Esempio di progetto: Interfaccia IFMAX

L'interfaccia IFMAX riceve in continuazione da un bus esterno $\texttt{EXTDATA}_{0-15}$ dati numerici paralleli a 16 bit in complemento a 2, sincronizzati a un clock esterno `EXTCLK` con frequenza 1 MHz. Su comando dalla CPU PD32, IFMAX esamina un blocco di 256 dati consecutivi, calcola il massimo dei valori presenti nel blocco, e trasmette il risultato alla CPU. A sua volta, la CPU mantiene in una variabile `VMINMAX` il minimo dei valori ricevuti.

Progettare l'interfaccia IFMAX e codificare il relativo software di pilotaggio.

* * * * * * * *

1 Generalità

L'interfaccia IFMAX è evidentemente interposta tra il bus di una CPU PD32 e il mondo esterno (Fig. 1): essa accetta in ingresso un gruppo di 16 linee digitali $\texttt{EXTDATA}_{0-15}$ che trasportano valori numerici binari in complemento a 2 e una linea digitale `EXTCLK` che trasporta un clock di sincronizzazione per i dati suddetti; poiché non è richiesto alcun accesso diretto alla memoria (DMA), la comunicazione con la CPU può essere limitata al solo bus di input/output.

Cominciamo col partizionare il progetto in blocchi funzionali (Fig. 2), che andremo successivamente a progettare con il dovuto dettaglio; tale partizionamento è abbastanza generale, e non dipende in maniera sostanziale dalle operazioni richieste a IFMAX. Poiché non vi sono specifiche sufficientemente precise e dettagliate sul comportamento di $\texttt{EXTDATA}_{0-15}$ rispetto a `EXTCLK`, avremo bisogno in primo luogo di un *blocco di condizionamento* dei segnali di ingresso, avente le funzioni di isolare adeguatamente i circuiti di IFMAX dal mondo esterno, di bufferizzare opportunamente i dati in ingresso e di sincronizzarli nella maniera più conveniente rispetto al clock esterno; tale blocco produce in uscita delle repliche dei segnali di ingresso, in particolare: il condizionamento di $\texttt{EXTDATA}_{0-15}$ produce le linee dati $\texttt{XDATA}_{0-15}$, mentre il condizionamento di `EXTCLK` produce un clock `XCLK` che viene distribuito all'interno di IFMAX.

Il complesso delle operazioni richieste a IFMAX viene eseguito all'interno del *sottosistema di calcolo* (SCA), mentre la sequenza delle operazioni suddette viene governata dal *sottosistema di controllo* (SCO); la temporizzazione di entrambi i sottosistemi viene prelevata da `XCLK`, mentre i due sottosistemi comunicano tra di loro mediante due gruppi di segnali, per il momento non meglio specificati:

- *control*, comandi generati dal sottosistema di controllo che governano le operazioni del sottosistema di calcolo,
- *status*, informazioni di stato generate dal sottosistema di calcolo e utilizzate dal sottosistema di controllo per prendere le dovute decisioni.

I dati $\texttt{XDATA}_{0-15}$ utilizzati dal sottosistema di calcolo, una volta processati, producono un risultato che, emesso sulle linee $\texttt{VMAX}_{0-15}$, viene trasmesso al blocco standard di interfaccia PD32 che provvederà a sua volta ad inoltrarlo al bus di sistema; tale blocco ha inoltre le funzioni di:

- trasmettere al resto di IFMAX il segnale di `RESET` per l'inizializzazione a freddo, ossia all'avviamento dell'intero sistema;
- notificare al sottosistema di controllo l'inizio delle operazioni di IFMAX mediante il segnale `STARD`;

[0] Tema assegnato come prova di esonero nel corso di Calcolatori Elettronici (II modulo) il 28 giugno 2002.
Autore: P. Marincola (14 luglio 2002)
Versione: 1.02 (15 giugno 2003)

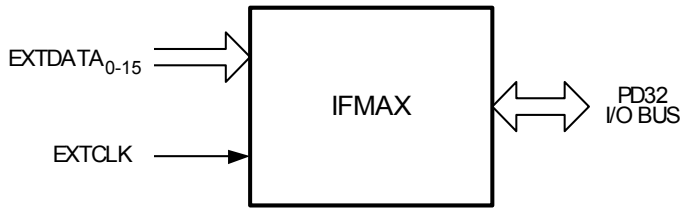


Fig. 1: Interfaccia IFMAX.

