

# Documentazione per Esercitazioni di Reti Logiche

Raffaele Zippo

3 gennaio 2025

# Indice

<b>I Documentazione Assembler</b>	<b>5</b>
<b>1 Architettura x86</b>	<b>7</b>
1.1 Registri . . . . .	7
1.2 Memoria . . . . .	8
1.3 Spazio di I/O . . . . .	8
1.4 Condizioni al reset . . . . .	8
<b>2 Istruzioni processore x86</b>	<b>9</b>
2.1 Spostamento di dati . . . . .	9
2.2 Aritmetica . . . . .	9
2.3 Logica binaria . . . . .	10
2.4 Traslazione e Rotazione . . . . .	10
2.5 Controllo di flusso . . . . .	11
2.6 Operazioni condizionali . . . . .	11
2.7 Istruzioni stringa . . . . .	13
2.8 Altre istruzioni . . . . .	13
<b>3 Sottoprogrammi di utility</b>	<b>15</b>
3.1 Terminologia . . . . .	15
3.2 Caratteri speciali . . . . .	15
3.3 Sottoprogrammi . . . . .	15
<b>4 Debugger gdb</b>	<b>17</b>
4.1 Controllo dell'esecuzione . . . . .	17
4.2 Ispezione dei registri . . . . .	18
4.3 Ispezione della memoria . . . . .	18
4.4 Gestione dei breakpoints . . . . .	18
<b>5 Tabella ASCII</b>	<b>21</b>
<b>6 Ambiente d'esame e i suoi script</b>	<b>23</b>
6.1 Aprire l'ambiente . . . . .	23
6.2 Il terminale Powershell . . . . .	24
6.3 Eseguire gli script . . . . .	24
<b>7 Problemi comuni</b>	<b>27</b>
7.1 Setup dell'ambiente . . . . .	27
7.2 Uso dell'ambiente . . . . .	28
<b>II Documentazione Verilog</b>	<b>29</b>
<b>8 Introduzione</b>	<b>31</b>
<b>9 Operatori</b>	<b>33</b>
9.1 Valori letterali ( <i>literal values</i> ) . . . . .	33
9.2 Operatori aritmetici . . . . .	33
9.3 Operatori logici e <i>bitwise</i> . . . . .	33
9.4 Operatore di selezione [...] . . . . .	34

9.5 Operatore di concatenazione {...}	34
9.6 Operazioni comuni	35
<b>10 Sintassi per reti combinatorie</b>	<b>37</b>
10.1 module	37
10.2 wire	37
10.3 Usare un module in un altro module	37
10.4 Tabelle di verità	38
10.5 Multiplexer	39
10.6 Reti parametrizzate	39
<b>11 Sintassi per reti sincronizzate</b>	<b>41</b>
11.1 Istanziamento	41
11.2 Collegamento a wire	41
11.3 Struttura generale di un blocco always	41
11.4 Comportamento al reset	42
11.5 Aggiornamento al fronte positivo del clock	42
11.6 Limitazioni della simulazione: temporizzazione, non-trasparenza e operatori di assegnamento	43
<b>12 Simulazione ed uso di GTKWave</b>	<b>45</b>
12.1 Compilazione e simulazione	45
12.2 Waveform e debugging	46
<b>III Uso di VS Code</b>	<b>49</b>
<b>13 Essere efficienti con VS Code</b>	<b>51</b>
13.1 Le basi elementari	51
13.2 Le basi un po' meno elementari	51
13.3 Editing multi-caret	51



## **Parte I**

# **Documentazione Assembler**



# 1. Architettura x86

Riportiamo qui una vista *semplificata e riassuntiva* dell'architettura x86 per la quale scriveremo programmi assembler.

L'architettura x86 è a 32 bit. Questo implica che i registri generali, così come tutti gli indirizzi per locazioni in memoria, sono a 32 bit. L'evoluzione di questa architettura, x64 a 64 bit, che è quella che troviamo nei processori in commercio, è del tutto retrocompatibile.

## Importanti semplificazioni

La visione del processore che proponiamo è molto limitata, ed omette diversi importanti registri, flag e funzionalità che saranno esplorati in corsi successivi. Questi includono, per esempio, il registro `ebp`, la natura dei meccanismi di protezione, il significato di `SEGMENTATION FAULT`, e che cosa sia un *kernel*.

Quanto discutiamo è tuttavia sufficiente agli scopi didattici di questo corso.

## 1.1 Registri

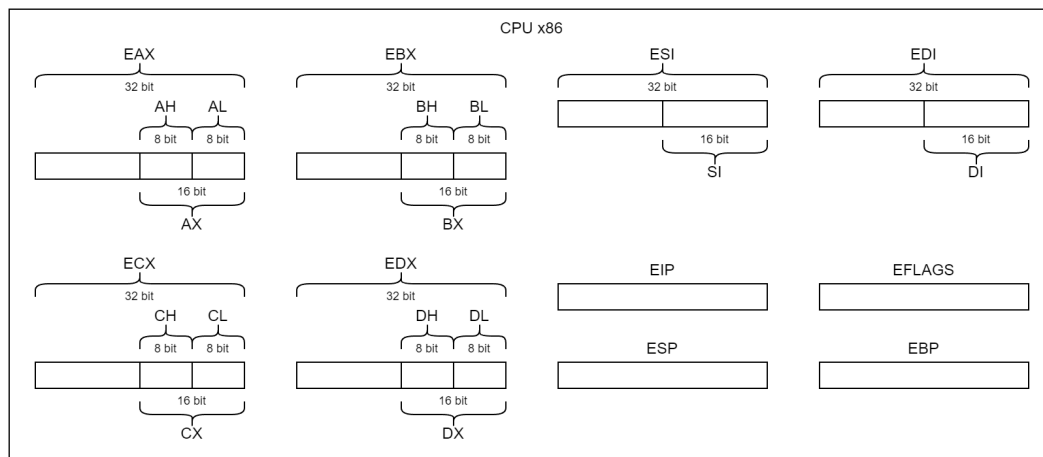
I registri che utilizzeremo *direttamente* sono 6: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`. Per i primi quattro di questi, è possibile operare sulle loro porzioni a 16 e 8 bit tramite `ax`, `ah`, `al` e così via. Per i registri `esi` ed `edi` è possibile operare solo sulle porzioni a 16 bit, tramite `si` e `di`. Tipicamente, i registri `eax`... `edx` sono utilizzati per processare dati, mentre `esi` ed `edi` sono utilizzati come registri puntatori. Questa divisione di utilizzo non è però affatto obbligatoria per la maggior parte delle istruzioni.

Altri registri sono invece utilizzati in modo indiretto:

- `esp` è il registro puntatore per la *cima* dello stack, viene utilizzato da `pop` / `push` per prelevare/spostare valori nella pila, e da `call` / `ret` per la chiamata di sottoprogrammi;
- `eip` è il registro puntatore verso la prossima istruzione da eseguire, viene incrementato alla fine del *fetch* di una istruzione e modificato da istruzioni che cambiano il flusso d'esecuzione, come `call`, `ret` e le varie `jmp`;
- `eflags` è il registro dei flag, una serie di booleani con informazioni sullo stato dell'esecuzione e sul risultato dell'ultima operazione aritmetica.

Sono tipicamente aggiornati dalle istruzioni aritmetiche, e testati indirettamente con istruzioni condizionali come `jcon`, `set` e `cmov`.

Di seguito uno schema funzionale dei registri del processore x86.



## 1.2 Memoria

Lo spazio di memoria dell'architettura x86 è indirizzato su 32 bit. Ciascun indirizzo corrisponde a un byte, ma è possibile eseguire anche letture e scritture a 16 e 32 bit.

Per tali casi è importante ricordare che l'architettura x86 è *little-endian*, che significa **little end first**, [un riferimento a I viaggi di Gulliver](#). Questo si traduce nel fatto che quando un valore di  $n$  byte viene salvato in memoria a partire dall'indirizzo  $a$ , il byte meno significativo del valore viene salvato in  $a$ , il secondo meno significativo in  $a + 1$ , e così via fino al più significativo in  $a + (n - 1)$ .

Questo ordinamento dei byte in memoria non inficia sulla coerenza dei dati nei registri: eseguendo `movl %eax, a` e `movl a, %eax` il contenuto di `eax` non cambia, e l'ordinamento dei bit rimane coerente.

I *meccanismi di protezione* ci precludono l'accesso alla maggior parte dello spazio di memoria. Potremmo accedere senza incorrere in errori solo

1. allo stack
2. allo spazio allocato nella sezione `.data`
3. alle istruzioni nella sezione `.text`

Queste sezioni tipicamente non includono gli indirizzi “bassi”, cioè a partire da `0x0`.

È importante anche tenere presente che

1. non è possibile *eseguire* istruzioni dallo stack e da `.data`
2. non è possibile *scrivere* nella sezione `.text`

Vanno quindi opportunamente dichiarate le sezioni, e vanno evitate operazioni di `jmp`, `call` etc. verso locazioni di `.data` così come le `mov` verso locazioni di `.text`.

