock, reset_
);
    output [2:0] out;
    input clock, reset_;

    reg [2:0] OUT;

```

```

assign out = OUT;

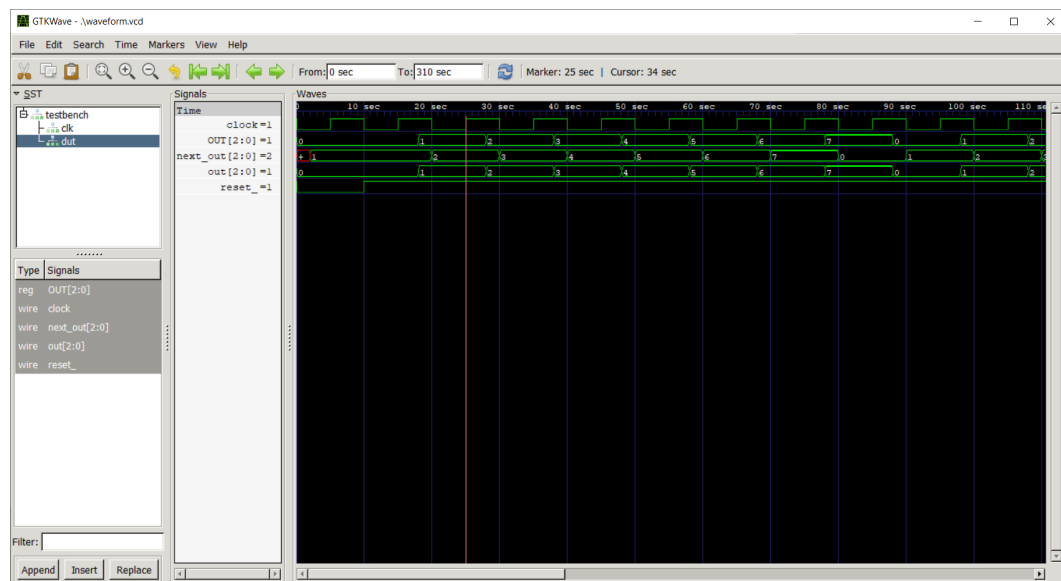
wire [2:0] next_out;
assign #2 next_out = OUT + 1;

always @(reset_ == 0) begin
    OUT <= 0;
end

always @(posedge clock) if (reset_ == 1) #3 begin
    OUT <= next_out;
end
endmodule

```

Simulando questa nuova rete, otteniamo la seguente waveform.



Vediamo ora chiaramente che la rete combinatoria che produce `next_out` risponde quasi immediatamente, ma il registro `OUT` non registrerà il suo valore fino al prossimo posedge del clock. Infatti, il periodo che separa due posedge è utilizzato proprio per far propagare i nuovi valori dei registri attraverso reti combinatorie, che andranno a produrre i nuovi ingressi dei registri che questi registreranno al prossimo posedge.

Questo modo di propagarsi dei valori tra un ciclo di clock è l'altro è fondamentale per capire come funzionano le reti sincronizzate ed essere quindi in grado di scrivere Verilog corrispondente alla macchina a stati che vogliamo realizzare. Allo stesso modo, riuscire a leggere questa evoluzione dalla waveform è fondamentale per rendere queste utili al debugging.

7.1.3 Mantenere un segnale per N cicli di clock

Vediamo ora l'esempio di una rete sincronizzata con uscita `out` a 1 bit, che, ciclicamente, viene tenuta a 1 per `N` clock e messa a 0 per 1 clock. Il codice è scaricabile in due versioni, [qui](#) senza wire di debug e [qui](#) con, mentre la testbench è [qui](#).

Per semplicità, discutiamo direttamente la versione che usa wire di debug per evidenziare gli ingressi dei registri.

```

module out_n_clock(
    out,
    clock, reset_
);
    output out;
    input clock, reset_;

    reg OUT;
    assign out = OUT;

    localparam N = 3;
    reg [3:0] COUNT;

    reg STAR;
    localparam S0 = 0, S1 = 1;

```

```

always @(reset_ == 0) begin
    COUNT <= N;
    OUT <= 0;
    STAR <= S0;
end

wire [3:0] next_count_s0;
assign #2 next_count_s0 = COUNT - 1;

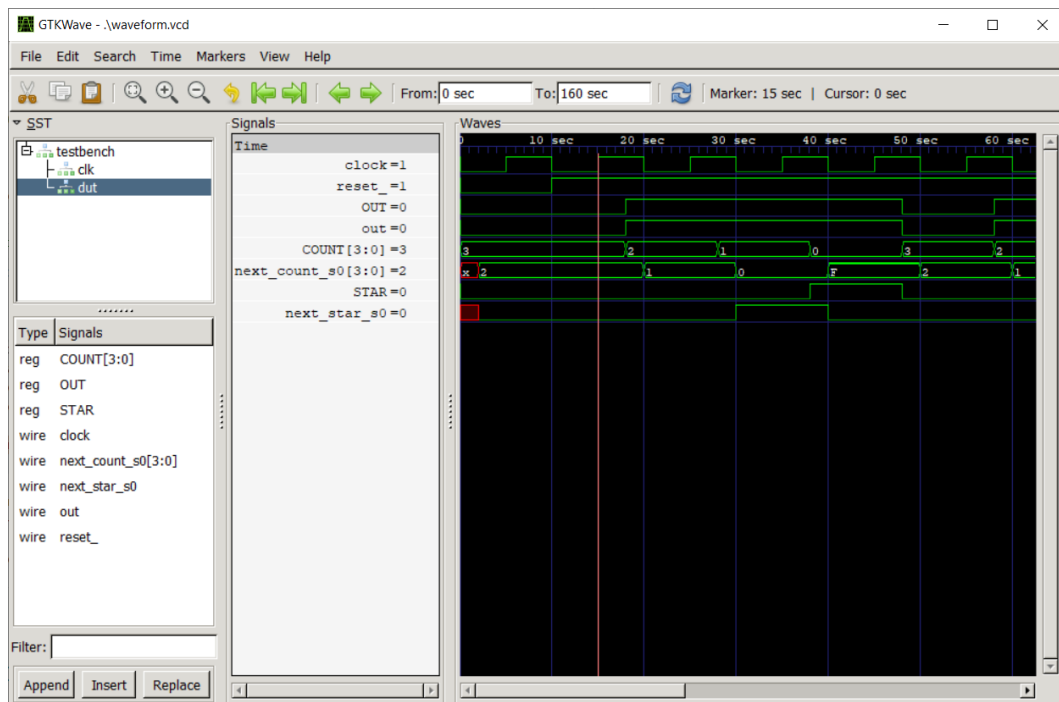
wire next_star_s0;
assign #2 next_star_s0 = (COUNT == 1) ? S1 : S0;

always @(posedge clock) if (reset_ == 1) #3 begin
    casex (STAR)
        S0: begin
            COUNT <= next_count_s0;
            OUT <= 1;
            STAR <= next_star_s0;
        end

        S1: begin
            COUNT <= N;
            OUT <= 0;
            STAR <= S0;
        end
    endcase
end
endmodule

```

Questa rete inizializza COUNT a N e, in S0, lo decrementa continuamente. Anziché saltare da S0 a S1 quando COUNT raggiunge 0, lo facciamo invece quando raggiunge 1. Perché? Guardiamo la waveform per capirlo meglio.



A $t = 15$ corrisponde il primo posedge del clock con $reset_a$ 1. Notiamo che a questo punto OUT è a 0 per via dell'inizializzazione, ma a $t = 18$ questo passa a 1, conseguenza della riga 33. Per $COUNT$, invece, notiamo che è stato inizializzato a 3, e subito dopo $next_count_s0$ ha calcolato 2 come prossimo valore. A $t = 18$, $COUNT$ decrementa a 2. Passiamo ora al clock successivo, cioè $t = 25$. $COUNT$ vale 2, assume poco dopo valore 1, a $t = 28$. Notiamo $next_star_s0$ a cavallo di questo cambiamento: subito dopo il passaggio di $COUNT$ a 1, $next_star_s0$ diventa S1. Siamo però a $t = 30$, non $t = 25$: il check sul valore di $COUNT$ non cambia fino a *dopo* il posedge del clock, e quindi $STAR$ rimane S0 per un altro ciclo. Al ciclo dopo, a $t = 38$, vediamo che lo stato passa dunque a S1, ma OUT resta 1: infatti solo al clock dopo il cambio di stato avrà effetto sui registri, compreso $COUNT$ che viene reinizializzato. Guardando il filo OUT in tutto ciò, notiamo che è rimasto effettivamente a 1 per $N = 3$ cicli di clock, come da specifica. Ci sono diversi fattori che possiamo cambiare, ottenendo risultati diversi. Se inizializziamo $COUNT$ a $N - 1$, seguendo la stessa logica dovremmo contare fino a 0. Se anticipiamo il cambio di OUT , usando $OUT <= (COUNT == 1) ? 0 : 1$, allora si perde il ciclo in più con OUT a 1 e dovremmo cambiare il conteggio di conseguenza. Anche per strutture apparentemente così semplici, è quindi possibile combinare tanti approcci diversi al punto tale che è difficile dedurre

a colpo d'occhio la durata del segnale. È per questo importante sapere come si evolvono i vari registri, come si propagano i loro cambi di valori, come ricostruire (e leggere) una waveform.

Partire da N bassi

Nello ragionare su comportamenti di questo tipo, che sia a mente o su carta, è una buona idea partire da N bassi, come 1 o 2, e calcolare la durata del segnale in termini di N. Per esempio, in questo esercizio abbiamo visto che inizializzando COUNT a N otteniamo OUT a 1 per N cicli di clock. Questo varrà che N sia 3 o che sia 12, ma $N = 3$ è molto più rapido da verificare, e $N = 2$ lo è ancora di più.

7.1.4 Esercizio: Handshake e reti combinatorie

Vediamo un esempio di semplice esercizio che segue uno schema tipico all'esame. Il testo è scaricabile [qui](#).

La testbench è più complessa di quanto visto finora, nella prossima sezione vedremo le principali caratteristiche utili per debugging. Per ora, vediamo come si realizza una rete che risponde a queste specifiche.

Il testo ci chiede di eseguire un handshake `dav_`/rfd con il produttore. Questo handshake prevede che, tramite il filo `rfd` ("ready for data") che va dal consumatore al produttore e il filo `dav_` ("data valid", attivo basso) che va dal produttore al consumatore, questi si coordinino per la corretta trasmissione del dato. Ricordiamo i passi di questo protocollo:

- A riposo: `dav_ = 1`, `rfd = 1`.
- Comincia il produttore: `dav_ = 0`. Questo segnala che il dato è valido.
- ACK del consumatore: `rfd = 0`. Questo segnala che il dato è stato letto.
- Reset del produttore: `dav_ = 1`.
- Reset del consumatore: `rfd = 1`.

Un *protocollo*, in generale, descrive come due o più attori devono interagire tra loro. Quando implementiamo un attore di un protocollo, ci sono due punti importanti da ricordare perché questo funzioni:

4. vanno eseguiti *tutti* gli step che ci competono, *quando* il protocollo ci dice di farlo;
4. quando il protocollo dice che *qualcun'altro* deve segnalare qualcosa, dobbiamo *attendere* che questo accada.

In questo esercizio, implementiamo il consumatore, che deve prelevare un dato dal produttore. Rileggiamo quindi il protocollo di sopra dal punto di vista del consumatore, per capire cos'è che dobbiamo fare nella nostra rete.

- A riposo, e in particolare al reset iniziale: `rfd = 1`.
- Attendiamo che il produttore segnali `dav_ = 0`.
- Leggiamo il dato.
- Comuniciamo l'avvenuta lettura con `rfd = 0`.
- Attendiamo che il produttore segnali `dav_ = 1`.
- Segnaliamo il reset del protocollo con `rfd = 1`.

Dall'altra parte, una volta ottenuto il dato valido per `a` e `b`, svolgiamo il conto prescritto utilizzando una rete combinatoria e ne emettiamo il risultato tramite l'uscita `p`.

Ordine delle operazioni

Non c'è nessuna prescrizione rigida sull'ordine delle operazioni tra gli step del protocollo e l'immissione del dato in uscita. È valido sia completare l'handshake fino al suo reset e *poi* trasmettere il dato, sia trasmettere immediatamente il dato e poi chiudere l'handshake.

La testbench dovrà tenere conto di ciò.

Nello svolgere il calcolo, dobbiamo implementare una semplice rete combinatoria. L'aspetto più interessante è come usarla: dobbiamo assicurarci di campionarne l'output solo quando gli input relativi sono validi.

Da questi ragionamenti, deriviamo la seguente descrizione, scaricabile [qui](#).