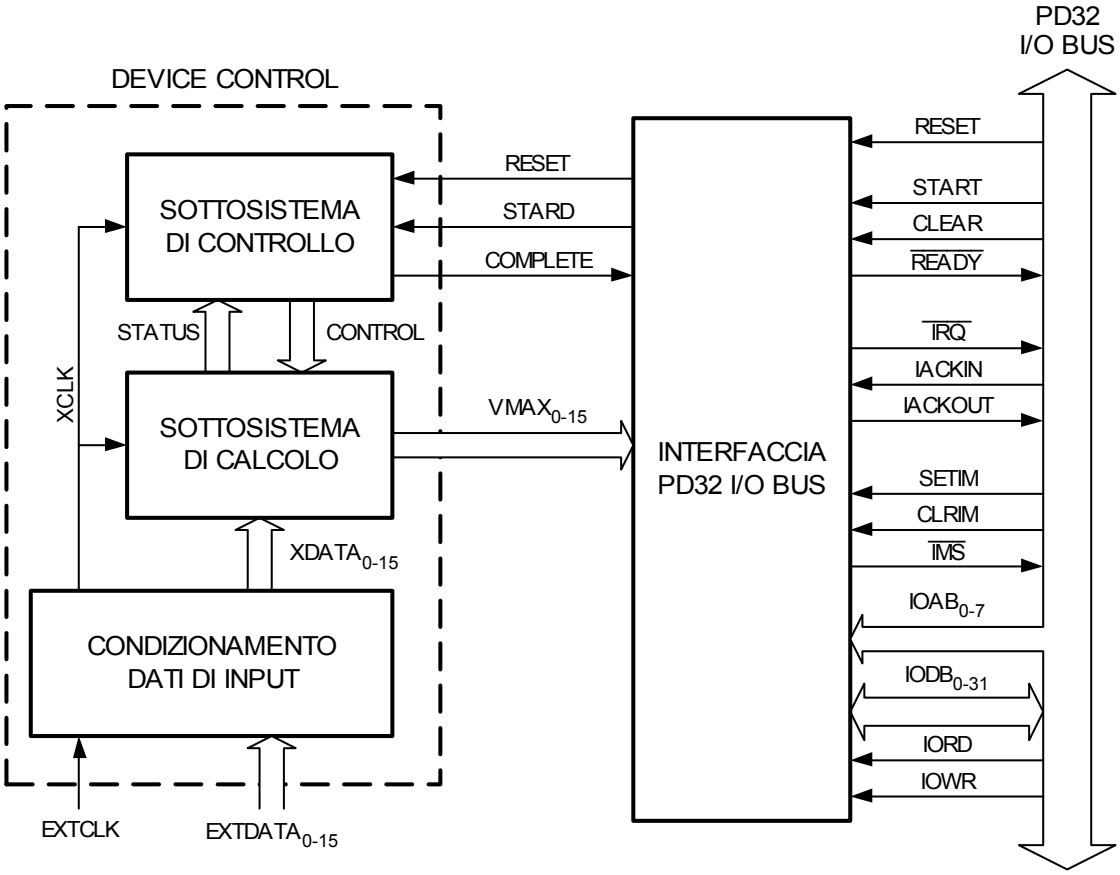


Fig. 2: Partizionamento di IFMAX in blocchi funzionali.

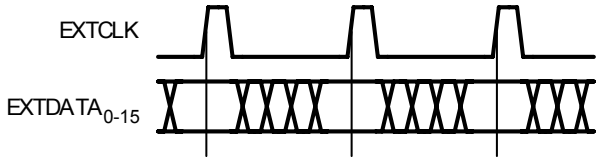


Fig. 3: Relazioni temporali tra clock e dati esterni.

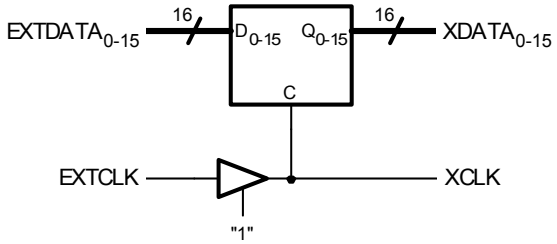


Fig. 4: Circuito di condizionamento.

- ricevere dallo stesso sottosistema l'avviso di completamento delle operazioni mediante il segnale `COMPLETE`.

È il caso di osservare come l'insieme del blocco di condizionamento, del sottosistema di controllo e del sottosistema di calcolo costituisca nel suo complesso il macromodulo noto come *Device Control*.

2 Condizionamento dei segnali

Come accennato in precedenza, non vi sono specifiche precise sulle relazioni temporali tra i dati esterni $\texttt{EXTDATA}_{0-15}$ e il clock `EXTCLK`; in tali circostanze, l'unica assunzione che è ragionevole adottare è quella che richiede ai dati di essere stabili entro un piccolo intorno temporale del fronte attivo (tipicamente il fronte di salita) del clock, come illustrato in Fig. 3. In tali condizioni, è sufficiente far transitare i dati entro un registro per garantire la stabilità dei dati stessi all'interno dell'intero periodo di clock; nel contempo, un semplice buffer tri-state[1] sempre abilitato è sufficiente a bufferizzare lo stesso segnale di clock in maniera che il carico ad esso applicato dai circuiti di IFMAX non influisca sulla qualità del segnale (Fig. 4). Le temporizzazioni del circuito sono visibili in Fig. 5, dove si può osservare agevolmente non solo come i dati vengano stabilizzati per l'intero periodo di clock, ma anche come il leggero ritardo introdotto dal tri-state non pregiudichi il corretto funzionamento del circuito.

3 Sottosistema di calcolo

Il flow-chart generico del processo di calcolo che IFMAX è chiamata ad eseguire, a meno delle ulteriori operazioni necessarie all'avvio del processo e alla sua conclusione, è illustrato in Fig. 6. Il sottosistema di calcolo, nel corso delle normali operazioni, deve pertanto essere in grado di eseguire le seguenti operazioni:

1. mantenere memoria del valore $V_{\max}$ affinché questo possa essere utilizzato nei successivi confronti e possa essere letto dalla CPU alla fine delle operazioni,

[1] I buffer tri-state sono specificamente progettati per avere un alto valore di fan-out con una bassa impedenza di uscita, e sono pertanto in grado di pilotare carichi anche molto elevati senza apprezzabile degradazione del segnale.