In caso di violazione di questi meccanismi, l'errore più tipico è `SEGMENTATION FAULT`.

## 1.3 Spazio di I/O

Lo spazio di I/O, sia quello fisico (monitor, speaker, tastiera, etc.) sia quello virtuale (terminale, files su disco, etc.) ci è in realtà precluso tramite *meccanismi di protezione*. Tentare di eseguire istruzioni `in` o `out` porterà infatti al brusco arresto del programma. Il nostro programma può interagire con lo spazio di I/O solo tramite il *kernel* del *sistema operativo*.

Tutta questa complessità è astratta tramite i *sottoprogrammi di input/output* dell'ambiente, documentati [qui](#).

## 1.4 Condizioni al reset

Il reset iniziale e l'avvio del nostro programma sono concetti completamente diversi e scollegati. Non possiamo sfruttare nessuna ipotesi sullo stato dei registri al momento dell'avvio del nostro programma, se non che il registro `eip` punterà ad un certo punto alla prima istruzione di `_main`.

Il fatto che `_main` sia l'entrypoint del nostro programma, così come l'uso di `ret` senza alcun valore di ritorno, è una caratteristica di *questo* ambiente.



## 2. Istruzioni processore x86

Le seguenti tabelle sono per *riferimento rapido* : sono utili per la programmazione pratica, ma omettono molteplici dettagli che serve sapere, e che trovate nel resto del materiale.

Si ricorda che utilizziamo la sintassi GAS/AT&T, dove le istruzioni sono nel formato *opcode source destination* . Nella colonna notazione, indicheremo con [bwl] le istruzioni che richiedono la specifica delle dimensioni. Quando la dimensione è deducibile dai registri utilizzati, questi suffissi si possono omettere.

Per gli operandi, indicheremo qui con *r* un registro (come in `mov %eax, %ebx`); con *m* un indirizzo di memoria (immediato, come in `mov numero, %eax`, o tramite registro, come in `mov (%esi), %eax`, o ancora con indice, come in `mov matrice(%esi, %ecx, 4)`); con *i* un valore immediato (come in `mov $0, %eax`).

Si ricorda che non tutte le combinazioni sono permesse nell'architettura x86: nessuna istruzione generale supporta l'indicazione di *entrambi* gli operandi in memoria (cioè, non si può scrivere `movl x, y` o `mov (%eax), (%ebx)`). Fanno eccezione le istruzioni *stringa* come la `movs`, usando operandi impliciti.

### 2.1 Spostamento di dati

Istruzione	Nome esteso	Notazione	Comportamento
mov	Move	mov[bwl] r/m/i, r/m	Scriva il valore sorgente nel destinatario. Non modifica ZF .
lea	Load Effective Address	lea a, r	Scriva l'indirizzo nel registro destinatario.
xchg	Exchange	xchg[bwl] r/m, r/m	Scambia il valore del sorgente con quello del destinatario.
cbw	Convert Byte to Word	cbw	Estende il contenuto di %al su %ax , interpretandone il contenuto come intero.
cwde	Convert Word to Doubleword	cwde	Estende il contenuto di %ax su %eax , interpretandone il contenuto come intero.
push	Push onto the Stack	push[wl] r/m/i	Aggiunge il valore sorgente in cima allo stack (destinatario implicito).
pop	Pop from the Stack	pop[wl] r/m	Rimuove un valore dallo stack (sorgente implicito) lo scrive nel destinatario.

### 2.2 Aritmetica

Istruzione	Nome esteso	Notazione	Comportamento
add	Addition	add[bwl] r/m/i, r/m	Somma sorgente e destinatario, scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna CF e OF .
sub	Subtraction	sub[bwl] r/m/i, r/m	Sottrae il sorgente dal destinatario, scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna CF e OF .
adc	Addition with Carry	adc[bwl] r/m/i, r/m	Somma sorgente, destinatario e CF , scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna CF e OF .
sbb	Subtraction with Borrow	sub[bwl] r/m/i, r/m	Sottrae il sorgente e CF dal destinatario, scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna CF e OF .
inc	Increment	inc[bwl] r/m	Somma 1 (sorgente implicito) al destinatario. Non aggiorna CF .
dec	Decrement	dec[bwl] r/m	Sottrae 1 (sorgente implicito) al destinatario. Non aggiorna CF .
neg	Negation	neg[bwl] r/m	Sostituisce il destinatario con il suo opposto. Aggiorna OF .

Le seguenti istruzioni hanno operandi e destinatari impliciti, che variano in base alla dimensione dell'operazione. Usano in oltre composizioni di più registri: useremo %dx\_%ax per indicare un valore i cui bit più significativi sono scritti in %dx e quelli meno significativi in %ax .

Istruzione	Nome esteso	Notazione	Comportamento
mul	Unsigned Multiply, 8 bit	mulb r/m	Calcola su 16 bit il prodotto tra naturali del sorgente e %al , scrive il risultato su %ax . Se il risultato non è riducibile a 8 bit, mette CF e OF a 1, altrimenti a 0.
mul	Unsigned Multiply, 16 bit	mulw r/m	Calcola su 32 bit il prodotto tra naturali del sorgente e %ax , scrive il risultato su %dx_%ax . Se il risultato non è riducibile a 16 bit, mette CF e OF a 1, altrimenti a 0.

mul	Unsigned Multiply, 32 bit	mull r/m	Calcola su 64 bit il prodotto tra naturali del sorgente e %eax , scrive il risultato su %edx_%eax . Se il risultato non è riducibile a 32 bit, mette CF e OF a 1, altrimenti a 0.
imul	Signed Multiply, 8 bit	imulb r/m	Calcola su 16 bit il prodotto tra interi del sorgente e %a1 , scrive il risultato su %ax . Se il risultato non è riducibile a 8 bit, mette CF e OF a 1, altrimenti a 0.
imul	Signed Multiply, 16 bit	imulw r/m	Calcola su 32 bit il prodotto tra interi del sorgente e %ax , scrive il risultato su %dx_%ax . Se il risultato non è riducibile a 16 bit, mette CF e OF a 1, altrimenti a 0.
imul	Signed Multiply, 32 bit	imull r/m	Calcola su 64 bit il prodotto tra interi del sorgente e %eax , scrive il risultato su %edx_%eax . Se il risultato non è riducibile a 32 bit, mette CF e OF a 1, altrimenti a 0.

Istruzione	Nome esteso	Notazione	Comportamento
div	Unsigned Divide, 8 bit	divb r/m	Calcola su 8 bit la divisione tra naturali tra %ax (dividendo implicito) e il sorgento (divisore). Scrive il quoziente su %a1 e il resto su %ah . Se il quoziente non è rappresentabile su 8 bit, causa <i>crash del programma</i> .
div	Unsigned Divide, 16 bit	divw r/m	Calcola su 16 bit la divisione tra naturali tra %dx_%ax (dividendo implicito) e il sorgento (divisore). Scrive il quoziente su %ax e il resto su %dx . Se il quoziente non è rappresentabile su 16 bit, causa <i>crash del programma</i> .
div	Unsigned Divide, 32 bit	divl r/m	Calcola su 32 bit la divisione tra naturali tra %edx_%eax (dividendo implicito) e il sorgento (divisore). Scrive il quoziente su %eax e il resto su %edx . Se il quoziente non è rappresentabile su 32 bit, causa <i>crash del programma</i> .
idiv	Signed Divide, 8 bit	idivb r/m	Calcola su 8 bit la divisione tra interi tra %ax (dividendo implicito) e il sorgento (divisore). Scrive il quoziente su %a1 e il resto su %ah . Se il quoziente non è rappresentabile su 8 bit, causa <i>crash del programma</i> .
idiv	Signed Divide, 16 bit	idivw r/m	Calcola su 16 bit la divisione tra interi tra %dx_%ax (dividendo implicito) e il sorgento (divisore). Scrive il quoziente su %ax e il resto su %dx . Se il quoziente non è rappresentabile su 16 bit, causa <i>crash del programma</i> .
idiv	Signed Divide, 32 bit	idivl r/m	Calcola su 32 bit la divisione tra interi tra %edx_%eax (dividendo implicito) e il sorgento (divisore). Scrive il quoziente su %eax e il resto su %edx . Se il quoziente non è rappresentabile su 32 bit, causa <i>crash del programma</i> .

## 2.3 Logica binaria

Le seguenti istruzioni operano *bit a bit* : data per esempio la and , l'i-esimo bit del risultato è l'and logico tra gli i-esimi bit di sorgente e destinatario.

Istruzione	Notazione	Comportamento
not	not[bwl] r/m	Sostituisce il destinatario con la sua negazione.
and	and r/m/i, r/m	Calcola l'and logico tra sorgente e destinatario, scrive il risultato sul destinatario.
or	or r/m/i, r/m	Calcola l'or logico tra sorgente e destinatario, scrive il risultato sul destinatario.
xor	xor r/m/i, r/m	Calcola lo xor logico tra sorgente e destinatario, scrive il risultato sul destinatario.