```

module ABC(
    a, b, p,
    dav_, rfd,
    clock, reset_
);
    input [3:0] a, b;
    output [5:0] p;

    input dav_;
    output rfd;

    input clock, reset_;

    reg [5:0] P;
    assign p = P;

    reg RFD;
    assign rfd = RFD;

    reg [3:0] A, B;

    wire [5:0] out_rc;
    PERIMETRO_RC rc(
        .a(A), .b(B),
        .p(out_rc)
    );

    reg [2:0] STAR;
    localparam
        S0 = 0,
        S1 = 1,
        S2 = 2,
        S3 = 3;

    always @(reset_ == 0) begin
        RFD <= 1;
        P <= 0;
        STAR <= S0;
    end

    always @(posedge clock) if (reset_ == 1) #3 begin
        casex (STAR)
            S0: begin
                A <= a;
                B <= b;
                STAR <= (dav_ == 0) ? S1 : S0;
            end

            S1: begin
                P <= out_rc;
                STAR <= S2;
            end

            S2: begin
                RFD <= 0;
                STAR <= (dav_ == 1) ? S3 : S2;
            end

            S3: begin
                RFD <= 1;
                STAR <= S0;
            end
        endcase
    end
endmodule

module PERIMETRO_RC(
    a, b,
    p
);
    input [3:0] a, b;
    output [5:0] p;

    wire [4:0] somma;
    add #( .N(4) ) adder(
        .x(a), .y(b), .c_in(1'b0),
        .s(somma[3:0]), .c_out(somma[4])
    );

```

```

    assign p = somma[3:0], 1'b0 ;
endmodule

```

7.1.5 Testbench con input e output per reti sincronizzate

Ci muoviamo ora verso reti sincronizzate più complesse, che prendono input da altre reti, svolgono conti, ed emettono output.

Blocchi in parallelo: fork ... join

Per scrivere testbench per reti combinatorie abbiamo sfruttato la loro inerente semplicità: dato un nuovo ingresso, una rete combinatoria emette l'output corrispondente dopo un certo tempo. Questo output non varierà nel tempo finché manteniamo l'input costante, anzi in ogni caso allo stesso input corrisponde lo stesso output. Questo ci permette di scrivere testbench semplici basate sulla struttura 1) assegno gli ingressi, 2) attendo un tempo sufficiente, 3) controllo le uscite.

Questo non è però fattibile con le reti sincronizzate: a un singolo ingresso possono corrispondere diversi cambi di stato ed uscite diverse, e il tempo necessario al calcolo è difficilmente prevedibile. Inoltre, se la rete si coordina tramite handshake con altre reti, non si può determinare a priori in quale ordine eseguirà questi handshake. È necessario quindi adottare una struttura che permette a ciascun componente con cui la rete testata interagisce di comportarsi come un componente *indipendente* che non è bloccato dal proseguire degli altri - proprio come hardware vero.

Questo è possibile con il costrutto `fork ... join`. All'interno di un `fork ... join` possiamo definire diversi blocchi `begin ... end` il cui codice verrà eseguito indipendentemente e in parallelo. Possiamo quindi sfruttare questo per rappresentare diversi componenti.

```

fork
    begin : Producer_1
        ...
    end

    begin : Producer_2
        ...
    end

    begin : Consumer
        ...
    end
join

```

All'interno dei blocchi `Producer` scriveremo codice per fornire dati di input alla rete, all'interno dei blocchi `Consumer` scriveremo codice per ottenere i dati di output della rete e verificare che questi corrispondano a quanto atteso.

Timeout di simulazione

Un tipo di problema che possiamo incontrare nelle reti sincronizzate ma non nelle reti combinatorie è la situazione in cui un componente resta in attesa di un segnale che in realtà non verrà mai emesso. Per esempio, questo avviene se la rete da noi realizzata non rispetta il protocollo di handshake.

In questi casi, la simulazione può proseguire *indefinitivamente*.

È quindi necessario prevedere un meccanismo di timeout che interrompe la simulazione quando questa stia durando molto più di quanto è ragionevole aspettarsi. Possiamo realizzare questo utilizzando sempre `fork ... join`.

```

//the following structure is used to wait for expected signals, and fail if too much time passes
fork : f
    begin
        #100000;
        $display("Timeout - waiting for signal failed");
        disable f;
    end
//actual tests start here
begin
    //reset phase
    ...

    fork
        begin : Producer_1
            ...
        end

```

```

begin : Producer_2
    ...
end

begin : Consumer
    ...
end

join
end
join

$finish;

```

Combinando `disable f`, che interrompe ogni esecuzione all'interno del `fork ... join` iniziale, e `$finish` dopo di questo ci assicuriamo che quando il timeout è raggiunto la simulazione viene terminata. Questo ci lascerà una waveform che potremo analizzare per capire da dove sia scaturito il blocco.

Linee di errore

Le simulazioni di reti sincronizzate possono essere molto lunghe (in termini di tempo simulativo, non tempo reale) producendo di conseguenza waveform molto lunghe. Analizzare queste waveform in cerca di errori può essere molto tedioso. Per questo, le testbench d'esame includono solitamente delle *linee di errore*, che evidenziano a colpo d'occhio dov'è che sia avvenuto un problema.

Queste linee sono realizzate nella testbench con una variabile `reg error` inizializzata a 0 e un blocco `always` che risponde ad ogni variazione di `error` per rimetterla a 0 dopo una breve attesa. Questa attesa breve ma non nulla fa sì che basti assegnare 1 ad `error` per ottenere un'impulso sulla linea, facilmente visibile.

```

module testbench();
    ...
    initial begin
        ...
    end

    reg error;
    initial error = 0;
    always @(posedge error) #1
        error = 0;
endmodule

```

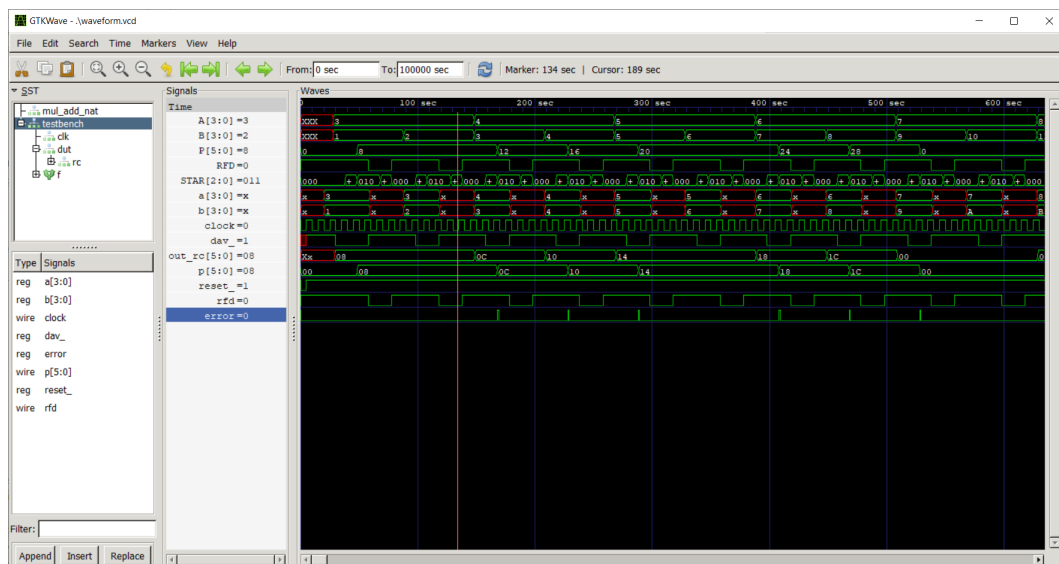
Possiamo quindi scrivere un check dell'output come segue.

```

if(p != t_p) begin
    $display("Expected %d, received %d", t_p, p);
    error = 1;
end

```

In GTKWave, guardando alla linea `error` della testbench questi punti saranno facilmente identificabili nella waveform, come dall'esempio seguente.



Capitolo 8

Esercitazione 4

In questa esercitazione vedremo un esercizio d'esame svolto interamente, sia come descrizione che come sintesi. Il testo dell'esercizio è scaricabile [qui](#).

8.1 Esercizio 4.1: Descrizione

La rete sincronizzata ABC deve ottenere tre valori da altrettanti produttori, sostenendo tre handshake *soc* / *eoc* indipendenti. Tipicamente, questo vuol dire che siamo liberi di eseguire i tre handshake in qualunque ordine, per esempio in serie. In questo caso però, avendo una sola uscita *soc*, siamo costretti a far proseguire i tre handshake secondo gli stessi passi.

Se deriviamo il seguente algoritmo per ABC :

- A riposo: *soc* a 0, tutti gli *eoc* a 1, i dati sono validi
- Inizia ABC mettendo *soc* a 1
- Attende la risposta *eoc* a 0 da tutti i produttori
- Risponde con *soc* a 0
- Attende la conferma *eoc* a 1 da tutti i produttori
- A questo punto, i dati sono validi, e lo resteranno fino al prossimo ciclo

Alla fine di un tale handshake, possiamo svolgere il conto. Le specifiche fanno riferimento a trovare il minimo tra le tensioni v_1, v_2 e v_3 , che sono, in generale, numeri reali, non rappresentabili in un numero finito di bit.

Tramite i convertitori da analogico a digitale si utilizza però una *legge di corrispondenza* che associa, con un certo errore, a ogni valore reale v un il valore x rappresentabile su un numero finito di bit. In questo caso, l'esercizio specifica che queste tensioni sono rappresentate in *binario unipolare* : questo significa che i vari x sono numeri naturali. Questa associazione preserva gli ordinamenti: a $x_1 \leq x_2$ corrisponde $v_1 \leq v_2$. Possiamo quindi tradurre il problema di minimo tra tensioni v in un problema di minimo tra naturali x .

Il risultato dovrà poi essere trasferito al consumatore usando un handshake *dav_* / *rfd*. Per far questo, deriviamo il seguente algoritmo per ABC :

- A riposo: *dav_* a 1, *rfd* a 1
- All'ottenimento di un nuovo dato da trasferire, si imposta l'uscita *min* a tale dato e solo poi si mette *dav_* a 0 per segnalarlo. L'uscita *min* non può oscillare o cambiare
- Si attende la risposta *rfd* a 0
- Si mette *dav_* a 1, ora l'uscita *min* è di nuovo libera di oscillare o cambiare
- Si aspetta la risposta con *rfd* a 1

Possiamo quindi scrivere la seguente descrizione in Verilog, scaricabile [qui](#).

```
module ABC(  
    x1, x2, x3,  
    eoc1, eoc2, eoc3,  
    soc,  
    min, dav_, rfd,
```


La descrizione della rete combinatoria è opzionale

Gli esercizi d'esame chiedono descrizione e sintesi per la rete sincronizzata, ma solo la sintesi per la rete combinatoria. È quindi del tutto opzionale mostrare una descrizione di quest'ultima.

È spesso però utile scrivere prima una descrizione, soprattutto se facilmente interpretabile. Così facendo, in caso di errori nei test, sappiamo di dover debuggare *solo* la descrizione della rete sincronizzata.

8.2 Esercizio 4.1: Sintesi della rete combinatoria

La rete da sintetizzare calcola il minimo di tre numeri naturali. Cominciamo osservando che il minimo di tre numeri è un problema che si può scomporre per associatività: $\min(a, b, c) = \min(\min(a, b), c)$. Questo vuol dire che posso costruire una rete combinatoria per il minimo di tre numeri combinando solo reti per il minimo di due numeri. Lo stesso principio si generalizza per il minimo di n numeri.

Nell'esercizio 2.1 abbiamo visto come costruire una rete combinatoria per il massimo tra due numeri, utilizzando un comparatore. Modificare tale rete per produrre il minimo anziché il massimo è semplice, basta cambiare il multiplexer in fondo. Dato che questo esercizio ci fornisce un sottrattore, possiamo usarlo direttamente anziché costruirlo come nell'esercizio 2.1

```
module MINIMO_2(
    a, b,
    min
);
    input [7:0] a, b;
    output [7:0] min;

    wire b_out;
    diff #( .N(8) ) d(
        .x(a), .y(b), .b_in(1'b0),
        .b_out(b_out)
    );

    assign #1 min = b_out ? a : b;
endmodule
```

Essendo questa una sintesi di MINIMO_2, possiamo usare questo componente per la sintesi di MINIMO_3.

```
module MINIMO_3(
    a, b, c,
    min
);
    input [7:0] a, b, c;
    output [7:0] min;

    // min_3(a, b, c) = min_2( min_2(a, b), c );

    wire [7:0] m_ab_out;
    MINIMO_2 m_ab(
        .a(a), .b(b),
        .min(m_ab_out)
    );