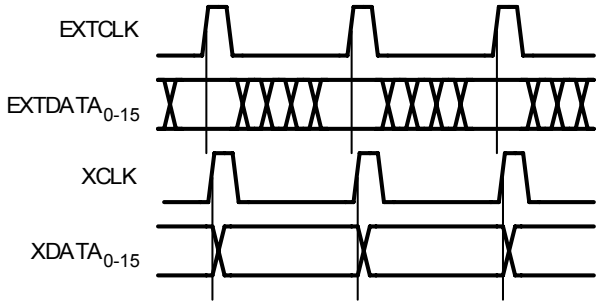


Fig. 5: Temporizzazioni del circuito di condizionamento.

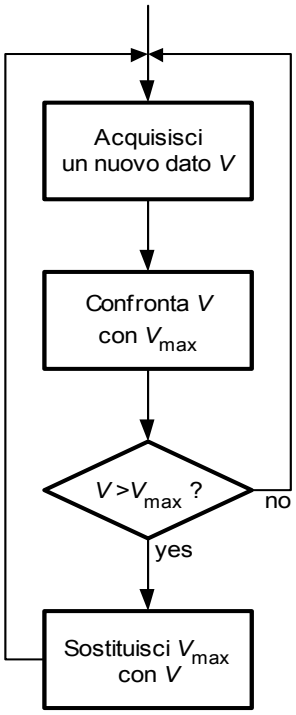


Fig. 6: Operazioni del sottosistema di calcolo.

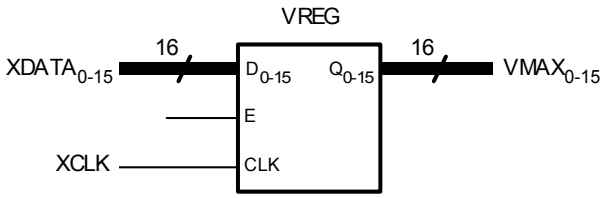


Fig. 7: Registro per la memorizzazione di $V_{\max}$.

2. confrontare il valore $V_{\max}$ corrente con il valore V presente sulle linee `XDATA`,
3. aggiornare $V_{\max}$ quando $V > V_{\max}$.

Un'ulteriore funzione che deleghiamo al sottosistema di calcolo, ma i cui risultati vengono utilizzati essenzialmente dal sottosistema di controllo, è la seguente:

4. mantenere un conteggio del numero di dati di ingresso sottoposti ad elaborazione — tale conteggio verrà poi utilizzato per determinare la fine del ciclo di operazioni.

3.1 Memorizzazione del risultato

L'operazione (3) richiede evidentemente che $V_{\max}$ venga memorizzato in un registro a 16 bit, che chiameremo VREG. L'ingresso a tale registro è costituito dalle linee `XDATA`, poiché da esse proviene l'informazione che dovrà essere eventualmente catturata; inoltre, poiché l'aggiornamento può avvenire solo quando si presenta un nuovo dato di ingresso V, e questo si presenta solo ad ogni ciclo del clock `XCLK`, sarà questo segnale ad essere utilizzato come clock del registro stesso; infine, dal momento che la cattura delle informazioni può non avvenire ad ogni ciclo di clock, VREG dovrà essere dotato di un ingresso `E` di abilitazione (sincrona) al caricamento. Il circuito corrispondente appare in Fig. 7.

3.2 Comparazione dei dati

L'operazione (1) richiede l'impiego di un modulo comparatore in complemento a 2 a 16 bit, che chiameremo VCOMP. Poiché i moduli comparatori standard operano su numeri assoluti, VCOMP dovrà essere appositamente progettato — e di questo ci occuperemo più avanti. Per il momento, tuttavia, basta stabilire che VCOMP dovrà accettare due gruppi di ingressi $\mathtt{A}_{0-15}$, $\mathtt{B}_{0-15}$ per i valori da confrontare, e dovrà generare una singola linea di uscita `AGTB` (*A greater than B*) che dovrà essere attiva quando il valore presente sulle linee $\mathtt{A}_{0-15}$ è numericamente maggiore del valore presente sulle linee $\mathtt{B}_{0-15}$ (Fig. 8). Il comparatore, che non deve tener memoria di alcuna informazione e che sarà dunque un circuito puramente combinatorio, accetta come ingresso `A` il valore V e come ingresso `B` il valore $V_{\max}$; la sua uscita `AGTB` viene utilizzata per abilitare il caricamento del registro VREG (operazione 2). Il circuito corrispondente appare in Fig. 9. (È il caso di osservare che, se sarà necessario per semplificare i circuiti di VCOMP, potremo sostituire la sua uscita `AGTB` con un'uscita `AGEB` (*A greater than or equal to B*) che diventi attiva non già quando $A > B$ ma quando $A \geq B$: in tal caso, VREG verrebbe caricato non solo quando $A > B$, ma anche quando $A = B$, ed è evidente che ciò non altererebbe in alcun modo la correttezza del comportamento complessivo di IFMAX.)