## 2.4 Traslazione e Rotazione

Istruzione	Nome esteso	Notazione	Comportamento
shl	Shift Logical Left	shl[bwl] i/r r/m	Sia n l'operando sorgente, esegue lo shift a sinistra del destinatario n volte. In ciascuno shift, il bit più significativo viene lasciato in CF . Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.
sal	Shift Arithmetic Left	sal[bwl] i/r r/m	Sia n l'operando sorgente, esegue lo shift a sinistra del destinatario n volte. Se il bit più significativo ha cambiato valore almeno una volta, imposta OF a 1. Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.
shr	Shift Logical Right	shr[bwl] i/r r/m	Sia n l'operando sorgente, esegue lo shift a destra del destinatario n volte, impostando a 0 gli n bit più significativi. In ciascuno shift, il bit più significativo viene lasciato in CF . Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.
sar	Shift Arithmetic Right	sar[bwl] i/r r/m	Sia n l'operando sorgente e s il valore del bit più significativo del destinatario, esegue lo shift a destra del destinatario n volte, impostando a s gli n bit più significativi. Il bit più significativo tra quelli rimossi viene lasciato in CF . Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.

rol	Rotate Left	rol[bwl] i/r r/m	Sia n l'operando sorgente, esegue la rotazione a sinistra del destinatario n volte. In ciascuna rotazione, il bit più significativo viene <i>sia</i> lasciato in CF <i>sia</i> ricopiato al posto del bit meno significativo. Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.
ror	Rotate Right	ror[bwl] i/r r/m	Sia n l'operando sorgente, esegue la rotazione a destra del destinatario n volte. In ciascuna rotazione, il bit meno significativo viene <i>sia</i> lasciato in CF <i>sia</i> ricopiato al posto del bit più significativo. Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.
rcl	Rotate with Carry Left	rcl[bwl] i/r r/m	Sia n l'operando sorgente, esegue la rotazione con carry a sinistra del destinatario n volte. In ciascuna rotazione, il bit più significativo viene lasciato in CF , mentre il valore di CF viene ricopiato al posto del bit meno significativo. Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.
rcr	Rotate with Carry Right	rcr[bwl] i/r r/m	Sia n l'operando sorgente, esegue la rotazione con carry a destra del destinatario n volte. In ciascuna rotazione, il bit meno significativo viene lasciato in CF , mentre il valore di CF viene ricopiato al posto del bit più significativo. Come registro sorgente si può utilizzare solo %c1 . Il sorgente può essere omissso, in quel caso n=1.

## 2.5 Controllo di flusso

Istruzione	Nome esteso	Notazione	Comportamento
jmp	Unconditional Jump	jmp m/r	Salta incondizionatamente all'indirizzo specificato.
call	Call Procedure	call m/r	Chiamata a procedura all'indirizzo specificato. Salva l'indirizzo della prossima istruzione nello stack, così che il flusso corrente possa essere ripreso con una ret .
ret	Return from Procedure	ret	Ritorna ad un flusso di esecuzione precedente, rimuovendo dallo stack l'indirizzo precedentemente salvato da una call .

La tabella seguente elenca i salti condizionati. I salti condizionati usano i flag per determinare se la condizione di salto è vera. Per un uso sempre coerente, assicurarsi che l'istruzione di salto segua immediatamente una cmp , o altre istruzioni che non hanno modificano i flag dopo la cmp . Dati gli operandi della cmp ed una condizione c , per esempio c = “maggiore o uguale”, la condizione è vera se destinatario c sorgente. Nella tabella che segue, quando ci si riferisce ad un confronto fra sorgente e destinatario si intendono gli operandi della cmp precedente.

Istruzione	Nome esteso	Notazione	Comportamento
cmp	Compare Two Operands	cmp[bwl] r/m/i, r/m	Confronta i due operandi e aggiorna i flag di conseguenza.
je	Jump if Equal	je m	Salta se destinatario == sorgente.
jne	Jump if Not Equal	jne m	Salta se destinatario != sorgente.
ja	Jump if Above	ja m	Salta se, interpretandoli come naturali, destinatario > sorgente.
jae	Jump if Above or Equal	jae m	Salta se, interpretandoli come naturali, destinatario >= sorgente.
jb	Jump if Below	jb m	Salta se, interpretandoli come naturali, destinatario < sorgente.
jbe	Jump if Below or Equal	jbe m	Salta se, interpretandoli come naturali, destinatario <= sorgente.
jg	Jump if Greater	jg m	Salta se, interpretandoli come interi, destinatario > sorgente.
jge	Jump if Greater or Equal	jge m	Salta se, interpretandoli come interi, destinatario >= sorgente.
jl	Jump if Less	jl m	Salta se, interpretandoli come interi, destinatario < sorgente.
jle	Jump if Less or Equal	jle m	Salta se, interpretandoli come interi, destinatario <= sorgente.
jz	Jump if Zero	jz m	Salta se ZF è 1.
jnz	Jump if Not Zero	jnz m	Salta se ZF è 0.
jc	Jump if Carry	jc m	Salta se CF è 1.
jnc	Jump if Not Carry	jnc m	Salta se CF è 0.
jo	Jump if Overflow	jo m	Salta se OF è 1.
jno	Jump if Not Overflow	jno m	Salta se OF è 0.
js	Jump if Sign	js m	Salta se SF è 1.
jns	Jump if Not Sign	jns m	Salta se SF è 0.

## 2.6 Operazioni condizionali

Per alcune operazioni tipiche, sono disponibili istruzioni specifiche il cui comportamento dipende dai flag e, quindi, dal risultato di una precedente cmp . Anche qui, quando ci si riferisce ad un confronto fra sorgente e destinatario si intendono gli operandi della cmp precedente.

La famiglia di istruzioni loop supportano i cicli condizionati più tipici. Rimangono d'interesse didattico come istruzioni specializzate ma, curiosamente, nei processori moderni sono generalmente meno performanti degli equivalenti che usino `dec`, `cmp` e salti condizionati.

Istruzione	Nome esteso	Notazione	Comportamento
loop	Unconditional Loop	loop m	Decrementa <code>%ecx</code> e salta se il risultato è (ancora) diverso da 0.
loope	Loop if Equal	loope m	Decrementa <code>%ecx</code> e salta se entrambe le condizioni sono vere: 1) <code>%ecx</code> è (ancora) diverso da 0, 2) destinatario == sorgente.
loopne	Loop if Not Equal	loopne m	Decrementa <code>%ecx</code> e salta se entrambe le condizioni sono vere: 1) <code>%ecx</code> è (ancora) diverso da 0, 2) destinatario != sorgente.
loopz	Loop if Zero	loopz m	Decrementa <code>%ecx</code> e salta se entrambe le condizioni sono vere: 1) <code>%ecx</code> è (ancora) diverso da 0, 2) ZF è 1.
loopnz	Loop if Not Zero	loopnz m	Decrementa <code>%ecx</code> e salta se entrambe le condizioni sono vere: 1) <code>%ecx</code> è (ancora) diverso da 0, 2) ZF è 0.

La famiglia di istruzioni set permettono di salvare il valore di un confronto in un registro o locazione di memoria. Tale operando può essere solo da 1 byte.

Istruzione	Nome esteso	Notazione	Comportamento
sete	Set if Equal	sete r/m	Imposta l'operando a 1 se destinatario == sorgente, a 0 altrimenti.
setne	Set if Not Equal	setne r/m	Imposta l'operando a 1 se destinatario != sorgente, a 0 altrimenti.
seta	Set if Above	seta r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario > sorgente, a 0 altrimenti.
setae	Set if Above or Equal	setae r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario >= sorgente, a 0 altrimenti.
setb	Set if Below	setb r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario < sorgente, a 0 altrimenti.
setbe	Set if Below or Equal	setbe r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario <= sorgente, a 0 altrimenti.
setg	Set if Greater	setg r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario > sorgente, a 0 altrimenti.
setge	Set if Greater or Equal	setge r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario >= sorgente, a 0 altrimenti.
setl	Set if Less	setl r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario < sorgente, a 0 altrimenti.
setle	Set if Less or Equal	setle r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario <= sorgente, a 0 altrimenti.
setz	Set if Zero	setz r/m	Imposta l'operando a 1 se ZF è 1, a 0 altrimenti.
setnz	Set if Not Zero	setnz r/m	Imposta l'operando a 1 se ZF è 0, a 0 altrimenti.
setc	Set if Carry	setc r/m	Imposta l'operando a 1 se CF è 1, a 0 altrimenti.
setnc	Set if Not Carry	setnc r/m	Imposta l'operando a 1 se CF è 0, a 0 altrimenti.
seto	Set if Overflow	seto r/m	Imposta l'operando a 1 se OF è 1, a 0 altrimenti.
setno	Set if Not Overflow	setno r/m	Imposta l'operando a 1 se OF è 0, a 0 altrimenti.
sets	Set if Sign	sets r/m	Imposta l'operando a 1 se SF è 1, a 0 altrimenti.
setns	Set if Not Sign	setns r/m	Imposta l'operando a 1 se SF è 0, a 0 altrimenti.