    MINIMO_2 m_abc(
        .a(m_ab_out), .b(c),
        .min(min)
    );
endmodule
```

8.3 Esercizio 4.1: Sintesi di rete sincronizzata

La *descrizione* di una rete sincronizzata è molto utile a capire come si evolve nel tempo: mette in evidenza lo stato interno (registro STAR), i suoi registri operativi, e come questi cambiano nel tempo in base sia al loro contenuto sia agli ingressi della rete.

Quest'uso del Verilog è affine a un diagramma di stato della stessa rete: è certamente più prolisso (per rendere il codice non ambiguo e simulabile) ma è comunque facile seguire l'evoluzione della rete usando una descrizione Verilog allo stesso modo in cui si fa lo stesso con un diagramma di stato.

Una *sintesi*, invece, si occupa di dirci come realizzare tale rete. In teoria, per le descrizioni che scriviamo servirebbero solitamente pochi accorgimenti perché siano direttamente realizzabili in hardware. In pratica, però, questo porterebbe a un uso particolarmente costoso e inefficiente di silicio. Seguiamo quindi un *algoritmo euristico* che ci porta ad hardware molto più efficiente, ragionevolmente vicino a qualcosa che ha senso realizzare.

Non esiste l'ottimo

I ragionamenti fatti in questo corso sull'ottimalità delle sintesi portano solo laddove ci può portare la matematica. Per andare oltre, bisogna entrare nei dettagli pratici, e affrontare trade-off tra processi di produzione, costi in termini di superficie, consumi energetici, esposizione a rischi di progettazione, etc. La progettazione di circuiti integrati è un tema di ingegneria a sé.

Corrispondenza tra sintesi e descrizione

Una sintesi corrisponde a una descrizione: mostra più in dettaglio come realizzare in hardware una rete che si comporta, esternamente, *allo stesso modo* della descrizione. Una sintesi che si comporti in modo diverso dalla descrizione non ha alcun senso.

Il modello di sintesi che ci aspettiamo negli esercizi d'esame è il modello con **parte operativa** e **parte controllo**, dove la parte controllo è implementata secondo il modello a **microindirizzi**.

Riassumendo, questo significa che:

- la rete è divisa in una *parte operativa*, che contiene solo i registri operativi e le relative reti combinatorie, e una *parte controllo*, che contiene il solo registro di stato STAR
- La parte operativa riceve *variabili di comando* dalla parte controllo, che determinano i comportamenti dei registri operativi
- La parte controllo riceve *variabili di condizionamento* dalla parte operativa, che determinano i salti a due vie della parte controllo

Vedremo come seguire l'algoritmo passo passo per ottenere questa sintesi a partire dalla descrizione scritta prima. Da una parte, ciò che si valuta all'esame è solo quanto viene prodotto, mentre il processo seguito per scrivere codice è libera scelta dello studente. Dall'altra, visto che il tempo è necessariamente limitato, avere un processo efficiente permette di esprimere meglio le proprie competenze in tale tempo.

Per questo, vedremo un processo efficiente per fare la sintesi sfruttando l'editor VS Code. Questo sfrutta le scorciatoie dell'editor e l'editing multicaret. Inoltre, avremo vari *checkpoint*, indicati con ∇ , dove il codice intermedio è compilabile e simulabile, permettendo di controllare di avere ancora lo stesso comportamento della descrizione.

8.3.1 Passo 0: ricopiare su un nuovo file

Ricopiamo il nostro file di descrizione, sia *descrizione.v*, su un nuovo file per la sintesi, sia *sintesi.v*. Nel resto delle istruzioni, assumeremo quindi la descrizione "congelata" e modificheremo il file *sintesi.v*.

8.3.2 Passo 1: rendere la descrizione omogenea

Questo processo sfrutta pattern nel codice per eseguire tutte le manipolazioni della descrizione in modo efficiente. Il primo è quindi regolarizzare la descrizione secondo tali pattern.

Per questo, puntiamo a un blocco `always @(posedge clock)` dove:

- Ogni stato contiene esplicitamente il comportamento di ciascun registro, anche se questo è *conservare*.
- L'assegnamento di ciascun registro è su una riga *diversa*.
- Le righe corrispondenti ai vari registri appaiono nello stesso ordine in tutti gli stati.
- Tutti gli stati e assegni seguono la stessa spaziatura: o tutti `REG<=` o tutti `REG<=;`; o tutti `SX: begin` o tutti `SX: begin; end` su una riga a sé.

Per il terzo punto, qualunque ordine va bene, purché sia coerente. Otteniamo quindi il seguente codice ∇ .

```
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: begin
            SOC <= 1;
            DAV_ <= DAV_;
            MIN <= MIN;
            STAR <= (eoc1, eoc2, eoc3 == 3'b000) ? S1 : S0;
        end
        S1: begin
            SOC <= 0;
```

```

        DAV_ <= DAV_;
        MIN <= out_rc;
        STAR <= (eoc1, eoc2, eoc3 == 3'b111) ? S2 : S1;
    end
    S2: begin
        SOC <= SOC;
        DAV_ <= 0;
        MIN <= MIN;
        STAR <= (rfd == 1) ? S2 : S3;
    end
    S3: begin
        SOC <= SOC;
        DAV_ <= 1;
        MIN <= MIN;
        STAR <= (rfd == 0) ? S3 : S0;
    end
endcase
end

```

Corrispondenza tra sintesi e descrizione


Se il comportamento di un registro in un dato stato è assente nella descrizione, questo è implicitamente *conservazione*, e così deve apparire nella sintesi.

Quando si vuole utilizzare un comportamento esplicito per ottimizzazioni, questo va fatto a partire dalla descrizione.

8.3.3 Passo 2: separazione dei blocchi operativi

Separiamo ora i blocchi operativi dei registri - incluso STAR. Per farlo, cominciamo con ricopiare il blocco `always @(posedge clock` tante volte quanti sono i registri. Ciascuna di queste copie verrà poi modificata per lasciare *solo* gli assegnamenti di uno specifico registro, usando l'editing multicaret.

Vediamo per esempio come ottenere il blocco operativo del registro SOC. Selezioniamo la prima occorrenza di `SOC <=`, e premiamo `ctrl + d` tre volte per selezionare tutti gli statement di assegnamento a `SOC <=`. Queste sono ciascuna la prima riga del proprio stato, e vogliamo che diventino *l'unica* riga del proprio stato. Possiamo farlo cancellando le 4 righe sotto, usando i tasti `home` e `end` per muoversi tra righe di dimensioni diverse. Una volta rimasto un solo assegnamento, possiamo rimuovere anche i `begin ... end`.

Facciamo quindi lo stesso per tutti i registri, e separiamo anche gli statement di reset. Otteniamo quindi il seguente codice .

```

always @(reset_ == 0) #1 SOC <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: SOC <= 1;
        S1: SOC <= 0;
        S2: SOC <= SOC;
        S3: SOC <= SOC;
    endcase
end

always @(reset_ == 0) #1 DAV_ <= 1;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: DAV_ <= DAV_;
        S1: DAV_ <= DAV_;
        S2: DAV_ <= 0;
        S3: DAV_ <= 1;
    endcase
end

always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: MIN <= MIN;
        S1: MIN <= out_rc;
        S2: MIN <= MIN;
        S3: MIN <= MIN;
    endcase
end

always @(reset_ == 0) #1 STAR <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: STAR <= (eoc1, eoc2, eoc3 == 3'b000) ? S1 : S0;
        S1: STAR <= (eoc1, eoc2, eoc3 == 3'b111) ? S2 : S1;
        S2: STAR <= (rfd == 1) ? S2 : S3;
    endcase
end

```

```

        S3: STAR <= (rfd == 0) ? S3 : S0;
    endcase
end

```

8.3.4 Passo 3: variabili di comando

In questo passo definiamo per ciascun registro il numero minimo di variabili di comando necessarie per pilotarne il comportamento. Questo numero dipende dal numero di compartimenti *distinti* di tale registro. Nel caso di SOC e DAV_, abbiamo tre comportamenti: 1, 0 e conservazione. Nel caso di MIN ne abbiamo due: campionamento di out_rc o conservazione.

A livello di codice, dovremmo:

- sostituire i casex basati su STAR con casex basati sulle variabili di comando
- aggiungere la definizione delle variabili di comando

Può essere utile fare questo per passaggi successivi, anche se non compilabili, per mantenere il riferimento con la descrizione e riconoscere eventuali errori.

Prendiamo il caso del registro SOC, questo si svolge in tre passi:

- Si aggiungono delle variabili di comando, senza cancellare gli stati
- Si aggiunge la definizione di queste variabili, con riferimento agli stati
- Si cancellano gli stati lasciati prima, ottenendo codice compilabile

Al passo 1, avremo

```

always @(reset_ == 0) #1 SOC <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b1, b0)
        2'b00 S0: SOC <= 1;
        2'b01 S1: SOC <= 0;
        2'b1X S2: SOC <= SOC;
        2'b1X S3: SOC <= SOC;
    endcase
end

```

Al passo 2, scriviamo

```

wire b1, b0;
assign #1 b1, b0 =
    (STAR == S0)? 2'b00 :
    (STAR == S1)? 2'b01 :
    (STAR == S2)? 2'b1X :
    (STAR == S3)? 2'b1X :
    /*default*/

```


Questa notazione a tabella torna molto utile più avanti, quando scriveremo la ROM.

Al passo 3, cancelliamo gli stati dal blocco @ (posedge clock) per riottenere codice valido, ed eliminiamo i doppioni.

```

always @(reset_ == 0) #1 SOC <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b1, b0)
        2'b00: SOC <= 1;
        2'b01: SOC <= 0;
        2'b1X: SOC <= SOC;
    endcase
end

```

Otteniamo quindi il seguente codice .

```

always @(reset_ == 0) #1 SOC <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b1, b0)
        2'b00: SOC <= 1;
        2'b01: SOC <= 0;
        2'b1X: SOC <= SOC;
    endcase
end

always @(reset_ == 0) #1 DAV_ <= 1;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b3, b2)
        2'b00: DAV_ <= DAV_;

```

```

        2'b01: DAV_ <= 0;
        2'b1X: DAV_ <= 1;
    endcase
end

always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b4)
        1'b0: MIN <= MIN;
        1'b1: MIN <= out_rc;
    endcase
end

wire b4, b3, b2, b1, b0;
assign #1 b4, b3, b2, b1, b0 =
    (STAR == S0) ? 5'b00000 :
    (STAR == S1) ? 5'b10001 :
    (STAR == S2) ? 5'b0011X :
    (STAR == S3) ? 5'b01X1X :
    /*default*/    5'bXXXXX ;

```

Ottimizzare la codifica

La procedura mostrata definisce le variabili di comando e le rispettive codifiche *indipendentemente*. È talvolta possibile ridurre il numero complessivo di variabili di comando trovando codifiche che permettano di usare la stessa variabile per più registri. Questa è una ottimizzazione solo se non si aggiunge alcuna variabile di comando a nessun registro.

Non usare la codifica dello stato

L'obiettivo delle variabili di comando è ottimizzare la dimensione dei multiplexer in input a ciascun registro: usare il registro STAR significa usare un multiplexer a tanti ingressi quanti sono gli stati, mentre un registro operativo ha tipicamente molti meno comportamenti distinti.

Usare direttamente la codifica dello stato è un errore grave, a meno che non si *dimostri* che sia proprio la codifica più efficiente. A tale scopo, usare commenti nel codice.

8.3.5 Passo 4: variabili di condizionamento


In questo passo definiamo il numero minimo di variabili necessarie per guidare i cambi di stato della parte controllo. Per far questo, dobbiamo guardare alle condizioni *indipendenti* dei salti a due vie.

```

always @(reset_ == 0) #1 STAR <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: STAR <= (eoc1, eoc2, eoc3 == 3'b000) ? S1 : S0;
        S1: STAR <= (eoc1, eoc2, eoc3 == 3'b111) ? S2 : S1;
        S2: STAR <= (rfd == 1) ? S2 : S3;
        S3: STAR <= (rfd == 0) ? S3 : S0;
    endcase
end

```

In questo caso notiamo che le condizioni di S2 e S3 sono una la negazione dell'altra: non sono indipendenti. Possiamo guidare entrambi i salti con una sola variabile di condizionamento, se invertiamo uno dei due salti. In questo caso, scegliamo di invertire il salto in S3.


Otteniamo quindi il seguente codice .

```

wire c2, c1, c0;
assign #1 c0 = eoc1, eoc2, eoc3 == 3'b000;
assign #1 c1 = eoc1, eoc2, eoc3 == 3'b111;
assign #1 c2 = rfd == 1;

always @(reset_ == 0) #1 STAR <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: STAR <= c0 ? S1 : S0;
        S1: STAR <= c1 ? S2 : S1;
        S2: STAR <= c2 ? S2 : S3;
        S3: STAR <= c2 ? S0 : S3;
    endcase
end

```

Le variabili di condizionamento vanno sintetizzate a livello di bit, sostituendo ai confronti con == le corrispondenti espressioni con porte logiche. In questo caso, otteniamo .

```

assign #1 c0 = ~eoc1 & ~eoc2 & ~eoc3;
assign #1 c1 = eoc1 & eoc2 & eoc3;
assign #1 c2 = rfd;

```

Quando invertire i salti

Viene considerata valida anche una sintesi che usi il numero minimo di variabili di condizionamento ma senza invertire i salti nel Verilog. Per esempio, si può scrivere `S3: STAR <= ~c2 ? S3 : S0;`. Tuttavia non si ha questa libertà nella ROM, dove l'inversione è obbligatoria. Dato che la ROM si fa di solito in fondo è facile dimenticarsene, ed è quindi consigliato di invertire già nel Verilog della parte controllo.


Sintesi a livello di bit

La richiesta di sintesi delle variabili di condizionamento è flessibile per esercizi dove si hanno molti bit (per esempio, un registro COUNT a 16 bit). Nel dubbio, si può chiedere durante l'esame.

8.3.6 Passo 5: separare le parti

Fin a questo punto abbiamo modificato il codice della rete senza separare effettivamente la parte operativa dalla parte controllo. Questo è utile per mantenere la rete simulabile e controllare di non aver introdotto nuovi errori rispetto alla descrizione.

Arrivati a questo punto possiamo separare le parti. Nella parte operativa andranno i registri operativi, le reti combinatorie che ne pilotano gli ingressi, e le reti combinatorie che generano le variabili di condizionamento. A questa parte vengono inoltre collegati gli ingressi e le uscite della rete complessiva. Nella parte controllo, invece, ci sarà solo il registro STAR e la ROM a microindirizzi, di cui scriviamo in Verilog, in particolare, la parte che genera le variabili di controllo.

Otteniamo quindi il seguente codice .