Un esempio di comportamento del circuito è illustrato nelle temporizzazioni di Fig. 10. Supponiamo che, nel corso delle operazioni, il valore contenuto in VREG sia pari a -18 e che la sequenza numerica che appare su `XDATA` sia -35, -7, -105, 24, 61, 15 negli intervalli di clock rispettivamente indicati come $t_1, ..., t_6$. In primo luogo, occorre osservare come, ad ogni commutazione delle linee `XDATA` o `VMAX`, il modulo VCOMP, essendo un circuito puramente combinatorio, potrà produrre delle uscite spurie

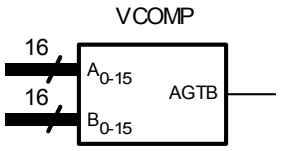


Fig. 8: Modulo comparatore in complemento a 2.

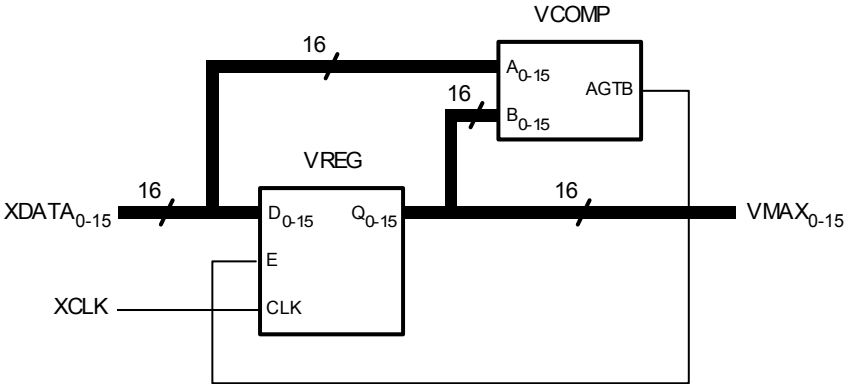


Fig. 9: Circuito per il calcolo e l'aggiornamento di $V_{\max}$.

(dovute alle cosiddette *àlee* e indicate in Fig. 10 come una sequenza di impulsi di brevissima durata); tuttavia, poiché l'uscita `AGTB` di VCOMP viene campionata solo alla fine del ciclo di clock, tali uscite spurie non provocano alcun inconveniente[2].

Nell'intervallo t_1, VCOMP vede $A = -35$, $B = -18$ e dunque la sua uscita `AGTB` è bassa, per cui il registro VREG non viene abilitato al caricamento e di conseguenza, al termine dell'intervallo t_1, esso non viene aggiornato. Nel successivo intervallo t_2, VCOMP vede $A = -7$ e $B = -18$; la sua uscita `AGTB` si porta adesso alta, abilitando così il caricamento di VREG che avverrà al successivo fronte attivo del clock, con conseguente aggiornamento di `VMAX` al valore -7; e così via.

Il confronto aritmetico tra due valori A e B a 16 bit in complemento a 2 può essere implementato in vari modi. I comparatori standard, come accennato in precedenza, operano su numeri assoluti e producono un uscita tre segnali: `AGTB`, `AEQB`, `ALTB` corrispondenti ai tre casi possibili $A > B$, $A = B$ e $A < B$. Nel caso di numeri in complemento a 2, ricordando che essi vanno considerati positivi o nulli se hanno il bit più significativo (bit di segno) pari a 0, mentre vanno considerati negativi se hanno bit di segno pari ad 1, è agevole osservare che:

- se i segni degli operandi A e B sono diversi, $\mathtt{A}_{15} \neq \mathtt{B}_{15}$, allora è $A > B$ se $\mathtt{A}_{15} = 0$ e $\mathtt{B}_{15} = 1$, mentre è $A < B$ se $\mathtt{A}_{15} = 1$ e $\mathtt{B}_{15} = 0$, indipendentemente dai valori dei restanti bit sia di A che di B;
- se i segni degli operandi A e B sono uguali, $\mathtt{A}_{15} = \mathtt{B}_{15}$, allora, detti A' il numero assoluto ottenuto dai bit 0-14 di A e B' il numero assoluto ottenuto dai bit 0-14 di B, è $A > B$, $A = B$ oppure $A < B$ a seconda che $A' > B'$, $A' = B'$, oppure, rispettivamente, $A' < B'$.

Possiamo di conseguenza costruire una logica combinatoria che utilizza $\mathtt{A}_{15}$, $\mathtt{B}_{15}$ e l'uscita `AGTB15` di un comparatore standard a 15 bit per generare l'uscita `AGTB` di confronto in complemento a 2; la tavola di verità di tale circuito è:

[2] Naturalmente, sotto l'ipotesi che il periodo di `XCLK` sia adeguatamente maggiore del tempo di assestamento dell'uscita del comparatore.

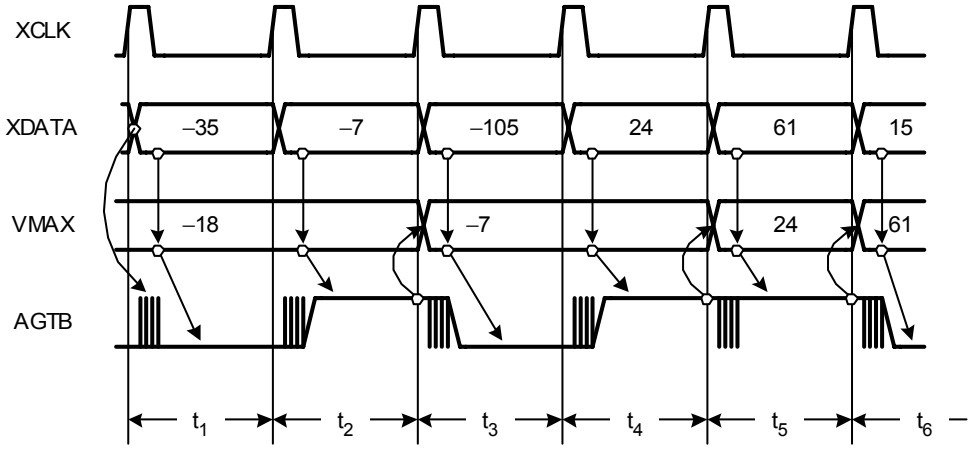


Fig. 10: Temporizzazioni del circuito di Fig. 9.