Istruzione	Nome esteso	Notazione	Comportamento
cmove	Move if Equal	cmove r/m r	Esegue la mov se destinatario == sorgente, altrimenti non fa nulla.
cmovne	Move if Not Equal	cmovne r/m	Esegue la mov se destinatario != sorgente, altrimenti non fa nulla.
cmova	Move if Above	cmova r/m	Esegue la mov se, interpretandoli come naturali, destinatario > sorgente, altrimenti non fa nulla.
cmovae	Move if Above or Equal	cmovae r/m	Esegue la mov se, interpretandoli come naturali, destinatario >= sorgente, altrimenti non fa nulla.
cmovb	Move if Below	cmovb r/m	Esegue la mov se, interpretandoli come naturali, destinatario < sorgente, altrimenti non fa nulla.
cmovbe	Move if Below or Equal	cmovbe r/m	Esegue la mov se, interpretandoli come naturali, destinatario <= sorgente, altrimenti non fa nulla.
cmovg	Move if Greater	cmovg r/m	Esegue la mov se, interpretandoli come interi, destinatario > sorgente, altrimenti non fa nulla.
cmovge	Move if Greater or Equal	cmovge r/m	Esegue la mov se, interpretandoli come interi, destinatario >= sorgente, altrimenti non fa nulla.
cmovl	Move if Less	cmovl r/m	Esegue la mov se, interpretandoli come interi, destinatario < sorgente, altrimenti non fa nulla.
cmovle	Move if Less or Equal	cmovle r/m	Esegue la mov se, interpretandoli come interi, destinatario <= sorgente, altrimenti non fa nulla.

cmovz	Move if Zero	cmovz r/m	Esegue la mov se ZF è 1, altrimenti non fa nulla.
cmovnz	Move if Not Zero	cmovnz r/m	Esegue la mov se ZF è 0, altrimenti non fa nulla.
cmovc	Move if Carry	cmovc r/m	Esegue la mov se CF è 1, altrimenti non fa nulla.
cmovnc	Move if Not Carry	cmovnc r/m	Esegue la mov se CF è 0, altrimenti non fa nulla.
cmovo	Move if Overflow	cmovo r/m	Esegue la mov se OF è 1, altrimenti non fa nulla.
cmovno	Move if Not Overflow	cmovno r/m	Esegue la mov se OF è 0, altrimenti non fa nulla.
cmovs	Move if Sign	cmovs r/m	Esegue la mov se SF è 1, altrimenti non fa nulla.
cmovns	Move if Not Sign	cmovns r/m	Esegue la mov se SF è 0, altrimenti non fa nulla.

## 2.7 Istruzioni stringa

Le istruzioni stringa sono ottimizzate per eseguire operazioni tipiche su vettori in memoria. Hanno esclusivamente operandi impliciti, che rende la specifica delle dimensioni *non* opzionale.

Istruzione	Nome esteso	Notazione	Comportamento
cld	Clear Direction Flag	cld	Imposta DF a 0, implicando che le istruzioni stringa procederanno per indirizzi crescenti.
std	Set Direction Flag	std	Imposta DF a 1, implicando che le istruzioni stringa procederanno per indirizzi decrescenti.
lods	Load String	lods[bwl]	Legge 1/2/4 byte all'indirizzo in %esi e lo scrive in %al / %ax / %eax. Se DF è 0, incrementa %esi di 1/2/4, se è 1 lo decrementa.
stos	Store String	stos[bwl]	Legge il valore in %al / %ax / %eax e lo scrive nei 1/2/4 byte all'indirizzo in %edi. Se DF è 0, incrementa %edi di 1/2/4, se è 1 lo decrementa.
movs	Move String to String	movs[bwl]	Legge 1/2/4 byte all'indirizzo in %esi e lo scrive nei 1/2/4 byte all'indirizzo in %edi. Se DF è 0, incrementa %edi di 1/2/4, se è 1 lo decrementa.
cmps	Compare Strings	cmps[bwl]	Confronta gli 1/2/4 byte all'indirizzo in %esi (sorgente) con quelli all'indirizzo in %edi (destinatario). Aggiorna i flag così come fa cmp.
scas	Scan String	scas[bwl]	Confronta %al / %ax / %eax (sorgente) con gli 1/2/4 byte all'indirizzo in %edi (destinatario). Aggiorna i flag così come fa cmp.

### Repeat Instruction

Le istruzioni stringa possono essere ripetute senza controllo di programma, usando il prefisso rep.

Istruzione	Nome esteso	Notazione	Comportamento
rep	Unconditional Repeat Instruction	rep [opcode]	Dato n il valore in %ecx, ripete l'operazione opcode n volte, decrementando %ecx fino a 0. Compatibile con lods, stos, movs.
repe	Repeat Instruction if Equal	repe [opcode]	Dato n il valore in %ecx, decrementa %ecx e ripete l'operazione opcode finché 1) %ecx è (ancora) diverso da 0, e 2) gli operandi di questa ripetizione erano uguali. Compatibile con cmps e scas.
repne	Repeat Instruction if Not Equal	repne [opcode]	Dato n il valore in %ecx, decrementa %ecx e ripete l'operazione opcode finché 1) %ecx è (ancora) diverso da 0, e 2) gli operandi di questa ripetizione erano disuguali. Compatibile con cmps e scas.

## 2.8 Altre istruzioni

Istruzione	Nome esteso	Notazione	Comportamento
nop	No Operation	nop	Non cambia lo stato del processore in alcun modo, eccetto per il registro %ip.

Le seguenti istruzioni sono di interesse didattico ma non per le esercitazioni, in quanto richiedono privilegi di esecuzione.

Istruzione	Nome esteso	Notazione	Comportamento
in	Input from Port	in r/i r	Legge da una porta di input ad un registro.
out	Output to Port	out r r/i	Scrive da un registro ad una porta di output.
ins	Input String from Port	ins[bwl]	Legge 1/2/4 byte dalla porta di input indicata in %dx e li scrive nei 1/2/4 byte all'indirizzo in %edi.
outs	Output String to Port	outs[bwl]	Legge 1/2/4 byte all'indirizzo indicato da %esi e li scrive alla porta di output indicata in %dx.
hlt	Halt	hlt	Blocca ogni operazione del processore.



## 3. Sottoprogrammi di utility

Nell'architettura del processore, menzioniamo registri, istruzioni e locazioni di memoria. Quando scriviamo programmi, sfruttiamo però il concetto di *terminale*, un'interfaccia dove l'utente legge caratteri e ne scrive usando la tastiera. Come questo possa avvenire è argomento di altri corsi, dove verranno presentate le *interruzioni*, il *kernel*, e in generale cosa fa un *sistema operativo*.

In questo corso ci limitiamo a sfruttare queste funzionalità tramite del codice ad hoc contenuto in `utility.s`. Queste funzionalità sono fornite come sottoprogrammi, che hanno i loro specifici comportamenti da tenere a mente.

Per utilizzare questi sottoprogrammi, utilizziamo la direttiva

```
.include "../files/utility.s"
```

### 3.1 Terminologia

Con *leggere caratteri da tastiera* si intende che il programma resta in attesa che l'utente prema un tasto sulla tastiera, inviando la codifica di quel tasto al programma.

Con *mostrare a terminale* si intende che il programma stampa un carattere a video.

Con *fare eco* di un carattere si intende che il programma, subito dopo aver letto un carattere da tastiera, lo mostra anche a schermo. Questo è il comportamento interattivo a cui siamo più abituati, ma non è automatico.

Con *ignorare caratteri* si intende che il programma, dopo aver letto un carattere, controlli che questo sia del tipo atteso: se lo è ne fa eco o comunque risponde in modo interattivo, se non lo è ritorna in lettura di un altro carattere, mostrandosi all'utente come se non avesse, appunto, ignorato il carattere precedente.

### 3.2 Caratteri speciali

Avanzamento linea ( *line feed*, LF): carattere `\n`, codifica `0x0A`.

Ritorno carrello ( *carriage return*, RF): carattere `\r`, codifica `0x0D`.

Il significato di questi ha a che vedere con le macchine da scrivere, dove *avanzare alla riga successiva* e *riportare il carrello a sinistra* erano azioni ben distinte.