```

module ABC(
    x1, x2, x3,
    eoc1, eoc2, eoc3,
    soc,
    min, dav_, rfd,
    clock, reset_
);
    input [7:0] x1, x2, x3;
    input eoc1, eoc2, eoc3;
    output soc;
    output [7:0] min;
    output dav_;
    input rfd;
    input clock, reset_;

    wire b4, b3, b2, b1, b0;
    wire c2, c1, c0;

    ABC_P0 po(
        x1, x2, x3,
        eoc1, eoc2, eoc3,
        soc,
        min, dav_, rfd,
        b4, b3, b2, b1, b0,
        c2, c1, c0,
        clock, reset_,
    );

    ABC_PC pc(
        b4, b3, b2, b1, b0,
        c2, c1, c0,
        clock, reset_,
    );
endmodule

module ABC_P0(
    x1, x2, x3,
    eoc1, eoc2, eoc3,
    soc,
    min, dav_, rfd,
    b4, b3, b2, b1, b0,
    c2, c1, c0,
    clock, reset_
);
    input [7:0] x1, x2, x3;
    input eoc1, eoc2, eoc3;

```



```

output soc;
output [7:0] min;
output dav_;
input rfd;
input clock, reset_;

reg SOC;
assign soc = SOC;
reg [7:0] MIN;
assign min = MIN;
reg DAV_;
assign dav_ = DAV_;

wire [7:0] out_rc;
MINIMO_3 min_rc(
    .a(x1), .b(x2), .c(x3),
    .min(out_rc)
);

input b4, b3, b2, b1, b0;
output c2, c1, c0;

assign #1 c0 = ~eoc1 & ~eoc2 & ~eoc3;
assign #1 c1 = eoc1 & eoc2 & eoc3;
assign #1 c2 = rfd;

always @(reset_ == 0) #1 SOC <= 0;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b1, b0)
        2'b00: SOC <= 1;
        2'b01: SOC <= 0;
        2'b1X: SOC <= SOC;
    endcase
end

always @(reset_ == 0) #1 DAV_ <= 1;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b3, b2)
        2'b00: DAV_ <= DAV_;
        2'b01: DAV_ <= 0;
        2'b1X: DAV_ <= 1;
    endcase
end

always @(posedge clock) if(reset_ == 1) #3 begin
    casex (b4)
        1'b0: MIN <= MIN;
        1'b1: MIN <= out_rc;
    endcase
end
endmodule

module ABC_PC(
    b4, b3, b2, b1, b0,
    c2, c1, c0,
    clock, reset_,
);
    input clock, reset_;

    output b4, b3, b2, b1, b0;
    input c2, c1, c0;

    reg [1:0] STAR;
    localparam
        S0 = 0,
        S1 = 1,
        S2 = 2,
        S3 = 3;

    assign #1 b4, b3, b2, b1, b0 =
        (STAR == S0) ? 5'b000000 :
        (STAR == S1) ? 5'b100001 :
        (STAR == S2) ? 5'b00111X :
        (STAR == S3) ? 5'b01X1X :
        /*default*/ 5'bXXXXXX ;

    always @(reset_ == 0) #1 STAR <= 0;
    always @(posedge clock) if(reset_ == 1) #3 begin

```

```

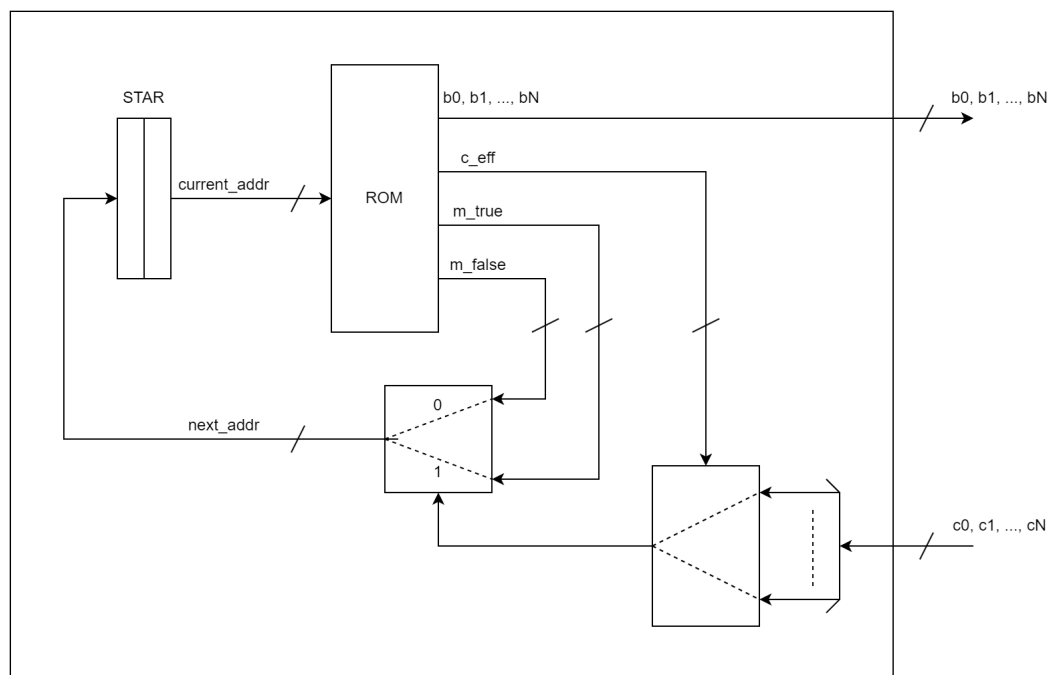
    case (STAR)
      S0: STAR <= c0 ? S1 : S0;
      S1: STAR <= c1 ? S2 : S1;
      S2: STAR <= c2 ? S2 : S3;
      S3: STAR <= c2 ? S0 : S3;
    endcase
  end
endmodule

// sintesi di rete combinatoria omessa

```

8.3.7 Passo 6: la ROM

Secondo il modello a microindirizzi, la parte controllo è implementata come una ROM che usa il registro STAR come *microindirizzo*, e fornisce in uscita le variabili di controllo per la parte operativa così come *c_eff*, *m_true* e *m_false*. Questi ultimi tre elementi, insieme alle variabili di condizionamento in ingresso dalla parte operativa, sono utilizzati con dei multiplexer in cascata per determinare lo stato successivo della rete. Lo schema circuitale è il seguente.



Fare la sintesi direttamente come ROM è possibile, ma è ben più tedioso e difficile da debuggare. Per questo manteniamo il linguaggio di trasferimento tra registri per indicare in Verilog il comportamento di STAR, e scriviamo invece come commento nel codice una tabella che mostri il contenuto della ROM.

Per evitare ogni ambiguità, si comincia specificando codifiche binarie per gli stati e i *nomi* delle variabili di comando. Dopodiché, si può riempire la tabella con quanto scritto in Verilog.

```

/*
  S0 = 00, S1 = 01, S2 = 10, S3 = 11
  c0 = 00, c1 = 01, c2 = 1X

  M-addr | b4, b3, b2, b1, b0 | c_eff | M-addr-T | M-addr-F
  -----
  00     | 00000             | 00   | 01         | 00
  01     | 01100             | 01   | 10         | 01
  10     | 1X001             | 1X   | 10         | 11
  11     | 1X01X             | 1X   | 00         | 11
*/

```

A questo punto la sintesi è completa. È scaricabile [qui](#).

Menzioniamo a parte il caso dei salti incondizionati, del tipo del tipo `STAR <= S1`. In questo caso, va scritta la ROM in modo che *next_address* sia *S1* per qualunque valore delle variabili di condizionamento. Questo si può ottenere mettendo sia *m_true* che *m_false* a *S1*, lasciando non-specificato *c_eff*. In tabella, questo si traduce in

```

/*
  ... | c_eff | M-addr-T | M-addr-F
  -----
  ... | X   | 01       | 01
*/

```

Formattazione della tabella

Non essendo parte del codice, non c'è una sintassi o formattazione precisa da seguire per la tabella, purché sia non-ambigua quando letta dai docenti. Questo include le varie alternative come `m-true`, `m-addr-true`, `m-T` etc., o usare `?` al posto di `x`.

Valori non specificati

La tabella della ROM è difatto una tabella di verità per una rete combinatoria, che è quindi sintetizzabile utilizzando i metodi visti come le mappe di Karnaugh.

I valori non specificati per variabili di comando e condizionamento sono quindi utili per permettere ottimizzazioni in tale sintesi.

Salti incondizionati, non indeterminati

È purtroppo frequente vedere ROM dove uno o entrambi gli indirizzi `m_true` o `m_false` sono non specificati, come nell'esempio seguente

```
/*
... | c_eff | M-addr-T | M-addr-F
-----
... | X    | 01      | X          Errore!
*/
```

Una ROM del genere è priva di senso, perché determina una rete che salta *a uno stato a caso*.

Capitolo 9

Esercitazione 5

9.1 Esercizio 5.1: esame 2023-07-18

[Qui](#) testo e soluzione.

Provare da sé

Provare a svolgere da sé l'esercizio, prima di guardare la soluzione o andare oltre per la discussione.

Esercizi con interfacce parallele

Il corretto pilotaggio di interfacce parallele richiede inevitabilmente più stati di quanto è solito usare per esercizi, per esempio, con handshake `dav_rfd` o `soc/eoc`.

Per questo, sono tipicamente gli esercizi che hanno più tempo a disposizione in sede d'esame.

In questo esercizio si lavora con interfacce parallele connesse a un bus, una interfaccia di ingresso con handshake (da cui ottenere dati in ingresso) e una interfaccia di ingresso e uscita senza handshake (a cui scrivere il risultato del calcolo). Il calcolo da svolgere è semplice: bisogna moltiplicare l'ingresso per 5, esprimendo il risultato su 16 bit. È *opzionale* sintetizzarlo con rete combinatoria.

La parte critica dell'esercizio è il corretto pilotaggio del bus, evitando corse critiche e usando porte tri-state per evitare situazioni di corto circuito, e delle interfacce.

Per quanto riguarda le operazioni sul bus, ricordiamo che le interfacce si attivano alla ricezione di segnali `ior_` e `iow_`. Questi segnali arrivano a tutte le interfacce sul bus, ma è solo quella *selezionata* tramite `addr` che si attiva, o leggendo `data` o assegnandogli un valore. È quindi critico che i fili di uscita `addr` e `data` siano stabili prima di portare `ior_` o `iow_` a 0.

Per `data`, questo si traduce, per scritture, nel valore assegnato stabile (per esempio, con registro `DATA`) e la porta tri-state abilitata (solitamente, `DIR = 1`); per le letture invece si disabilita la porta tri-state (solitamente, `DIR = 0`) per lasciare che sia l'interfaccia a assegnargli un valore. Ricordiamo che il problema qui è di tipo elettrico: assegnare un valore logico a un filo equivale a imporre una tensione, e se più dispositivi assegnano tensioni diverse sullo stesso filo la differenza di potenziale porta a un disastroso corto circuito.

Questo schema di pilotaggio va ripetuto più volte per accedere ai diversi registri. L'interfaccia parallela di ingresso con handshake richiede che si legga il suo registro `RSR`, ed in particolare il suo flag `FI`, in attesa che `FI = 1` segnali la presenza di un dato da poter leggere. La lettura verrà fatta sul registro `RBR`. Per l'interfaccia parallela di ingresso-uscita senza handshake, dovremo invece fare due scritture sul registro `TBR`.

9.2 Esercizio 5.1: esame 2024-01-26

[Qui](#) testo e soluzione.

Provare da sé

Provare a svolgere da sé l'esercizio, prima di guardare la soluzione o andare oltre per la discussione.

Il testo di questo esercizio è pensato per apparire fuori dalla norma a un occhio poco preparato, ma si rivela molto semplice con le dovute osservazioni.

Tralasciando le curiosità sulla *congettura di Collatz*, ciò che ci interessa è osservare che il calcolo di n_{i+1} a partire da n_i è di tipo combinatorio. Ciò è anche suggerito dal testo, che ci chiede di sintetizzare proprio questo.

Ciò che non è combinatorio è invece il calcolo di n_{i+1} a partire da n_0 : questa è infatti una operazione *iterativa*, che implica una struttura ad anello che svolge più passaggi. Come dovremmo ben sapere, tali anelli non possono essere

reti combinatorie, e vanno invece implementate con reti sincronizzate che avanzano a passaggi discreti guidati dal segnale del clock. Anche questa osservazione è suggerita dal testo, visto che il modo più immediato per ottenere k è *contare* le iterazioni necessarie per arrivare ad 1.

La struttura chiave quindi è la seguente: abbiamo un registro N che conterrà l'attuale n_i , inizializzato con n_0 . Al posedge del clock, campionando l'uscita della rete combinatoria CALCOLO_ITERAZIONE, riceve il nuovo valore n_{i+1} . Contemporaneamente, un registro K , inizializzato a 0, conta con $K \leq K + 1$; quanti posedge sono necessari perché N arrivi ad 1. Questo è garantito dall'uso di un cambio di stato che interrompe il conteggio quando la condizione è raggiunta, fatte le solite osservazioni per il corretto conteggio di cicli di clock. Questo ciclo può essere espresso come nel seguente pseudo-codice:

```
...
CALCOLO_ITERAZIONE ci(
    .n_curr(N),
    .n_next(n_next)
);
...
always @(posedge clock) if(reset_ == 1) #3 begin
    casex(STAR)
        ...
        S_init: begin
            N <= n_0;
            K <= 0;
            ...
        end
        S_loop: begin
            N <= n_next;
            K <= K + 1;
            STAR <= (n_next == 1) ? S_after : S_loop;
        end
        S_after: ...
    endcase
end
...
```

Una volta chiarito questo processo, il resto dell'esercizio è molto semplice. Va dimensionato N e la relativa rete combinatoria: il testo ci indica che il massimo raggiungibile è inferiore a 'h4000, implicando che 14 bit sono sufficienti. Va poi sintetizzata la rete combinatoria, che altro non è che un multiplexer, guidato dal bit meno significativo, i cui due ingressi sono uno shift a destra e un moltiplicatore con $y = 3$ e $c = 1$. Infine, la rete sincronizzata campiona n_0 e invia k tramite un singolo handshake soc/eoc.

Capitolo 10

Esercitazione 6

10.1 Esercizio 6.1: esame 2024-07-16

[Qui](#) testo e soluzione.

Provare da sé

Provare a svolgere da sé l'esercizio, prima di guardare la soluzione o andare oltre per la discussione.

Esercizi *senza* sintesi di reti combinatorie

Non tutti gli esercizi includono la sintesi di reti combinatorie, così come non tutti i pretest, esercizi di Assembler o domande all'orale coprono un dato argomento del corso.

Nel complesso, ogni esame ambisce a coprire tutti gli argomenti del corso.

L'aspetto particolare di questo esercizio è la richiesta di utilizzare microsottoprogrammi. Questo significa codificare una serie di stati a cui si può saltare da diversi altri stati, a cui poi si intende tornare, proprio come i sottoprogrammi nel software. Al posto di `call` e `ret`, però, si usa il registro MJR (Multiway Jump Register).

La struttura per la descrizione è la seguente. Siano `S0... SN` gli stati della sequenza principale, e siano `Smp0... SmpN` gli stati del microsottoprogramma. Uno stato della sequenza principale può saltare a un microsottoprogramma così

```
S0: begin
    ...
    MJR <= S1;
    STAR <= Smp0;
end
```

Al termine del microsottoprogramma si salterà poi alla sequenza principale così

```
SmpN: begin
    ...
    STAR <= MJR;
end
```

All'interno del processore sEP8 visto nel corso si fa un uso massiccio di microsottoprogrammi per separare le varie operazioni del processore in sequenze generiche, atomiche e ben riconoscibili, come la fase di esecuzione di una specifica istruzione, l'accesso a una locazione di memoria, l'accesso a una interfaccia di I/O, etc.

Nella soluzione di questo esercizio, si usa un microsottoprogramma che esegue l'handshake e preleva un dato dal convertitore A/D. Il flusso principale non fa che saltare a questo microsottoprogramma 3 volte per prelevare i dati di cui ha bisogno. In S3, il calcolo è solo descritto.