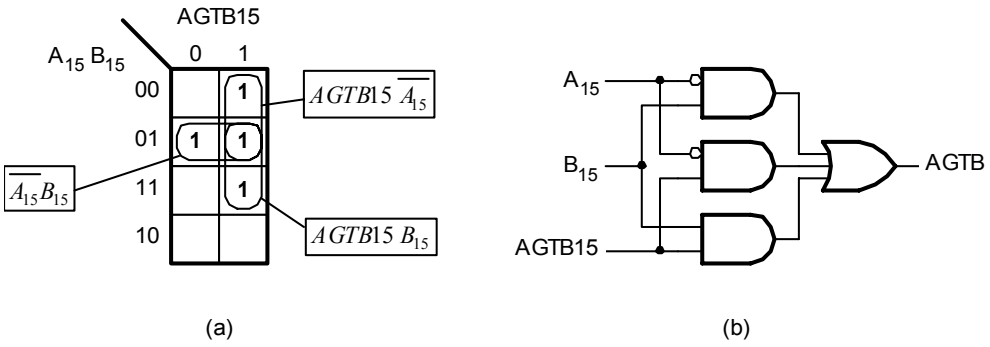


Fig. 11: Logica per la generazione di `AGTB`.

A_{15}	B_{15}	AGTB15	AGTB
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

La mappa di Karnaugh corrispondente a tale tavola è illustrata in Fig. 11*a*, mentre il circuito combinatorio corrispondente appare in Fig. 11*b*; il circuito completo del comparatore è poi riportato in Fig. 12. (Si noti come il comparatore standard a 15 bit sia stato ottenuto da un modulo a 16 bit forzando a 0 i bit più significativi dei suoi ingressi dati[3].) Naturalmente, il circuito combinatorio relativo alla tavola di cui sopra può in alternativa essere realizzato con un multiplexer; un esempio di tale implementazione è riportato in Fig. 13.

Un altro modo per realizzare VCOMP può essere quello di eseguire la sottrazione $Z = B - A$ e

[3]Naturalmente, avremmo potuto in alternativa forzare a 0 (o ad 1) i bit meno significativi, o addirittura una qualunque altra coppia di bit omologhi.

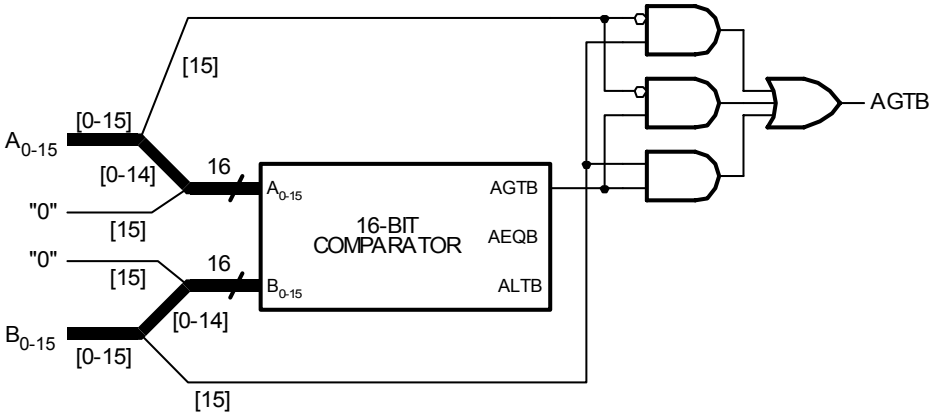


Fig. 12: Comparatore in complemento a 2 con uso di comparatore standard.

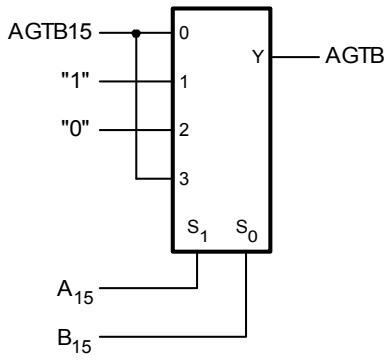


Fig. 13: Logica di comparazione realizzata con multiplexer.

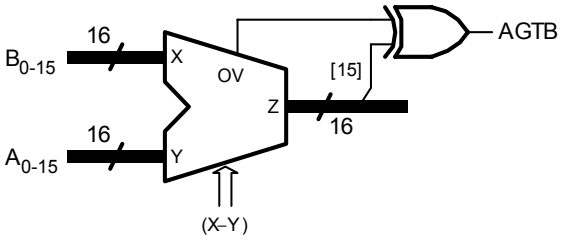


Fig. 14: Comparatore realizzato con ALU a 16 bit programmata in sottrazione.