### 3.3 Sottoprogrammi

Nome	Comportamento
<code>inchar</code>	Legge da tastiera un carattere ASCII e ne scrive la codifica in <code>%a1</code> . Non mostra a terminale il carattere letto.
<code>outchar</code>	Legge la codifica di un carattere ASCII dal registro <code>%a1</code> e lo mostra a terminale.
<code>inbyte</code> / <code>inword</code> / <code>inlong</code>	Legge dalla tastiera 2/4/8 cifre esadecimali (0-9 e A-F), facendone eco e ignorando altri caratteri. Salva quindi il byte/word/long corrispondente a tali cifre in <code>%a1</code> / <code>%ax</code> / <code>%eax</code> .
<code>outbyte</code> / <code>outword</code> / <code>outlong</code>	Legge il contenuto di <code>%a1</code> / <code>%ax</code> / <code>%eax</code> e lo mostra a terminale sottoforma di 2/4/8 cifre esadecimali.
<code>indecimal_byte</code> / <code>indecimal_word</code> / <code>indecimal_long</code>	Legge dalla tastiera fino a 3/5/10 cifre decimali (0-9), o finché non è inserito un <code>\r</code> , facendone eco e ignorando altri caratteri. Interpreta queste come cifre di un numero naturale, e salva quindi il byte/word/long corrispondente in <code>%a1</code> / <code>%ax</code> / <code>%eax</code> .
<code>outdecimal_byte</code> / <code>outdecimal_word</code> / <code>outdecimal_long</code>	Legge il contenuto di <code>%a1</code> / <code>%ax</code> / <code>%eax</code> , lo interpreta come numero naturale e lo mostra a terminale sottoforma di cifre decimali.
<code>outmess</code>	Dato l'indirizzo <code>v</code> in <code>%ebx</code> e il numero <code>n</code> in <code>%cx</code> , mostra a terminale gli <code>n</code> caratteri ASCII memorizzati a partire da <code>v</code> .
<code>outline</code>	Dato l'indirizzo <code>v</code> in <code>%ebx</code> , mostra a terminale i caratteri ASCII memorizzati a partire da <code>v</code> finché non incontra un <code>\r</code> o raggiunge il massimo di 80 caratteri.

<code>inline</code>	Dato l'indirizzo $v$ in <code>%ebx</code> e il numero $n$ in <code>%cx</code> , legge da tastiera caratteri ASCII e li scrive a partire da $v$ finché non è inserito un <code>\r</code> o raggiunge il massimo di $n - 2$ caratteri. Pone poi in fondo i caratteri <code>\r\n</code> . Supporta l'uso di <code>backspace</code> per correggere l'input.
<code>newline</code>	Porta l'output del terminale ad una nuova riga, mostrando i caratteri <code>\r\n</code> .



## 4. Debugger gdb

`gdb` è un debugger a linea di comando che ci permette di eseguire un programma passo passo, seguendo lo stato del processore e della memoria.

Il concetto fondamentale per un debugger è quello di *breakpoint*, ossia un punto del codice dove l'esecuzione dovrà fermarsi. I breakpoints ci permettono di eseguire rapidamente le parti del programma che non sono di interesse e fermarsi ad osservare solo le parti che ci interessano.

Quella che segue è comunque una presentazione sintetica e semplificata. Per altre opzioni e funzionalità del debugger, vedere la documentazione ufficiale o il comando `help`.

### 4.1 Controllo dell'esecuzione

Per istruzione corrente si intende *la prossima da eseguire*. Quando il debugger si ferma ad un'istruzione, si ferma *prima* di eseguirla.

Nome completo	Nome scorciatoia	Formato	Comportamento
<code>frame</code>	<code>f</code>	<code>f</code>	Mostra l'istruzione corrente.
<code>list</code>	<code>l</code>	<code>l</code>	Mostra il sorgente attorno all'istruzione corrente.
<code>break</code>	<code>b</code>	<code>b label</code>	Imposta un breakpoint alla prima istruzione dopo <i>label</i> .
<code>continue</code>	<code>c</code>	<code>c</code>	Prosegue l'esecuzione del programma fino al prossimo breakpoint.
<code>step</code>	<code>s</code>	<code>s</code>	Esegue l'istruzione corrente, fermandosi immediatamente dopo. Se l'istruzione corrente è una <code>call</code> , l'esecuzione si fermerà alla prima istruzione del sottoprogramma chiamato.
<code>next</code>	<code>n</code>	<code>n</code>	Esegue l'istruzione corrente, fermandosi all'istruzione successiva del sottoprogramma corrente. Se l'istruzione corrente è una <code>call</code> , l'esecuzione si fermerà <i>dopo</i> il <code>ret</code> di del sottoprogramma chiamato. Nota: aggiungere una <code>nop</code> dopo ogni <code>call</code> prima di una nuova <code>label</code> .
<code>finish</code>	<code>fin</code>	<code>fin</code>	Continua l'esecuzione fino all'uscita dal sottoprogramma corrente ( <code>ret</code> ). L'esecuzione si fermerà alla prima istruzione dopo la <code>call</code> .
<code>run</code>	<code>r</code>	<code>r</code>	Avvia (o riavvia) l'esecuzione del programma. Chiede conferma.
<code>quit</code>	<code>q</code>	<code>q</code>	Esce dal debugger. Chiede conferma.

I seguenti comandi sono *definiti ad-hoc nell'ambiente del corso*, e non sono quindi tipici comandi di `gdb`.

Nome completo	Nome scorciatoia	Formato	Comportamento
<code>rrun</code>	<code>rr</code>	<code>rr</code>	Avvia (o riavvia) l'esecuzione del programma, senza chiedere conferma.
<code>qquit</code>	<code>qq</code>	<code>qq</code>	Esce dal debugger, senza chiedere conferma.

#### Problemi con next

Si possono talvolta incontrare problemi con il comportamento di `next`, che derivano da come questa è definita e implementata. Il comando `next` distingue i *frame* come le sequenze di istruzioni che vanno da una `label` alla successiva. Il suo comportamento è, in realtà, di continuare l'esecuzione finché non incontra di nuovo una nuova istruzione nello stesso *frame* di partenza.

Questa logica può essere facilmente rotta con del codice come il seguente, dove *non esiste* una istruzione di `punto_1` che viene incontrata dopo la `call`. Quel che ne consegue è che il comando `next` si comporta come `continue`.

```
punto_1:
...
call newline
punto_2:
...
```

Per ovviare a questo problema, è una buona abitudine quella di aggiungere una `nop` dopo ciascuna `call`. Tale `nop`, appartenendo allo stesso *frame* `punto_1`, farà regolarmente sospendere l'esecuzione.

```
punto_1:
...
    call newline
    nop
punto_2:
...
```

## 4.2 Ispezione dei registri

Nome completo	Nome scorciatoia	Formato	Comportamento
info registers	i r	i r	Mostra lo stato di (quasi) tutti i registri. Non mostra separatamente i sotto-registri, come %ax .
info registers	i r	i r reg	Mostra lo stato del registro <i>reg</i> specificato. <i>reg</i> va specificato in minuscolo senza caratteri preposti, per esempio i r eax . Si possono specificare anche sotto-registri, come %ax , e più registri separati da spazio.

`gdb` supporta viste alternative con il comando `layout` che mettono più informazioni a schermo. In particolare, `layout regs` mostra l'equivalente di `i r` e `l`, evidenziando gli elementi che cambiano ad ogni step di esecuzione.

## 4.3 Ispezione della memoria

Nome completo	Nome scorciatoia	Formato	Comportamento
x	x	x/ N FU addr	Mostra lo stato della memoria a partire dall'indirizzo <i>addr</i> , per le <i>N</i> locazione di dimensione <i>U</i> e interpretate con il formato <i>F</i> . Comando con memoria, i valori di <i>N</i> , <i>F</i> e <i>U</i> possono essere omessi (insieme allo / ) se uguali a prima.

Il comando `x` sta per *examine memory* , ma differenza degli altri non ha una versione estesa.

Il parametro *N* si specifica come un numero intero, il valore di default (all'avvio di `gdb` ) è 1.

Il parametro *F* può essere

- `x` per esadecimale
- `d` per decimale
- `c` per ASCII
- `t` per binario
- `s` per stringa delimitata da `0x00`

Il valore di default (all'avvio di `gdb` ) è `x` .

Il parametro *U* può essere

- `b` per byte
- `h` per word (2 byte)
- `w` per long (4 byte)

Il valore di default (all'avvio di `gdb` ) è `h` .

L'argomento *addr* può essere espresso in diversi modi, sia usando label che registri o espressioni basate su aritmetica dei puntatori. Per esempio:

- letterale esadecimale: `x 0x56559066`
- label: `x &label`
- registro puntatore: `x $esi`
- registro puntatore e registro indice: `x (char*)$esi + $ecx`

Notare che nell'ultimo caso, dato che ci si basa su aritmetica dei puntatori, il tipo all'interno del cast determina la *scala* , ossia la dimensione di ciascuna delle `$ecx` locazioni del vettore da saltare. Si può usare `(char*)` per 1 byte, `(short*)` per 2 byte, `(int*)` per 4 byte.

Un alternativa a questo è lo scomporre, anche solo temporaneamente, le istruzioni con indirizzamento complesso. Per esempio, si può sostituire `movb (%esi, %ecx), %al` con `lea (%esi, %ecx), %ebx` seguita da `movb (%ebx), %al` , così che si possa eseguire semplicemente `x $ebx` nel debugger.