```
...
always @(posedge clock) if(reset_ == 1) #3 begin
    casex (STAR)
        S0: begin
            OUT <= 0;
            MJR <= S1;
            STAR <= S_read0;
        end
        S1: begin
            X2 <= X0;
            MJR <= S2;
            STAR <= S_read0;
        end
        S2: begin
            X1 <= X0;
            MJR <= S3;
        end
    endcase
end
```

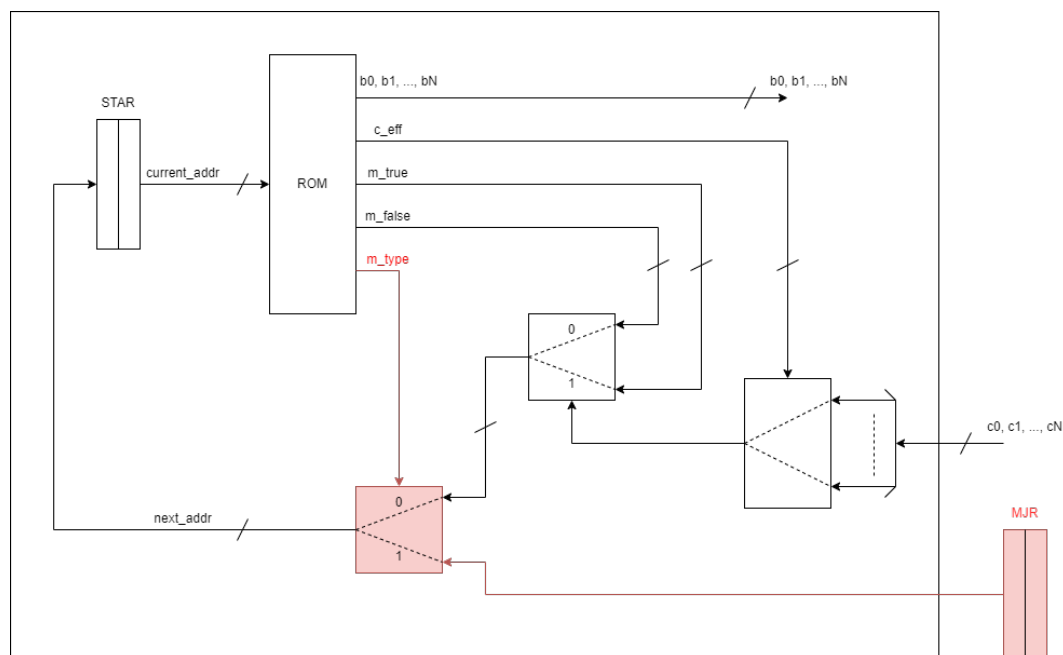
```

STAR <= S_read0;
end
S3: begin
    OUT <= ((X0 + X1 + X2) >= 164) ? 1 : 0;
    STAR <= S0;
end

// microsottoprogramma per l'acquisizione di un campione
// il dato acquisito viene lasciato in X0
S_read0: begin
    SOC <= 1;
    STAR <= (eoc == 1'b0) ? S_read1 : S_read0;
end
S_read1: begin
    SOC <= 0;
    STAR <= (eoc == 1'b1) ? S_read2 : S_read1;
end
S_read2: begin
    X0 <= x;
    STAR <= MJR;
end
endcase
end

```

Un altro aspetto critico è come sintetizzare una rete del genere, cioè come si implementa effettivamente dell'hardware che si comporta in questo modo. L'aspetto chiave è il fatto che quando non si usano microsottoprogrammi, i valori assegnati a STAR sono sempre delle costanti, che come abbiamo visto possono essere sintetizzate usando una ROM. I salti che usano MJR invece no, perché, per l'appunto, usano un registro da cui viene letto il prossimo stato. Va quindi utilizzata una architettura *diversa*. Una di quelle viste nel corso è così schematizzata.



In questa architettura notiamo che si aggiunge un nuovo filo in uscita alla ROM per distinguere i salti (in)condizionati dai salti che leggono da MJR. Possiamo quindi sintetizzare la parte controllo di questo esercizio con una ROM come la seguente.

```
// Per utilizzare il registro MJR, va esteso il modello di sintesi della parte controllo e la relativa ROM, in mod
```

```
// Per distinguere questi salti da quelli guidati da MJR, introduciamo un altro multiplexer guidato dal campo m-ty
// Questo varrà 0 per i salti incondizionati o a due vie e 1 per i salti guidati da MJR.
```

```
// Per i salti incondizionati o a due vie, si utilizzano i campi m-addr T ed m-addr F della ROM, e un multiplexer
// Dato che, in questo caso, abbiamo una sola variabile di condizionamento, non c'è bisogno di distinguerle tramite
```

m-addr	m-code	m-addr T	m-addr F	m-type
001 (S1)	000000101	100 (S_read0)	100 (S_read0)	0
000 (S0)	100000010	100 (S_read0)	100 (S_read0)	0
010 (S2)	010000011	100 (S_read0)	100 (S_read0)	0
011 (S3)	000001100	000 (S0)	000 (S0)	0

100 (S_read0)	0001X0000	100 (S_read0)	101 (S_read1)	0
101 (S_read1)	000010000	110 (S_read2)	101 (S_read1)	0
110 (S_read2)	001000000	XXX	XXX	1

*/

Niente salti condizionati con MJR

L'architettura presentata permette solo

- Salti incondizionati a stato *costante*, del tipo $\text{STAR} \leq S_0$; da sintetizzare con $m\text{-type} = 0$, $c_{\text{eff}} = X$, $m\text{-true} = m\text{-false} = S_0$.
- Salti condizionati a stati *costanti*, del tipo $\text{STAR} \leq c_1 ? S_0 : S_1$; da sintetizzare con $m\text{-type} = 0$, $c_{\text{eff}} = c_1$, $m\text{-true} = S_0$, $m\text{-false} = S_1$.
- Salti incondizionati a MJR, del tipo $\text{STAR} \leq \text{MJR}$; da sintetizzare con $m\text{-type} = 1$, $c_{\text{eff}} = X$, $m\text{-true} = X$, $m\text{-false} = X$.

Non sono sintetizzabili invece salti del tipo $\text{STAR} \leq c_1 ? \text{MJR} : S_1$. Per far questo ci vorrebbe un'altra architettura, *diversa* da quelle viste in questo corso.

10.2 Esercizio 6.2: esame 2024-09-10

[Qui](#) testo e soluzione.

Provare da sé

Provare a svolgere da sé l'esercizio, prima di guardare la soluzione o andare oltre per la discussione.

Il primo aspetto interessante di questo esercizio è la ricezione di byte tramite linea seriale, la cui specifica è data dal testo. Dalla specifica, quando un valore viene trasmesso si imposta la linea su 0, la lunghezza dell'intervallo in cui la linea è a 0 ci indica quale bit è stato trasmesso. Ci sono due range dati: $[2, 7]$ periodi di clock per un bit 1, $[11, 15]$ per un bit 0.

Assenza di errori

In esercizi d'esame come questo, si *assume* che non ci siano errori di alcun tipo nel trasmettitore o sulla linea. Non ci saranno quindi periodi di lunghezze diversa dagli intervalli indicati. La distanza tra 7 e 11 permette di escludere ambiguità nella misurazione della lunghezza degli intervalli.

Abbiamo quindi bisogno di due ingredienti. Il primo è un registro su cui campioniamo in shift continuo. Dato che l'esercizio indica i bit inviati a partire dal più significativo, si può campionare un byte eseguendo 8 volte $\text{BYTE} \leftarrow \text{nuovo_bit}, \text{BYTE}[7:1]$; Il secondo ingrediente è aspettare che rx_d vada a 0, e poi *contare* per quanti periodi di clock rimane a 0, per esempio con un registro COUNT . Possiamo quindi calcolare $\text{nuovo_bit} = (\text{COUNT} \leq 7)$, che è ottimizzabile (ma non è indispensabile) notando che è equivalente a $\text{nuovo_bit} = \sim \text{COUNT}[3]$.

Il secondo aspetto interessante dell'esercizio riguarda il calcolo combinatorio da eseguire con i byte ricevuti. L'esercizio chiede di interpretarli come numeri *interi* in complemento alla radice x_0, \dots, x_n , calcolandone di volta in volta la somma s_i che è posta in uscita. Quando tale somma sarà non più rappresentabile, si torna alle condizioni al reset (ossia il prossimo campione sarà x_0).

Per far questo utilizziamo iterativamente un sommatore, a cui colleghiamo in ingresso i registri s (inizializzato a 0) e BYTE . Sia s l'uscita di tale sommatore. Ognivolta che un nuovo campione è stato ricevuto completamente, campioniamo la nuova somma con $s \leftarrow s$.

Dato che si parla di numeri interi in complemento alla radice, come criterio di rappresentabilità dobbiamo usare il filo di overflow ow di questo sommatore. L'uscita c_{out} è invece irrilevante, sarebbe di interesse solo se questi fossero numeri *naturali*.

Parte II

Verilog - Documentazione

Capitolo 11

Introduzione

Questa documentazione è organizzata per fornire riferimenti rapidi per ciascun contesto d'uso del Verilog. Nel far questo, prendiamo in considerazione il fatto che in Verilog la stessa sintassi può avere usi diversi in contesti diversi: per esempio, si parlerà in modo diverso di `reg` per testbench simulative rispetto a come se ne parla per reti sincronizzate.

Le definizioni “vere” di queste sintassi sono più astratte di quanto presentato qui, proprio per accomodare usi diversi. Un esempio di documentazione più completa ma non orientata agli usi di questo corso si trova su www.chipverif.com.

Capitolo 12

Operatori

12.1 Valori letterali (*literal values*)

In ogni linguaggio, i *literal values* sono quelle parti del codice che rappresentano valori costanti. Per ovvi motivi, in Verilog questi sono principalmente stringhe di bit.

La definizione (completa) di un valore letterale è data da

1. dimensione in bit
1. formato di rappresentazione
1. valore

Per esempio, `4'b0100` indica un valore di 4 bit, espressi in notazione *binaria*, il cui valore in binario è `0100`. Le altre notazioni che useremo sono `d` per decimale (`4'd7` corrisponde al binario `0111`) e `h` per esadecimale (`8'had` corrisponde al binario `10101101`).

12.1.1 Estensione e troncamento

Verilog automaticamente estende e tronca i letterali la cui parte valore è sopra o sottospecificata rispetto al numero di bit. Per esempio, `4'b0` viene automaticamente esteso a `4'b0000`, mentre `6'had` viene automaticamente troncato a `6'b101101`.

12.2 Operatori aritmetici

Il Verilog supporta molti degli operatori comuni, che possiamo usare in espressioni combinatorie: `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `==`.

Prestare attenzione, però, ai dimensionamenti in bit degli operandi e a come Verilog li estende per eseguire le operazioni.

12.3 Operatori logici e *bitwise*

Verilog supporta i classici operatori logici `&&`, `||` e `!`. Questi lavorano su valori booleani (`0` è `false`, diverso da zero è `true`), e producono un solo bit come risultato.

Operatore logico	Tipo di operazione
<code>&&</code>	and
<code> </code>	or
<code>!</code>	not

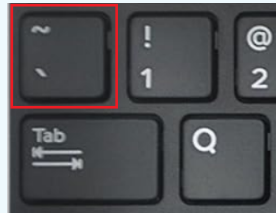
Questi vanno distinti dagli operatori *bitwise* (in italiano *bit a bit*), che lavorano invece per un bit alla volta (e per bit corrispondenti) producendo un risultato delle stesse dimensioni degli operandi.

Operatore bitwise	Tipo di operazione
$\&$	and
$\sim\&$	nand
$ $	or
$\sim $	nor
\wedge	xor
$\sim\wedge$	xnor
\sim	not

Per indicare porte logiche, utilizzare gli operatori bitwise.

Come scrivere la tilde ~

Nel layout di tastiera QWERTY internazionale, la tilde ha un tasto dedicato, a sinistra dell'1.



Nel layout di tastiera QWERTY italiano, invece, la tilde non è presente. Ci sono 3 opzioni:

3. passare al layout QWERTY internazionale
3. imparare scorciatoie alternative, che dipendono dal sistema operativo
3. usare scripting come AutoHotkey per personalizzare il layout

L'opzione 1 richiede di imparare un layout diverso, ma è consigliabile per tutti gli usi di programmazione dato che risolve altri problemi come il backtick ` e rende più semplici da scrivere caratteri come `[]` e `;`. [Qui](#) le istruzioni per cambiare layout su Windows.

L'opzione 2 varia da sistema a sistema. Su Windows, la combinazione di tasti è `alt + 126`, facendo attenzione a digitare il numero usando il tastierino numerico e non la riga dei numeri.

L'opzione 3 non è utilizzabile all'esame. Per uso personale, vedere [qui](#).

12.3.1 Reduction operators

I *reduction operators* applicano un'operazione tra tutti i bit di un elemento di più bit, producendo un risultato su un solo bit. Sia per esempio x di valore `4'b0100`, allora la sua riduzione `and x`, equivalente a `x[3] & x[2] & x[1] & x[0]`, varrà `1'b0`; mentre la sua riduzione `or x`, varrà `1'b1`. Le riduzioni possono rendere alcune espressioni combinatorie più semplici da scrivere.

Operatore	Tipo di riduzione
$\&$	and
$\sim\&$	nand
$ $	or
$\sim $	nor
\wedge	xor
$\sim\wedge$	xnor

12.4 Operatore di selezione [...]

Quando si dichiara un elemento, come un wire, si utilizza la notazione `[N:0]` per indicare l'elemento ha $N+1$ bit, indicizzati da 0 a N . Per esempio, per dichiarare un filo da 8 bit, scriveremo