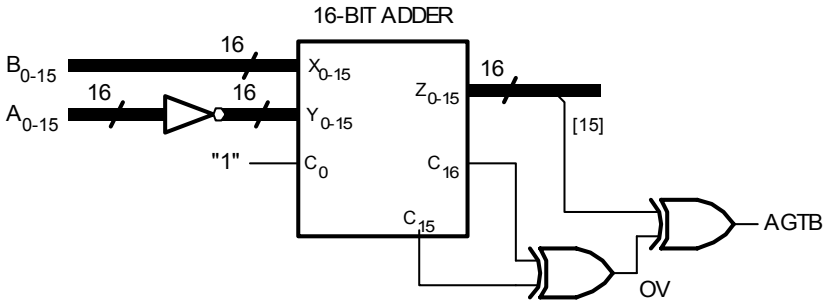


Fig. 15: Comparatore realizzato mediante addizionatore configurato in sottrazione.

testare il segno del risultato: se questo è non negativo ($\mathtt{Z}_{15} = 0$) allora $A \leq B$, se invece è negativo ($\mathtt{Z}_{15} = 1$) allora $A > B$. In tal caso, tuttavia, se si utilizzano circuiti aritmetici a 16 bit, occorrerà tener conto anche della possibilità di overflow in sottrazione. Poiché in condizioni di overflow il segno $\mathtt{Z}_{15}$ del risultato è per definizione errato, la logica per la generazione di `AGTB` in funzione di $\mathtt{Z}_{15}$ e dell'overflow `OV` è data dalla tavola che segue:

`OV`	$\mathtt{Z}_{15}$	segno apparente	segno reale	`AGTB`
0	0	*positivo*	*positivo*	0
0	1	*negativo*	*negativo*	1
1	0	*positivo*	*negativo*	1
1	1	*negativo*	*positivo*	0

Appare evidente come `AGTB` possa essere allora realizzato con un semplice XOR: $\mathtt{AGTB} = \mathtt{OV} \oplus \mathtt{Z}_{15}$.

La sottrazione $Z = B - A$ può a sua volta essere realizzata con una *Unità Logico-Aritmetica* (*Arithmetic Logic Unit*, ALU) a 16 bit, programmata per eseguire una sottrazione degli operandi X e Y (Fig. 14). In alternativa, ricordando che la sottrazione $B - A$ in complemento a 2 equivale all'operazione $B + \overline{A} + 1$, si può utilizzare un addizionatore a 16 bit e configurarlo in sottrazione negando l'operando Y e applicando un carry di ingresso $C_{in} = \mathtt{C}_0 = 1$; il segnale `OV` di overflow, come si ricorderà, viene ricavato mediante XOR dei due bit più significativi di carry: $\mathtt{OV} = \mathtt{C}_{15} \oplus \mathtt{C}_{16}$, dove $\mathtt{C}_{16} = C_{out}$ è il carry di uscita dall'addizionatore; il circuito relativo appare in Fig. 15.

Naturalmente, se invece di ALU o addizionatori a 16 bit si utilizzano circuiti a numero maggiore di bit (avendo naturalmente cura di estendere opportunamente il segno degli operandi), non si ha in nessun caso overflow; tuttavia, la semplificazione della logica che ne conseguirebbe non sembra giustificare la maggior dimensione dei circuiti aritmetici, soprattutto se si tien conto che i moduli aritmetici reali aumentano di dimensione a passi da 4 o da 8 bit.

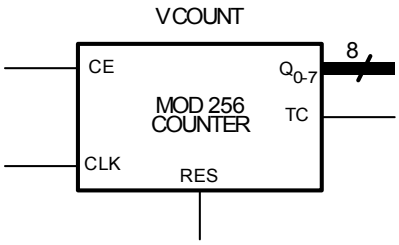


Fig. 16: Contatore dei dati sottoposti ad elaborazione.

3.3 Conteggio dei dati

Poiché IFMAX deve processare un blocco di 256 dati consecutivi, sarà necessario impiegare un contatore sincrono modulo 256, ossia ad 8 bit, che chiameremo VCOUNT. Poiché tale contatore deve stare al passo dei dati `XDATA`, esso sarà comandato dal clock `XCLK`; il segnale di *Terminal Count* (`TC`) prodotto dal contatore, che diventa attivo solo entro il ciclo di clock corrispondente al massimo conteggio[4] (255), verrà inviato al sottosistema di controllo per indicare la fine delle operazioni e, in quanto tale, farà parte delle informazioni di Status che viaggiano dal SCA al SCO.

D'altra parte, vi saranno delle circostanze in cui il contatore dovrà essere inibito al conteggio (ad esempio alla fine delle operazioni, quando avremo segnalato alla CPU l'avvenuto completamento del processo e staremo aspettando un nuovo segnale di `START`). Di conseguenza, VCOUNT deve essere dotato di un ingresso (sincrono) di abilitazione al conteggio `CE` (*Count Enable*); tale ingresso sarà controllato dal SCO, e di conseguenza farà parte dei segnali di Control che viaggiano da SCO a SCA.

Inoltre, il contatore dovrà essere inizializzato a zero ad ogni nuovo ciclo di operazioni (ed è in generale opportuno che ciò avvenga anche al Reset di sistema), per cui VCOUNT dovrà anche essere dotato di un ingresso (asincrono) di reset (`RES`). Anche tale ingresso sarà prodotto da SCO, e farà di conseguenza parte dei segnali di Control che viaggiano da SCO a SCA. In definitiva, il modulo VCOUNT che impiegheremo in IFMAX avrà la struttura illustrata in Fig. 16.