## 4.4 Gestione dei breakpoints

Oltre a crearli, i breakpoint possono anche essere rimossi o (dis)abilitati. Questi comandi si basano sulla conoscenza dell' *id* di un breakpoint: questo viene stampato quando un breakpoint viene creato o raggiunto durante l'esecuzione, oppure si possono ristampare tutti usando `info b` .

Nome completo	Nome scorciatoia	Formato	Comportamento
info breakpoints	info b	info b [ <i>id</i> ]	Stampa informazioni sul breakpoint <i>id</i> , o tutti se l'argomento è omissso.
disable breakpoints	dis	dis [ <i>id</i> ]	Disabilita il breakpoint <i>id</i> , o tutti se l'argomento è omissso.
enable breakpoints	en	en [ <i>id</i> ]	Abilita il breakpoint <i>id</i> , o tutti se l'argomento è omissso.
delete breakpoints	d	d [ <i>id</i> ]	Rimuove il breakpoint <i>id</i> , o tutti se l'argomento è omissso.

## Conditional Breakpoints

In alcuni casi, la complessità del programma, l'uso intensivo di sottoprogrammi o lunghi loop possono rendere molto lungo trovare il punto giusto dell'esecuzione. A questo scopo, è possibile definire dei *breakpoint condizionali* , per far sì che l'esecuzione si interrompa a tale breakpoint solo se la condizione è verificata.

Nome completo	Nome scorciatoia	Formato	Comportamento
condition	cond	cond <i>id</i> <i>cond</i>	Imposta la condizione <i>cond</i> per il breakpoint <i>id</i> .

La sintassi per una condizione è in “stile C”, come il comando `x` . Alcuni esempi di questa sintassi:

- `cond 2 $a1==5` per far sì che l'esecuzione si fermi al breakpoint 2 solo se il registro `a1` contiene il valore 5,
- `cond 2 (short *)$edi==-5` per far sì che l'esecuzione si fermi al breakpoint 2 solo se il registro `edi` contiene l'indirizzo di una word di valore -5,
- `cond 2 (int *)&count!=0` per far sì che l'esecuzione si fermi al breakpoint 2 solo se la locazione di 4 byte a partire da `count` contiene un valore diverso da 0,
- 

Fare attenzione alle conversioni automatiche di rappresentazione: quando si usa la rappresentazione decimale, `gdb` interpreta automaticamente i valori come interi. Una condizione come `cond 2 $a1==128` , per quanto accettata dal debugger, sarà sempre falsa perché la codifica `0x80` è interpretata in decimale come l'intero -128 , mai come il naturale 128 . È quindi una buona idea usare la notazione esadecimale in casi del genere, cioè quando il bit più significativo è 1.

Una feature disponibile in molti IDE è quello di creare dipendenze tra breakpoint, cioè abilitare un breakpoint solo se è stato prima colpito un altro. Questo però è [fin troppo ostico](#) da fare in `gdb` .

## Watchpoints

I watchpoint sono come dei breakpoint ma per dati (registri e memoria), non per il codice. Si creano indicando l'espressione del dato da controllare. Si gestiscono *con gli stessi comandi per i breakpoint* .

Nome completo	Nome scorciatoia	Formato	Comportamento
watchpoint	watch	watch <i>expr</i>	Imposta un watchpoint per l'espressione <i>expr</i> .
info watchpoints	info wat	info wat [ <i>id</i> ]	Stampa informazioni sul watchpoint <i>id</i> , o tutti se l'argomento è omissso.
disable breakpoints	dis	dis [ <i>id</i> ]	Disabilita il breakpoint o watchpoint <i>id</i> , o tutti se l'argomento è omissso.
enable breakpoints	en	en [ <i>id</i> ]	Abilita il breakpoint o watchpoint <i>id</i> , o tutti se l'argomento è omissso.
delete breakpoints	d	d [ <i>id</i> ]	Rimuove il breakpoint o watchpoint <i>id</i> , o tutti se l'argomento è omissso.

Un watchpoint richiede la specifica di un registro o locazione nella stessa notazione “stile C” del comando `x` , e interrompe l'esecuzione quando tale valore cambia. Per esempio, `watch $eax` crea un watchpoint che interrompe l'esecuzione ogni volta che `eax` cambia valore.



## 5. Tabella ASCII

Dalla tabella seguente sono esclusi caratteri non-stampabili che non sono di nostro interesse.

Codifica binaria	Codifica decimale	Codifica esadecimale	Carattere
0000 0000	00	0x00	\0
0000 1000	08	0x08	backspace
0000 1010	10	0x0A	\n , Line Feed
0000 1101	13	0x0D	\r , Carriage Return
0010 0000	32	0x20	space
0010 0001	33	0x21	!
0010 0010	34	0x22	"
0010 0011	35	0x23	#
0010 0100	36	0x24	\$
0010 0101	37	0x25	%
0010 0110	38	0x26	&
0010 0111	39	0x27	'
0010 1000	40	0x28	(
0010 1001	41	0x29	)
0010 1010	42	0x2A	*
0010 1011	43	0x2B	+
0010 1100	44	0x2C	,
0010 1101	45	0x2D	-
0010 1110	46	0x2E	.
0010 1111	47	0x2F	/
0011 0000	48	0x30	0
0011 0001	49	0x31	1
0011 0010	50	0x32	2
0011 0011	51	0x33	3
0011 0100	52	0x34	4
0011 0101	53	0x35	5
0011 0110	54	0x36	6
0011 0111	55	0x37	7
0011 1000	56	0x38	8
0011 1001	57	0x39	9
0011 1010	58	0x3A	:
0011 1011	59	0x3B	;
0011 1100	60	0x3C	<
0011 1101	61	0x3D	=
0011 1110	62	0x3E	>
0011 1111	63	0x3F	?
0100 0000	64	0x40	@
0100 0001	65	0x41	A
0100 0010	66	0x42	B
0100 0011	67	0x43	C
0100 0100	68	0x44	D
0100 0101	69	0x45	E
0100 0110	70	0x46	F
0100 0111	71	0x47	G
0100 1000	72	0x48	H
0100 1001	73	0x49	I
0100 1010	74	0x4A	J
0100 1011	75	0x4B	K
0100 1100	76	0x4C	L
0100 1101	77	0x4D	M
0100 1110	78	0x4E	N
0100 1111	79	0x4F	O
0101 0000	80	0x50	P
0101 0001	81	0x51	Q

0101 0010	82	0x52	R
0101 0011	83	0x53	S
0101 0100	84	0x54	T
0101 0101	85	0x55	U
0101 0110	86	0x56	V
0101 0111	87	0x57	W
0101 1000	88	0x58	X
0101 1001	89	0x59	Y
0101 1010	90	0x5A	Z
0101 1011	91	0x5B	[
0101 1100	92	0x5C	\
0101 1101	93	0x5D	]
0101 1110	94	0x5E	^
0101 1111	95	0x5F	_
0110 0000	96	0x60	`
0110 0001	97	0x61	a
0110 0010	98	0x62	b
0110 0011	99	0x63	c
0110 0100	100	0x64	d
0110 0101	101	0x65	e
0110 0110	102	0x66	f
0110 0111	103	0x67	g
0110 1000	104	0x68	h
0110 1001	105	0x69	i
0110 1010	106	0x6A	j
0110 1011	107	0x6B	k
0110 1100	108	0x6C	l
0110 1101	109	0x6D	m
0110 1110	110	0x6E	n
0110 1111	111	0x6F	o
0111 0000	112	0x70	p
0111 0001	113	0x71	q
0111 0010	114	0x72	r
0111 0011	115	0x73	s
0111 0100	116	0x74	t
0111 0101	117	0x75	u
0111 0110	118	0x76	v
0111 0111	119	0x77	w
0111 1000	120	0x78	x
0111 1001	121	0x79	y
0111 1010	122	0x7A	z
0111 1011	123	0x7B	{
0111 1100	124	0x7C	
0111 1101	125	0x7D	}
0111 1110	126	0x7E	~

From <https://en.wikipedia.org/wiki/ASCII>

## 6. Ambiente d'esame e i suoi script

Qui di seguito sono documentati gli script dell'ambiente. I principali sono `assembler.ps1` e `debug.ps1`, il cui uso è mostrato nelle esercitazioni. Gli script `run-test.ps1` e `run-tests.ps1` sono utili per automatizzare i test, il loro uso è del tutto opzionale.