```
wire [7:0] x;
```

Possiamo poi utilizzare l'operatore per selezionare uno o più bit di un tale componente. Per esempio, possiamo scrivere `x[2]`, che seleziona il bit di posizione 2 (*bit-select*), e `x[6:3]`, che seleziona i quattro bit dalla posizione 6 alla posizione 3 (*part-select*).

12.5 Operatore di concatenazione {...}

L'operatore di concatenazione viene utilizzato per combinare due o più espressioni, vettori, o bit in un'unica entità.

```
input [3:0] a, b;
wire [7:0] ab;
assign ab = a, b;
```

L'operatore può anche essere usato a sinistra di un assegnamento.

```
input [7:0] x;
wire [3:0] xh, xl;
assign xh, xl = x;
```

Maneggiare fili non ha nessun costo

Questo operatore corrisponde, circuitualmente, al semplice raggruppare o separare dei fili. Non è un'operazione combinatoria, e per questo non consuma tempo. È per questo che negli esempi sopra gli assign non hanno alcun ritardo #T.

12.5.1 Operatore di replicazione N{...}

L'operatore di replicazione semplifica il tipico caso d'uso di ripetere un bit o un gruppo di bit *N* volte. Si può utilizzare solo all'interno di un concatenamento che sia a *destra* di un assegnamento e con *N* costante. È equivalente a scrivere *N* volte ciò che si vuole ripetere.

```
input [3:0] x;
wire [15:0] x_repeated_4_times;
assign x_repeated_4_times = 4x; // equivalente a x, x, x, x
```

Il suo uso più comune è l'estensione di segno di interi, mostrato più avanti.

12.6 Operazioni comuni

12.6.1 Estensione di segno

Quando si estende un numero su più bit bisogna considerare se il numero è un naturale o un intero. Per estendere un naturale, basta aggiungere degli zeri.

```
wire [7:0] x_8;
wire [11:0] x_12;
assign x_12 = 4'h0, x_8;
```

Per estendere un intero, dobbiamo invece replicare il bit più significativo.

```
wire [7:0] x_8;
wire [11:0] x_12;
assign x_12 = 4x_8[7], x_8;
```

12.6.2 Shift a destra e sinistra

Per fare shift a destra e sinistra ci basta utilizzare gli operatori di selezione e concatenamento. Lo shift a sinistra è lo stesso per numeri naturali e interi, posto che non ci sia overflow.

```
input [7:0] x;
wire [7:0] x_mul_4;
assign x_mul_4 = x[5:0], 2'b0;
```

Lo shift a destra richiede invece di considerare il segno, se stiamo lavorando con interi.

```
input [7:0] x; // rappresenta un numero naturale
wire [7:0] x_div_4;
assign x_div_4 = 2'b0, x[7:2];
```

```
input [7:0] x; // rappresenta un numero intero
wire [7:0] x_div_4;
assign x_div_4 = 2x[7], x[7:2];
```

Capitolo 13

Sintassi per reti combinatorie

Una rete combinatoria si esprime come un `module` composto solo da `wire`, espressioni combinatorie e componenti che sono a loro volta reti combinatorie.

13.1 module

Il blocco `module ... endmodule` definisce un *tipo* di componente, che può poi essere istanziato in altri componenti. La dichiarazione di un `module` include il suo nome e la lista delle sue porte.

```
module nome_rete ( porta1, porta2, ... );  
    ...  
endmodule
```

13.1.1 input e output

Per ciascuna porta di un `module`, dichiariamo se è di `input` o `output`, e di quanti bit è composta. Se non specificata, la dimensione default è 1. La dichiarazione di porte con le stesse caratteristiche si può fare nella stessa riga.

Le porte `input` sono dei `wire` il cui valore va assegnato *al di fuori* di questa rete.

Le porte `output` sono dei `wire` il cui valore va assegnato *all'interno* di questa rete.

```
module nome_rete ( porta1, porta2, porta3, porta4 );  
    input [3:0] porta1, porta2;  
    output [3:0] porta3;  
    output porta4;  
    ...  
endmodule
```

inout

Non usiamo porte `inout` nelle reti combinatorie.

13.2 wire

Un `wire` è un filo che trasporta un valore logico. Se non specificata, la dimensione default è 1. La dichiarazione di `wire` con le stesse caratteristiche si può fare nella stessa riga.

```
wire [3:0] w1, w2;  
wire w3, w4, w5;
```

Con uno statement `assign` possiamo associare al `wire` una *espressione combinatoria*: il `wire` assumerà continuamente il valore dell'espressione, rispondendo ai cambiamenti dei suoi operandi. Lo statement `assign` può includere un fattore di ritardo, `#T`, per indicare che il valore del filo segue il valore dell'espressione con ritardo di `T` unità.

```
assign #1 w5 = w3 & w4;
```

Un `wire` può essere associato a una porta di un `module`, come mostrato nella sezione successiva.

13.3 Usare un module in un altro module

Una volta definito un module, possiamo istanziare componenti di questo *tipo* in un altro module.

```
nome_module nome_istanza (
    .porta1(...), .porta2(...), ...
);
```

Questo corrisponde, circuitualmente, al prendere un componente fisico di tipo `nome_module`, chiamato `nome_istanza` per distinguerlo dagli altri, e posizionarlo nella nostra rete collegandone i vari piedini con altri elementi.

All'interno degli statement `.porta(...)` specifichiamo quale porta, espressione o wire del module corrente va collegato alla porta del module istanziato.

Insieme agli statement `assign` e l'uso di `wire`, questo ci permette di comporre reti combinatorie su diversi livelli di complessità e con poca duplicazione del codice.

Come esempio, costruiamo un `and` a 1 ingresso e lo usiamo per comporre un `and` a 3 ingressi.

```
module and(a, b, z);
    input a, b;
    output z;

    assign #1 z = a & b;
endmodule
```

```
module and2(a, b, c, z);
    input a, b, c;
    output z;

    wire z1;
    and a1(
        .a(a), .b(b),
        .z(z1)
    );

    and a2(
        .a(c), .b(z1),
        .z(z)
    );
endmodule
```

13.4 Tabelle di verità

Talvolta il modo più immediato per esprimere una rete combinatoria è tramite la sua tabella di verità. È anche noto che data una tabella di verità possiamo ottenere una sintesi della rete combinatoria, utilizzando metodi come le mappe di Karnaugh.

In Verilog, il modo più immediato di esprimere una tabella di verità è utilizzando una catena di operatori ternari.

```
module and(x, y, z);
    input x, y;
    output z;
    assign #1 z =
        (x,y == 2'b00) ? 1'b0 :
        (x,y == 2'b01) ? 1'b0 :
        (x,y == 2'b10) ? 1'b0 :
        /*x,y == 2'b11*/ 1'b1;
```

Un'alternativa è l'uso di `function` e `casex`.

```
module and(x, y, z);
    input x, y;
    output z;
    assign #1 z = tabella_verita(a, b);

    function tabella_verita;
        input [1:0] ab;
        casex(ab)
            2'b00: tabella_verita = 1'b0;
            2'b01: tabella_verita = 1'b0;
            2'b10: tabella_verita = 1'b0;
            2'b11: tabella_verita = 1'b1;
        endcase
    endfunction
endmodule
```

Per indicare tabelle di verità con più di un bit in uscita si scrive, per esempio, `function [1:0] tabella_verita;`. Nel `casex` si può utilizzare anche un caso default, scrivendo come ultimo caso default: `tabella_verita = ...;`.

Attenzione all'uso delle function

Le *function* sono blocchi di *codice da eseguire*, parti del *behavioral modelling* di Verilog. Il simulatore ne svolge i passaggi come un programma, senza consumare tempo e senza alcun corrispettivo hardware previsto. È per questo, per esempio, che dobbiamo specificare noi il tempo consumato nello statement *assign*. L'uso mostrato qui delle *function* è l'unico ammesso per una *sintesi* di reti combinatorie. In presenza di ogni altra elaborazione algoritmica, di cui non sia evidente il corrispettivo hardware, sarà invece considerata una *descrizione* di rete combinatoria.

13.5 Multiplexer

I multiplexer sono da considerarsi noti e sintetizzabili, e si possono esprimere con uno o più operatori ternari ?.

Operatore ternario

La sintassi è della forma *cond ? v_t : v_f*, dove *cond* è un predicato (espressione *true* o *false*) mentre *v_t* e *v_f* sono espressioni dello stesso tipo.

L'espressione ha valore *v_t* se il predicato *cond* è *true*, *v_f* altrimenti.

Per un multiplexer con selettore a 1 bit, basterà un solo ?.

```
input sel;
assign #1 multiplexer = sel ? x0 : x1;
```

Per un selettore a più bit si dovranno usare in serie per gestire più casi

```
input [1:0] sel;
assign #1 multiplexer =
  (sel == 2'b00) ? x0 :
  (sel == 2'b01) ? x1 :
  (sel == 2'b10) ? x2 :
  /*sel == 2'b11*/ x3 ;
```

Differenza tra multiplexer e tabella di verità

La sintassi qui mostrata sembra identica a quella mostrata poco prima per le tabelle di verità. Sono quindi la stessa cosa? **No**.

Dato uno specifico ingresso, una rete combinatoria avrà come uscita sempre il valore corrispondente nella tabella di verità, che è specifico e costante (a meno di *non specificati*). Per un multiplexer, invece, l'uscita è il valore di uno degli ingressi, che è libero di mutare. Le realizzazioni circuitali di questi componenti sono completamente diverse.

Per la sintassi Verilog, invece, la differenza è da poco (prendere un *right hand side* da una variabile o da un letterale). Di nuovo, è importante stare attenti a *cosa si sta facendo* quando si scrive codice Verilog.

13.6 Reti parametrizzate

In un *module* si possono definire parametri per generalizzare la rete. In particolare, questo è utilizzato in *reti_standard.v* per fornire reti il cui dimensionamento va specificato da chi le utilizza.

Per esempio, vediamo come è definita una rete di somma a N bit.

```
module add(
  x, y, c_in,
  s, c_out, ow
);
  parameter N = 2;

  input [N-1:0] x, y;
  input c_in;

  output [N-1:0] s;
  output c_out, ow;

  assign #1 c_out, s = x + y + c_in;
  assign #1 ow = (x[N-1] == y[N-1]) && (x[N-1] != s[N-1]);
endmodule
```

Con *N = 2* viene impostato il valore di default del parametro. Quando istanziamo la rete altrove, possiamo modificare questo parametro, per esempio per ottenere un sommatore a 8 bit.


```
add #( .N(8) ) a (
    ...
);
```

Un module può avere più di un parametro, che possono essere impostati indipendentemente.