3.4 Inizializzazione

All'inizio di ogni ciclo di operazioni, il SCA non può partire da uno stato arbitrario ma deve essere opportunamente inizializzato. Lo stato del SCA, corrispondente alle variabili contenute nei suoi elementi di memoria, è costituito in definitiva da:

- valore di $V_{\max}$, contenuto nel registro VREG,
- valore di conteggio del contatore VCOUNT.

Abbiamo già accennato alla necessità di inizializzare VCOUNT col valore 0 ad ogni nuovo ciclo di operazioni, e abbiamo già visto come questa funzione viene delegata al SCO tramite la generazione da parte sua del segnale `RES` per VCOUNT. Per quanto riguarda l'inizializzazione di $V_{\max}$, abbiamo a disposizione due opzioni, che andrebbero entrambe implementate mediante comandi provenienti dal SCO:

Opzione 1 – Inizializzare $V_{\max}$ col minimo valore esprimibile su 16 bit in complemento a 2, ossia col valore $\texttt{8000}_{16}$ (corrispondente a -32768_{10}).

Opzione 2 – Forzare un caricamento incondizionato di VREG col primo dei 256 valori accettati dalle linee `XDATA`.

[4] Ricordiamo che un contatore standard modulo N produce i conteggi $0, 1, 2, \ldots, N-1$, dopo di che il ciclo ricomincia.

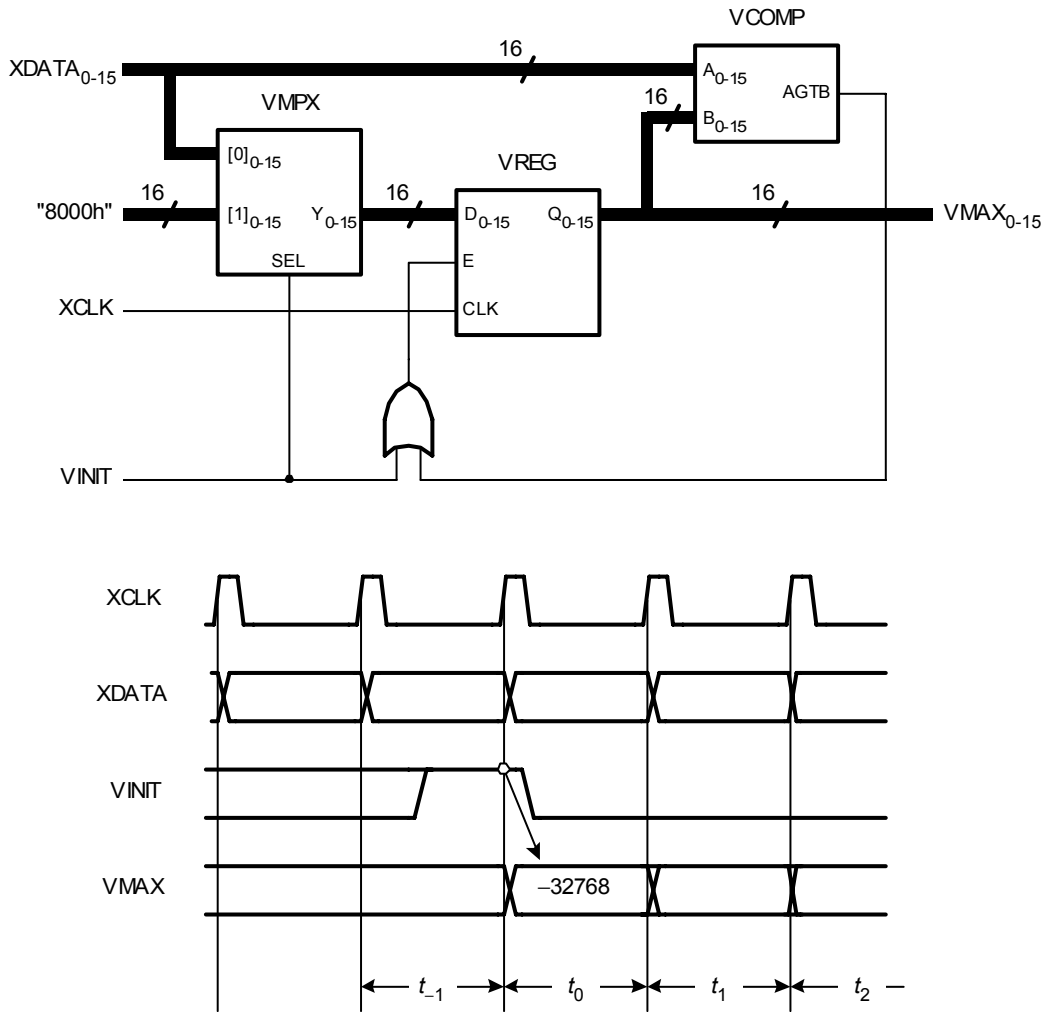


Fig. 17: Aggiunta dei circuiti di inizializzazione (opzione 1) e relative temporizzazioni.