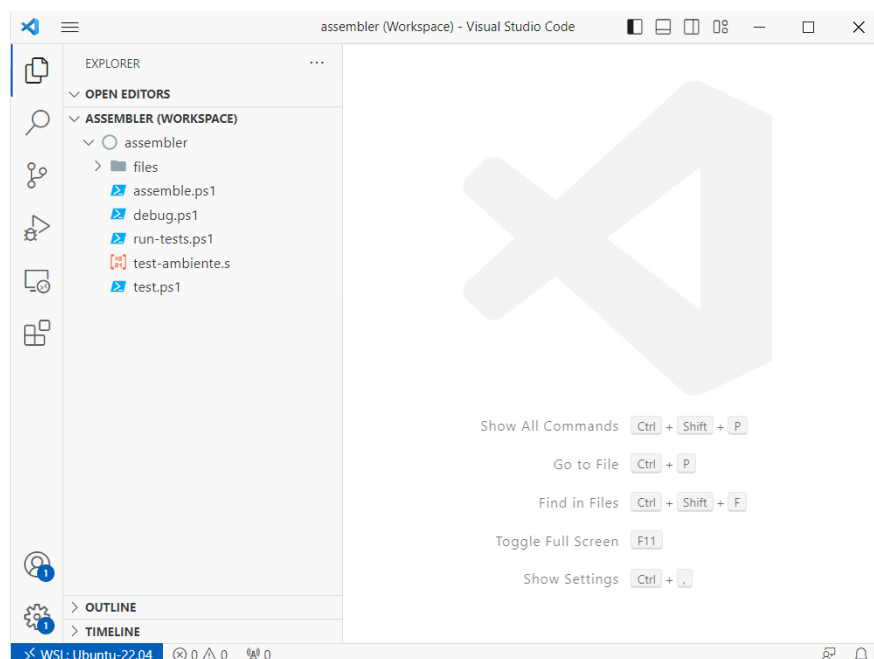
### 6.1 Aprire l'ambiente

Sulle macchine all'esame (o sulla propria, se si seguono tutti i passi indicati nel pacchetto di installazione) troverete una cartella `C:/reti_logiche` con contenuto come da figura.

- assembler
- dispense
- verilog
- assembler.code-workspace
- Istruzioni.txt
- verilog.code-workspace

Facendo doppio click sul file `assembler.code-workspace` verrà lanciato VS Code, collegandosi alla macchina virtuale WSL e la cartella di lavoro `C:/reti_logiche/assembler`.

La finestra VS Code che si aprirà sarà simile alla seguente.



Nell'angolo in basso a sinistra, `WSL: Ubuntu-22.04` sta a indicare che l'editor è correttamente connesso alla macchina virtuale.

I file e cartelle mostrati nell'immagine sono quelli che ci si deve aspettare dall'ambiente vuoto.

In caso si trovino file in più all'esame, si possono *cancellare*.

Il file `test-ambiente.s` è un semplice programma per verificare che l'ambiente funzioni. Il contenuto è il seguente:

```
.include "../files/utility.s"

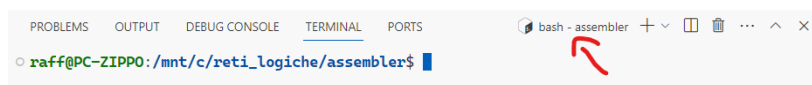
.data
messaggio: .ascii "Ok.\n"

.text
_main:
    nop
    lea messaggio, %ebx
    call outline
    ret
```

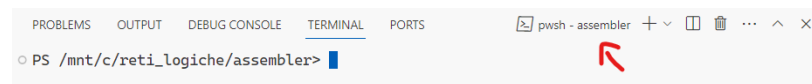
## 6.2 Il terminale Powershell

Per aprire un terminale in VS Code possiamo usare Terminale -> Nuovo Terminale. Per eseguire gli script dell'ambiente c'è bisogno di aprire un terminale *Powershell*. La shell standard di Linux, *bash*, non è in grado di eseguire questi script.

Non così:



Ma così:



Per cambiare shell si può usare il bottone + sulla sinistra, o lanciare il comando `pwsh` senza argomenti.

Se si preferisce, in VS Code si può aprire un terminale anche come tab dell'editor, o spostandolo al lato anziché in basso.

### 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](#).

## 6.3 Eseguire gli script

Gli script forniti permettono di assemblare, debuggare e testare il proprio programma. È importante che vengano eseguiti senza cambiare cartella, cioè non usando il comando `cd` o simili. Ricordarsi anche dei `./`, necessari per indicare al terminale che i file indicati vanno cercati nella cartella corrente.

Il tasto `tab` della tastiera invoca l'autocompletamento, che aiuta ad assicurarsi di inserire percorsi corretti.

Si ricorda inoltre di salvare il file sorgente prima di provare ad eseguire script.

### assemble.ps1

```
PS /mnt/c/reti_logiche/assembler> ./assemble.ps1 mio_programma.s
```

Questo script assembla un sorgente assembler in un file eseguibile. Lo script controlla prima che il file passato non sia un eseguibile, invece che un sorgente. Poi, il sorgente viene assemblato usando `gcc` ad includendo il sorgente `./files/main.c`, che si occupa di alcune impostazioni del terminale.

### debug.ps1

```
PS /mnt/c/reti_logiche/assembler> ./debug.ps1 mio_programma
```



Questo script lancia il debugger per un programma. Lo script controlla prima che il file passato non sia un sorgente, invece che un eseguibile. Poi, il debugger gdb viene lanciato con il programma dato, includendo le definizioni e comandi iniziali in `./files/gdb_startup`. Questi si occupano di definire i comandi `qquit` e `rrun` (non chiedono conferma), creare un breakpoint in `_main` e avviare il programma fino a tale breakpoint (così da saltare il codice di setup di `./files/main.c`).

### **run-test.ps1**

```
PS /mnt/c/reti_logiche/assembler> ./run-test.ps1 mio_programma input.txt output.txt
```

Lancia un eseguibile usando il contenuto di un file come input, e ne opzionalmente ne stampa l'output su file. Lo script fa ridirezione di input/output, con alcuni controlli. Tutti i caratteri del file di input verranno visti dal programma come se digitati da tastiera, inclusi i caratteri di fine riga.

### **run-tests.ps1**

```
PS /mnt/c/reti_logiche/assembler> ./run-tests.ps1 mio_programma cartella_test
```

Testa un eseguibile su una serie di coppie input-output, verificando che l'output sia quello atteso. Stampa riassuntivamente e per ciascun test se è stato passato o meno.

Lo script prende ciascun file di input, con nome nella forma `in_*.txt`, ed esegue l'eseguibile con tale input. Ne salva poi l'output corrispondente nel file `out_*.txt`. Confronta poi `out_*.txt` e `out_ref_*.txt`: il test è passato se i due file coincidono. Nel confronto, viene ignorata la differenza fra le sequenze di fine riga `\r\n` e `\n`.



## 7. Problemi comuni

Questa sezione include problemi che è frequente incontrare.

Come regola generale, in sede d'esame rispondiamo a tutte le domande relative a problemi di questo tipo e aiutiamo a proseguire - perché sono relative all'ambiente d'esame e non ai concetti *oggetto* d'esame.

Per altre domande, si può sempre contattare per email o Teams.

### 7.1 Setup dell'ambiente

#### 1. Ho trovato un ambiente assembler per Mac su Github, ma ho problemi ad usarlo

Non abbiamo fatto noi quell'ambiente, non sappiamo come funziona e non offriamo supporto su come usarlo.

#### 2. Ho trovato un ambiente basato su DOS, usato precedentemente all'esame, ma ho problemi ad usarlo

Ha probabilmente incontrato uno dei tanti motivi per cui l'ambiente basato su DOS è stato abbandonato. Questi problemi sono al più *aggirabili*, non *risolvibili*.

#### 3. Lanciando il file `assemble.code-workspace`, mi appare un messaggio del tipo **Unknown distro: Ubuntu**

Il file `assemble.code-workspace` cerca di lanciare via WSL la distro chiamata Ubuntu, senza alcuna specifica di versione. Nel caso la vostra installazione sia diversa, andrà modificato il file. Da un terminale Windows, lanciare `wsl --list -v`, dovreste ottenere una stampa del tipo

```
PS C:\Users\raffa> wsl --list -v
NAME                STATE              VERSION
* Ubuntu            Stopped            2
  Ubuntu-22.04       Stopped            2
```

La parte importante è la colonna NAME dell'immagine che vogliamo usare per l'ambiente assembler. Modificare il file `assemble.code-workspace` con un editor di testo (notepad o VS Code stesso, stando attenti ad aprirlo come file di testo e non come workspace) sostituendo tutte le occorrenze di `wsl+ubuntu` con `wsl+NOME-DELLA-DISTRO`. Per esempio, se volessi utilizzare l'immagine Ubuntu-22.04, sostituirei con `wsl+Ubuntu-22.04`.

#### 4. Sto utilizzando una sistema Linux desktop, come uso l'ambiente senza virtualizzazione?

Il file `assemble.code-workspace` fa tre cose

- Aprire VS Code nella macchina virtuale WSL
- Aprire la cartella assembler in tale ambiente
- Impostare `pwsh` come terminale default

È possibile fare manualmente gli step 2 e 3, o modificare `assemble.code-workspace` per non fare lo step 1. Per seguire questa seconda opzione, eliminare la riga con `"remoteAuthority":`, e modificare il percorso dopo `"uri":` perché sia semplicemente un percorso sul proprio disco, per esempio `"uri": "/home/raff/reti_logiche/_assembler"`.