```
nome_modulo #( .nome_parametro1(v1), .nome_parametro2(v2)... ) nome_istanza (
    ...
);
```

Immutabilità dei parametri

I parametri determinano la quantità di hardware, che non può essere cambiata mentre la rete è in uso. I valori associati devono essere costanti.

Parametrizzazione e sintesi di reti combinatorie

La parametrizzazione è facilmente applicabile a *descrizioni* di reti combinatorie dove si usano espressioni combinatorie che il simulatore è facilmente in grado di adattare a diverse quantità di bit.

È molto più complicato applicarla a *sintesi* di reti combinatorie, dato che non si possono instanziare componenti in modo parametrico, per esempio N full adder da 1 bit per sintetizzare un full adder a N bit.

Capitolo 14

Sintassi per reti sincronizzate

Una rete sincronizzata si esprime come un `module` contenente registri, che sono espressi con `reg` il cui valore è inizializzato in risposta a `reset_` ed aggiornato in risposta a fronti positivi del `clock`.

Gran parte della sintassi già vista per le reti combinatorie rimane valida anche qui, e dunque non la ripetiamo. Ci focalizziamo invece su come esprimere registri usando `reg`.

14.1 Istanziamento

Un registro si istanzia con statement simili a quelli per `wire` :

```
reg [3:0] R1, R2;  
reg R3, R4, R5;
```

Nomi in maiuscole e minuscolo

Verilog è *case sensitive*, cioè distingue come diversi nomi che differiscono solo per la capitalizzazione, come `out` e `OUT`.

Nel corso, utilizziamo questa feature per distinguere a colpo d'occhio `reg` e `wire`, utilizzando lettere maiuscole per i primi e minuscole per i secondi. Questo è particolarmente utile quando si hanno registri a sostegno di un `wire`, tipicamente un'uscita della rete o l'ingresso di un `module` interno.

Seguire questa convenzione non è obbligatorio, ma fortemente consigliato per evitare ambiguità ed errori che ne conseguono.

14.2 Collegamento a wire

Un `reg` si può utilizzare come “fonte di valore” per un `wire`. Questo equivale circuitalmente a collegare il `wire` all'uscita del `reg`.

```
output out;  
reg OUT;  
assign out = OUT;
```

In questo caso, `out` seguirà sempre e in modo continuo il valore di `OUT`, propagandolo a ciò a cui viene collegato a sua volta. In questo caso non introduciamo nessun ritardo `#T` nell' `assign` perché si tratta di un semplice collegamento senza logica combinatoria aggiunta.

Allo stesso modo, si può collegare un `reg` all'ingresso di una rete.

```
reg [3:0] X, Y;  
add #( .N(4) ) a(  
    .x(X), .y(Y), .c_in(1'b0),  
    ...  
);
```

Non ha invece alcun senso cercare di fare il contrario, ossia collegare direttamente un `wire` all'ingresso di un `reg`. Anche se questo ha senso circuitalmente, Verilog richiede di esprimere questo all'interno di un blocco `always` per indicare anche *quando* aggiornare il valore del `reg`.

14.3 Struttura generale di un blocco always

Il valore di un reg si aggiorna all'interno di blocchi `always`. La sintassi generale di questi blocchi è la seguente

```
always @( event ) [if( cond )] [ #T ] begin
    [multiple statements]
end
```

Il funzionamento è il seguente: ogni volta che accade `event`, se `cond` è vero e dopo tempo `T`, vengono eseguiti gli `statement` indicati. Se lo `statement` è uno solo, si possono anche omettere `begin` e `end`.

Per Verilog, qui come *statement* si possono usare tutte le sintassi procedurali che si desiderano, incluse quelle discusse per le testbench che permettono di scrivere un classico programma "stile C". Per noi, *no*. Useremo questi blocchi in dei modi specifici per indicare

3. come si comportano i registri al reset,
3. come si comportano i registri al fronte positivo del `clock`.

14.4 Comportamento al reset

Per indicare il comportamento al reset useremo `statement` del tipo

```
always @(reset_ == 0) begin
    R1 = 0;
end
```

Il funzionamento è facilmente intuibile: finché `reset_` è a 0, il reg è impostato al valore indicato. Il blocco `begin ... end` può contenere l'inizializzazione di più registri. Tipicamente, raggrupperemo tutte le inizializzazioni in una *descrizione*, mentre le terremo separate in una *sintesi*.

Un registro può non essere inizializzato: in tal caso, il suo valore sarà *non specificato*, in Verilog `x`. Ricordiamo che questo significa che il registro *ha* un qualche valore misurabile, ma non è possibile determinare logicamente a priori e in modo univoco quale sarà.

In un blocco `reset` è *indifferente* l'uso di `=` o `<=` per gli assegnamenti (vedere sezione più avanti).

Valore assegnato al reset

Per la sintassi Verilog, a destra dell'assegnamento si potrebbe utilizzare qualunque espressione, sia questa costante (per esempio, il letterale `1'b0` o un `parameter`) o variabile (per esempio, il wire `w`).

Se pensiamo però all'equivalente circuitale, hanno senso solo valori costanti. Infatti, impostare un valore al reset equivale a collegare opportunamente i piedini `preset_` e `preclear_` del registro.

14.5 Aggiornamento al fronte positivo del clock

Per indicare il comportamento al fronte positivo del `clock` useremo `statement` del tipo

```
always @(posedge clock) if(reset_ == 1) #3 begin
    OUT <= ~OUT;
end
```

Il funzionamento è il seguente: ad ogni fronte positivo del `clock`, se `reset_` è a 1 e dopo 3 unità di tempo, il registro viene aggiornato con il valore indicato. Differentemente dal reset, qui si può utilizzare qualunque logica combinatoria per il calcolo del nuovo valore del registro.

L'unità di tempo (impostato a 3 *in questo corso* solo per convenzione, così come il periodo del clock a 10 unità) rappresenta il tempo di propagazione $T_{propagation}$ del registro, ossia il tempo che passa dal fronte del clock prima che il registro mostri in uscita il nuovo valore.

Tutti gli assegnamenti in questi blocchi devono usare l'operatore `<=`, e non `=`. Come spiegato nella sezione più avanti, questo è necessario perché i registri simulati siano non-trasparenti.

Tipicamente usiamo registri *multifunzionali*, ossia che operano in maniera diversa in base allo stato della rete.

In una *descrizione*, questo si fa usando un singolo registro di stato `STAR` e indicando il comportamento dei vari registri multifunzionali al variare di `STAR`. Questo ci fa vedere in generale come si comporta l'intera rete al variare di `STAR`.

In questa notazione, è lecito omettere un registro in un dato stato, implicando che quel registro *conserva* il valore precedentemente assegnato.

```
localparam S0 = 0, S1 = 1;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex(STAR)
        S0: begin
```

```

        A <= ~B;
        B <= A;
        STAR <= (A == 1'b0) ? S1 : S0;
    end
    S1: begin
        A <= B;
        B <= ~A;
        STAR <= (B == 1'b1) ? S1 : S0;
    end
endcase
end

```

In una *sintesi*, invece, si sintetizza ciascun registro individualmente come un multiplexer guidato da una serie di *variabili di comando*. Il multiplexer ha come ingressi *tutti* i risultati combinatori che il registro utilizza, e in base allo stato (da cui vengono generate le variabili di comando) solo uno di questi è utilizzato per aggiornare il registro al fronte positivo del clock. Questo è rappresentato in Verilog utilizzando le variabili di comando per discriminare il *case*, e indicando un comportamento combinatorio per ciascun valore di queste variabili. In questa notazione, non è lecito omettere le operazioni di conservazione, mentre è lecito utilizzare non specificati per indicare comportamenti assegnati a più ingressi del multiplexer. Nell'esempio sotto, con `2'b1X` si indica che a entrambi gli ingressi `10` e `11` del multiplexer è collegato il valore `DAV_`.

```

always @(posedge clock) if(reset_ == 1) #3 begin
    casex(b1, b0)
        2'b00: DAV_ <= 0;
        2'b01: DAV_ <= 1;
        2'b1X: DAV_ <= DAV_;
    endcase
end

```

14.6 Limitazioni della simulazione: temporizzazione, non-trasparenza e operatori di assegnamento

Ci sono alcune differenze tra i registri, intesi come componenti elettronici, e i *reg* descritti in Verilog così come abbiamo visto. Queste differenze non sono d'interesse *se non si fanno errori*. In caso di errori, si potrebbero osservare comportamenti altrimenti inspiegabili, ed è per questo che è utile conoscere queste differenze per poter risalire alla fonte del problema.

I registri hanno caratteristiche di temporizzazione sia prima che dopo il fronte positivo del clock: ciascun ingresso va impostato almeno T_{setup} prima del fronte positivo, mantenuto fino ad almeno T_{hold} dopo, e il valore in ingresso è rispecchiato in uscita solo dopo $T_{propagation}$.

Date le semplici strutture sintattiche che utilizziamo, la simulazione non è così accurata e non considera T_{setup} e T_{hold} . In particolare, il simulatore campiona i valori in ingresso non *prima* del fronte positivo, ma direttamente quando aggiorna il valore dei registri, ossia *dopo* $T_{propagation}$ dal fronte positivo del clock.

In altre parole: tutti i campionamenti e gli aggiornamenti dei registri sono fatti allo stesso tempo di simulazione, ossia $T_{propagation}$ dopo il fronte positivo del clock.

Questo porterebbe a violare la non-trasparenza dei registri, se non fosse per l'operatore di assegnamento `<=`, detto *non-blocking assignment*. Questo operatore si comporta in questo modo: tutti gli assegnamenti `<=` contemporanei (ossia allo stesso tempo di simulazione) non hanno effetto l'uno sull'altro perché campionano il *right hand side* all'inizio del time-step e aggiornano il *left hand side* alla fine del time-step.

Questo simula correttamente la non-trasparenza dei registri, ma solo se *tutti* usano `<=`. Gli assegnamenti con `=`, detti *blocking assignment*, sono invece eseguiti completamente e nell'ordine in cui li incontra il simulatore (si assuma che quest'ordine sia del tutto casuale).

Al tempo di reset questo ci è indifferente, perché sono (circuitalmente) leciti solo assegnamenti con valori costanti e non si possono quindi creare anelli per cui è di interesse la non-trasparenza.

Capitolo 15

Simulazione ed uso di GTKWave

Documentiamo qui il software da utilizzare per il testing e debugging delle reti prodotte, ossia iverilog, vvp e GTKWave. A differenza dell'ambiente per Assembler, questi sono facilmente reperibili per ogni piattaforma, o compilabili dal sorgente. In sede d'esame si utilizzano da un normale terminale Windows, senza utilizzare macchine virtuali. [Qui](#) si trovano installer per Windows.

Negli esercizi di esame vengono forniti i file necessari a compilare simulazioni per testare la propria rete. Questi sono tipicamente i file `testbench.v` e `reti_standard.v`. Il primo contiene una serie di test che verificano il corretto comportamento della rete prodotta rispetto alle specifiche richieste. Il secondo contiene invece delle reti combinatorie che si potranno assumere note e sintetizzabili, da usare per la sintesi di rete combinatoria.

Non tutti gli esercizi hanno una parte di sintesi di rete combinatoria, e quindi il file `reti_standard.v`. Inoltre, ciascun esercizio ha il *proprio* file `reti_standard.v`, che sarà diverso da quelli allegati ad altri esercizi.

15.1 Compilazione e simulazione

Sia `descrizione.v` il sorgente contenente la descrizione della rete sincronizzata da noi prodotto, e che vogliamo testare.

Si compila la simulazione con il comando da terminale `iverilog`. Il comando richiede come argomenti i file da compilare assieme. Di default, il binario prodotto si chiamerà `a.out`, mentre con l'opzione `-o nome` è possibile impostarne uno a scelta. Per esempio:

```
iverilog -o desc testbench.v reti_standard.v descrizione.v
```

Il file prodotto non è eseguibile da solo, ma va lanciato usando `vvp`. Per esempio:

```
vvp desc
```

Questo lancerà la simulazione. In un test di successo, vedremo le seguenti stampe:

```
VCD info: dumpfile waveform.vcd opened for output.  
$finish called at [un numero]
```

La prima stampa ci informa che il file `waveform.vcd` sta venendo popolato, la seconda ci informa del tempo di simulazione al quale questa è terminata con il comando `$finish`. Alcune versioni di `vvp` non stampano quest'ultima di default - non è un problema.

Le testbench degli esercizi d'esame stampano a video quando incontrano un errore: un test fallito avrà quindi delle righe in più in mezzo a quelle presentate qui. Per esempio, `Timeout - waiting for signal failed` indica che la simulazione si era bloccata in attesa di un evento che non è mai accaduto, come un segnale di handshake.

Le testbench non sono mai complete

Se la simulazione non stampa errori, questo indica solo che la testbench *non ne ha trovato alcuno*. Non implica, invece, che *non ci siano* errori. Questo sia perché è impossibile scrivere una testbench davvero esaustiva per tutti i possibili percorsi di esecuzione, ma anche perché è facile scrivere Verilog che *sembra* funzionare bene ma che in realtà usa costrutti che rendono la rete irrealizzabile in hardware.

È sempre responsabilità dello studente assicurarsi che non ci siano errori. In fase di autocorrezione, anche se la testbench non trova nessun errore, è sempre possibile (anzi, dovuto) assicurarsi della correttezza del compito e fare correzioni se necessarie.

15.1.1 Testbench con `timescale

Con la sintassi ``timescale` è possibile controllare l'unità di misura default e la granularità della simulazione. Per esempio, un file `testbench.v` che comincia come segue imposta l'unità di misura a 1s (il solito) e la granularità di simulazione a 1ms, permettendo di osservare cambiamenti più veloci di un secondo.

```
`timescale 1s/ms

module testbench();
...

```

Questa sintassi è utilizzata in alcuni testi d'esame, per esempio se sono previste RC particolarmente veloci. Per maggiori dettagli, vedere [qui](#).

Se la sintassi ``timescale` è utilizzata, è *obbligatorio* compilare la simulazione ponendo il file `testbench.v` come primo file del comando, ossia `iverilog -o desc testbench.v`
In caso contrario, il compilatore stamperà il seguente warning:

```
warning: Found both default and `timescale based delays.
```

15.2 Waveform e debugging

La simulazione genera un file `waveform.vcd` contenente l'evoluzione di tutti i fili e registri nella simulazione. Questo file è prodotto grazie alle seguenti righe, incluse in tutte le testbench:

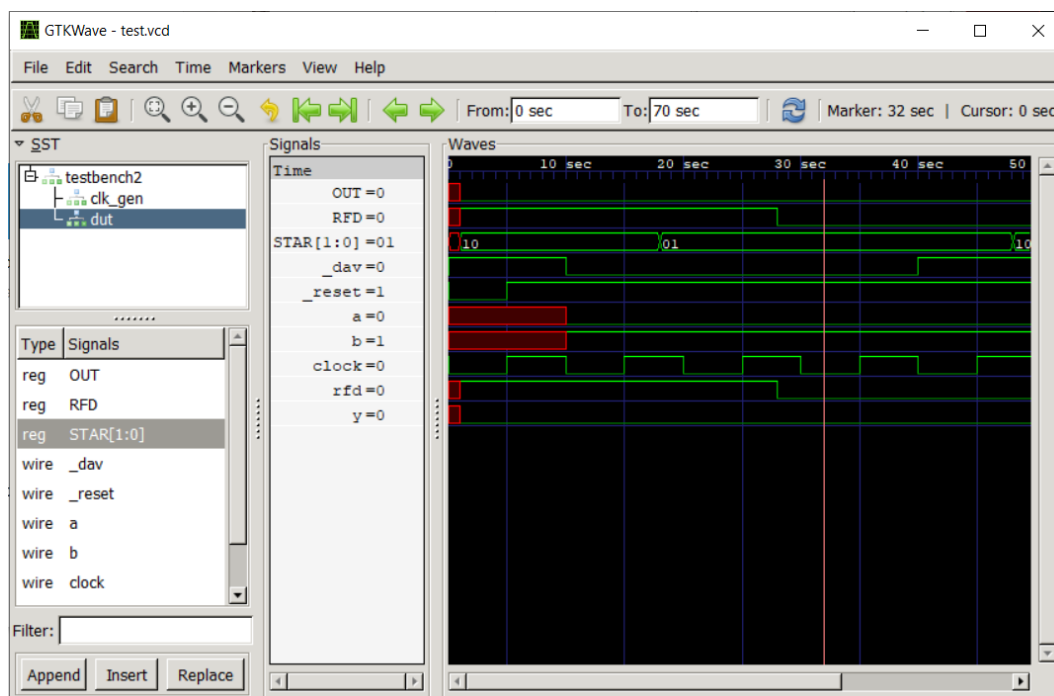
```
initial begin
    $dumpfile("waveform.vcd");
    $dumpvars;
    ...

```

Con questo file possiamo studiare l'evoluzione della rete e trovare eventuali errori. Per analizzarlo, usiamo GTK-Wave, richiamabile da terminale con

```
gtkwave waveform.vcd
```

Si dovrebbe aprire quindi una finestra dal quale possiamo analizzare l'evoluzione della rete.



Il programma mostra sulla sinistra le varie componenti nella simulazione e, se li selezioniamo, i fili e registri che li compongono. Ci interesserà in particolare `dut` (*device under test*), che sarà proprio il componente da noi realizzato. Selezionando poi i vari `wire` e `reg` che compaiono sotto, e cliccando "Append", compariranno nella schermata a destra, dove possiamo vedere l'evoluzione nel tempo.

15.2.1 Zoom, ordinamento, formattazione

Lo zoom della timeline a destra è regolabile, usando la rotellina del mouse o le lenti d'ingrandimento in alto a sinistra. Cliccando in punti specifici della timeline spostiamo il cursore, cioè la linea rossa verticale. Possiamo quindi leggere nella colonna centrale il valore di ciascun segnale all'istante dove si trova il cursore.

I segnali nella schermata principale sono ordinabili, per esempio è in genere utile spostare `clock` e `STAR` in alto. Di default, sono formattati come segnali binari, se composti da un bit, o in notazione esadecimale, se da più bit. Cliccando col destro su un segnale è possibile cambiare la formattazione in diversi modi, incluso decimale.

15.2.2 Non specificati e alta impedenza


Prestare particolare attenzione ai valori non specificati (`x`) e alta impedenza (`z`), che sono spesso sintomi di errori, per esempio per un filo di input non collegato.

Nella waveform, i valori non specificati sono evidenziati con un'area rossa, mentre i fili in alta impedenza sono evidenziati con una linea orizzontale gialla posta a metà altezza tra 0 e 1.

15.2.3 Pulsante *Reload*

Il comando `gtkwave waveform.vcd` blocca il terminale da cui viene lanciato, rendendo impossibile mandare altri comandi finché non viene chiuso. È quindi frequente vedere studenti chiudere e riaprire GTKWave ogni volta che c'è bisogno di risimulare la rete.

Questo approccio è però inefficiente, dato che si dovrà ogni volta riselectare i fili, riformattarne i valori, ritrovare il punto d'errore che si stava studiando.

Il pulsante *Reload*, indicato con l'icona , permette di ricaricare il file `waveform.vcd` senza chiudere e riaprire il programma, e mantenendo tutte le selezioni fatte.

È per questo una buona idea utilizzare una delle seguenti strategie:

3. usare due terminali, uno dedicato a `iverilog` e `vvp`, l'altro a `gtkwave`;
3. lanciare il comando `gtkwave` in background. Nell'ambiente Windows all'esame, questo si può fare aggiungendo un `&` in fondo: `gtkwave waveform.vcd &`.

In entrambi i casi, otteniamo di poter rieseguire la simulazione mentre GTKWave è aperto, e poter quindi sfruttare il pulsante *Reload*.

Se l'operatore `&` non funziona

In alcune installazioni di Powershell l'operatore `&` non funziona. L'operatore è un semplice alias per `Start-Job`, e si può ovviare al problema usando questo comando per esteso:

```
Start-Job gtwave waveform.vcd
```

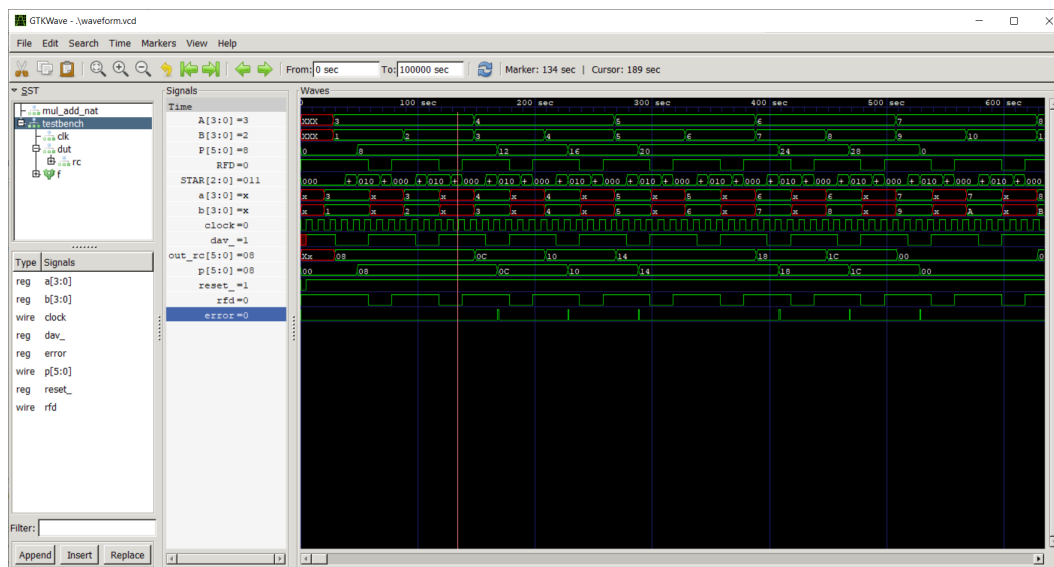
L'operatore è documentato [qui](#).

15.2.4 Linea di errore

Nelle testbench d'esame è (di solito) presente anche una *linea di errore* che permette di identificare subito i punti in cui la testbench ha trovato un errore. Questo è particolarmente utile per scorrere lunghe simulazioni.

Queste linee sono realizzate nella testbench con una variabile `reg error` inizializzata a 0 e un blocco `always` che risponde ad ogni variazione di `error` per rimetterla a 0 dopo una breve attesa. Questa attesa breve ma non nulla fa sì che basti assegnare 1 ad `error` per ottenere un'impulso sulla linea, facilmente visibile.

In GTKWave, possiamo trovare il segnale `error` tra i `wire` e `reg` del modulo `testbench` (*non* in `dut`). Mostrando questo segnale, possiamo riconoscere i punti di errore come impulsi, come nell'esempio seguente.



Parte III

Verilog - Appendice

Simulatore processore sEP8

Il processore sEP8 (Simple Educational Processor 8 bit) è un semplice processore a 8 bit, descritto nel libro [Dalle porte AND OR NOT al sistema calcolatore](#) del prof. Paolo Corsini e utilizzato nel corso per presentare i meccanismi fondamentali di un processore e la realizzazione dello stesso in hardware tramite il linguaggio Verilog. Oltre che a questi scopi didattici, è anche una buona base di partenza per esplorare architetture di processori e le loro implementazioni in hardware.

Nel repository <https://github.com/Unipisa/sEP8> contiene codice utilizzabile per simulare il processore, sperimentarne estensioni etc.

Attribuzione

Il codice attualmente presente nel repository è frutto del lavoro di [Nicola Ramacciotti](#) nell'ambito della sua Tesi di Laurea in Ingegneria Informatica, dal titolo " *Design e implementazione di un ambiente di simulazione e testing in Verilog per il processore sEP8* ".

Siamo disponibili a seguire tesi triennali sull'argomento.

16.1 Lancio di simulazioni

Il codice sorgente fornito include tutto il necessario per simulare l'architettura sEP8 e osservare il suo comportamento interno durante l'esecuzione di un semplice programma.

```
> iverilog -o sEP8 .\sEP8.v .\MEMORIA.v .\RAM.v .\ROM.v .\IO.v .\clock_generator.v .\testbench.v
> vvp ./sEP8
```

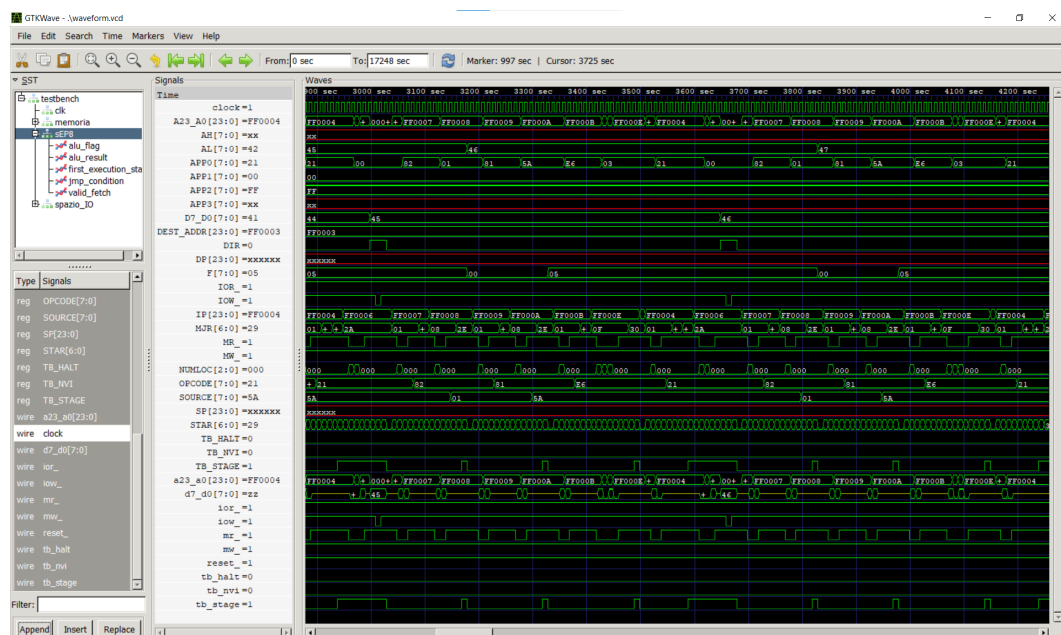
```
VCD info: dumpfile waveform.vcd opened for output.
```

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Simulazione terminata: il processore ha eseguito un'istruzione HLT

I caratteri che vediamo stampati (da A a Z) sono l'output del programma contenuto in ROM.v.

Dal file `waveform.vcd`, possiamo studiare il comportamento del processore.



16.2 Caricamento di programmi tramite ROM

In questo simulatore, il programma da eseguire è caricato tramite un modulo ROM, che viene montato a partire dall'indirizzo `24'hFF0000`. Tale modulo dovrà contenere le sequenze di byte corrispondenti alle istruzioni del programma. Un assembler basilare è fornito come script python, che traduce semplici programmi assembler per questo processore in un modulo ROM contenente la giusta sequenza di byte.

16.2.1 Riferimenti storici: le cartucce

Questo modo di caricare i programmi non è solo una semplificazione a scopo didattico, ma ha anche degli esempi storici concreti.

Infatti operavano così le prime console, dove gli esempi più famosi sono probabilmente le console Nintendo come N64 e GameBoy. In queste console, ciascun gioco è fornito come una *cartuccia* che va inserita nella console prima di accenderla. Ciascuna cartuccia è del vero e proprio hardware che contiene la ROM dentro la quale è scritto il *programma* del gioco, e i pin che collegano la cartuccia e la console sono proprio fili di indirizzamento e dati.



In realtà, permettendo generiche letture e scritture al range di indirizzi a cui è montata la cartuccia, questa poteva contenere diversi tipi di hardware, inclusi chip di RAM aggiuntivi, chip di memoria persistente per permettere di salvare il gioco, o ancora hardware dedicato come [rumble pak](#) o [fotocamera](#).