## 7.2 Uso dell'ambiente

### 5. Se premo *Run* su VS Code non viene lanciato il programma

Non è così che si usa l'ambiente di questo corso. Si deve usare un terminale, assemblare con `./assemble.ps1` `programma.s` e lanciare con `./programma`.

### 6. Provando a lanciare `./assemble.ps1` `programma.s` ricevo un errore del tipo `./assemble.ps1: line 1: syntax error near unexpected token`

State usando la shell da terminale sbagliata, `bash` invece che `pwsh`. Aprire un terminale Powershell da VS Code o utilizzare il comando `pwsh`.

### 7. Provando ad assemblare ricevo un warning del tipo `warning: creating DT_TEXTREL in a PIE`

Sostituire il file `assemble.ps1` con quello contenuto nel pacchetto più recente tra i file del corso. Oppure modificare manualmente il file, alla riga 29, da

```
gcc -m32 -o ...
```

```
a
```

```
gcc -m32 -no-pie -o ...
```

Riprovare quindi a riassemblare. Se il warning non sparisce, scrivermi. Allegando il sorgente.

### 8. Ho modificato il codice per correggere un errore, ma quando assemblo e eseguo il codice, continuo a vedere lo stesso errore.

Controllare di aver salvato il file. In alto, nella barra delle tab, VS Code mostra un pallino pieno, al posto della X per chiedere la tab, per i file modificati e non salvati.

## **Parte II**

# **Documentazione Verilog**



## 8. 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 è [www.chipverif.com](http://www.chipverif.com).





## 9. Operatori

### 9.1 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
2. formato di rappresentazione
3. 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` ).

#### 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`.

### 9.2 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.

### 9.3 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.

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

Operatore	Tipo di operazione
<code>&amp;</code>	and
<code>~&amp;</code>	nand
<code>\ </code>	or
<code>~\ </code>	nor
<code>^</code>	xor
<code>~^</code>	xnor
<code>~</code>	not

#### 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:

1. passare al layout a QWERTY internazionale
2. 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 (risolve altri problemi come il backtick ` e rende più semplici da scrivere `[]{};`). [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](#).

### 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
<code>&amp;</code>	and
<code>~&amp;</code>	nand
<code>\ </code>	or
<code>~\ </code>	nor
<code>^</code>	xor
<code>~^</code>	xnor

## 9.4 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*).

## 9.5 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;
```

**Raggruppare fili non ha nessun costo**

Questo operatore corrisponde, circuitalmente, al semplice raggruppare dei fili assieme. 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`.

**Operatore di replicazione  $N\{\dots\}$** 

L'operatore di ripetizione 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 = {4{x}}; // equivalente a {x, x, x, x}
```

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

**9.6 Operazioni comuni****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 = {4{x_8[7]}, x_8};
```

**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 = {2{x[7]}, x[7:2]};
```



# 10. 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.

## 10.1 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
```

### 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.

## 10.2 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.

## 10.3 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(...), ...
);
```

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
```

## 10.4 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 `case`.

```
1 module and (x, y, z);
2     input x, y;
3     output z;
4     assign #1 z = tabella_verita({a, b});
5
6     function tabella_verita;
7         input [1:0] ab;
8         casex(ab)
9             2'b00: tabella_verita = 1'b0;
10            2'b01: tabella_verita = 1'b0;
11            2'b10: tabella_verita = 1'b0;
12            2'b11: tabella_verita = 1'b1;
13        endcase
14    endfunction
15 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.

## 10.5 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.**

In una rete che implementa una data tabella di verità l'uscita è un valore specifico e costante, per un multiplexer è il valore di uno degli ingressi. 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.

## 10.6 Reti parametrizzate

In un module si possono definire parametri per generalizzare la rete. In particolare, questo è frequentemente utilizzato in `reti_standard.v` per fornire reti il cui dimensionamento è da specificare.

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 instanziamo 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 è mutabile! 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.



# 11. 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`.

## 11.1 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.

## 11.2 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`.

## 11.3 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

1. come si comportano i registri al reset,
2. come si comportano i registri al fronte positivo del clock .

## 11.4 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 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.

## 11.5 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
    case(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

```

## 11.6 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 assegni  $<=$  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.



# 12. 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.

## 12.1 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, per esempio 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 di 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.

## 12.2 Waveform e debugging

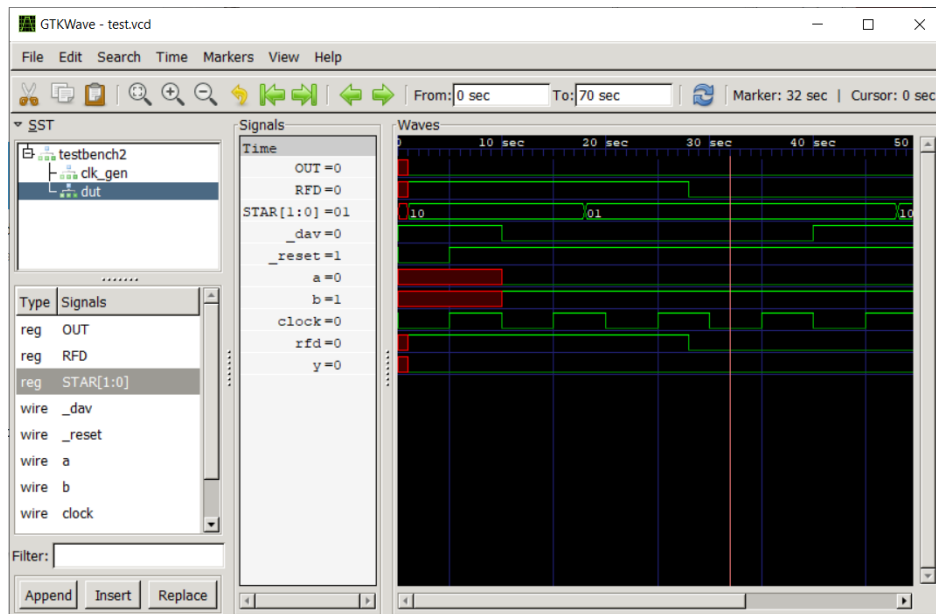
La simulazione genera un file waveform.vcd contenente l'evoluzione di tutti i fili e registri nella simulazione. Questo file è prodotto grazie alle seguenti righe, incluse in tutte le testbench:

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

Con questo file possiamo studiare l'evoluzione della rete e trovare eventuali errori. Per analizzarlo, usiamo GTKWave, 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.

### 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 rosso 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, composti se 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.

### 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.

## 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:

- usare due terminali, uno dedicato a `iverilog` e `vvp`, l'altro a `gtkwave` ;
- 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 }
```

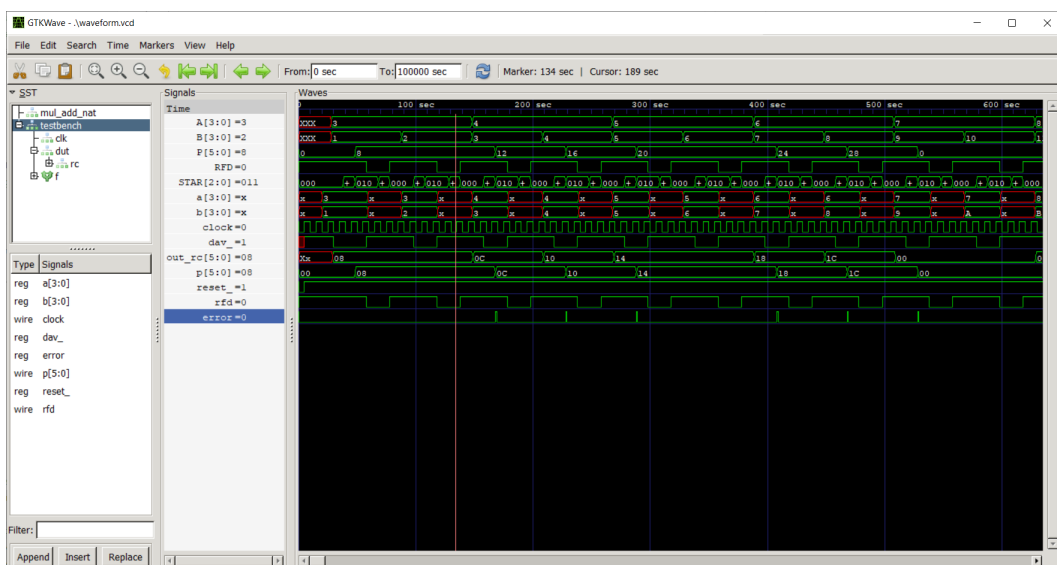
L'operatore è documentato [qui](#).

## 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 ed 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**

# **Uso di VS Code**



# 13. 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`.

## 13.1 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.

## 13.2 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 <code>^</code> )	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.

## 13.3 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.