In base all'opzione 1, dovremmo fare in modo che il SCO generi un segnale di inizializzazione `VINIT` e che l'ingresso dati di VREG possa accettare o `XDATA` (quando `VINIT` = 0) oppure la costante di inizializzazione (quando `VINIT` = 1); di conseguenza, potremmo utilizzare un multiplexer VMPX controllato da `VINIT` per produrre i dati da inviare all'ingresso di VREG. Inoltre, quando `VINIT` = 1, dovremmo anche forzare l'abilitazione al caricamento di VREG e, per di più, garantire che tale condizione rimanga stabile per almeno un intervallo del clock `XCLK` (altrimenti VREG non avrebbe modo di essere caricato). Il SCA modificato secondo l'opzione 1 diventerebbe allora quello illustrato in Fig. 17; si noti l'introduzione di una porta OR sull'abilitazione di VREG che produce un caricamento incondizionato quando `VINIT` = 1 mentre, quando `VINIT` = 0, lascia che il caricamento sia condizionato al valore dell'uscita `AGTB` del comparatore VCOMP. Osservando le temporizzazioni di Fig. 17, dove abbiamo indicato con t_0 il ciclo di clock in cui viene presentato il primo dei 256 valori `XDATA` da elaborare, è da osservare che la condizione `VINIT` = 1 deve necessariamente aver luogo al ciclo t_{-1}, ossia subito prima del primo valore della sequenza — essendo del tutto irrilevante il valore che essa assume nei cicli precedenti.

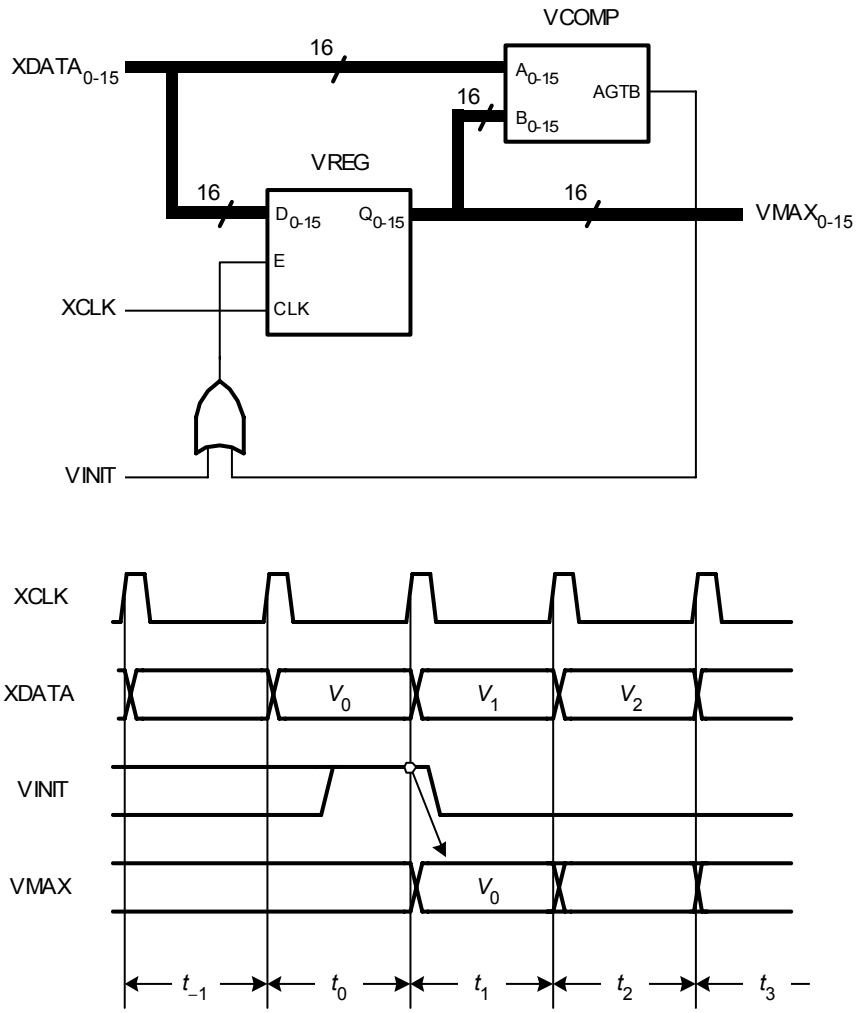


Fig. 18: Aggiunta dei circuiti di inizializzazione (opzione 2) e relative temporizzazioni.

In base all'opzione 2, invece, sarebbe sufficiente che il segnale `VINIT` forzasse un caricamento incondizionato di VREG e che venisse generato in corrispondenza al primo dei 256 valori della sequenza `XDATA` da elaborare, ossia in corrispondenza al ciclo di clock t_0 (essendo, anche in questo caso, irrilevante il valore assunto da `VINIT` nei cicli precedenti). In tal caso non avremmo necessità di multiplexer aggiuntivi; per di più, come vedremo in seguito, avremmo modo di generare `VINIT` in maniera molto semplice, poiché esso potrà essere vincolato a un ben determinato conteggio di VCOUNT o a un particolare stato della macchina sequenziale di controllo; la relativa modifica al SCA è illustrata in Fig. 18, assieme alle temporizzazioni corrispondenti. Dal momento che l'opzione 2 comporta la minore complessità circuitale come pure la maggior semplicità di controllo, la nostra scelta ricadrà senz'altro su di essa.

3.5 Fine delle operazioni

Come abbiamo già anticipato più sopra, il SCA può segnalare la fine delle operazioni mediante il Terminal Count (`TC`) di VCOUNT o mediante un segnale da esso derivato (ovviamente nell'ipotesi che il conteggio generato da VCOUNT rimanga correttamente al passo con i valori `XDATA` in arrivo). L'unica osservazione

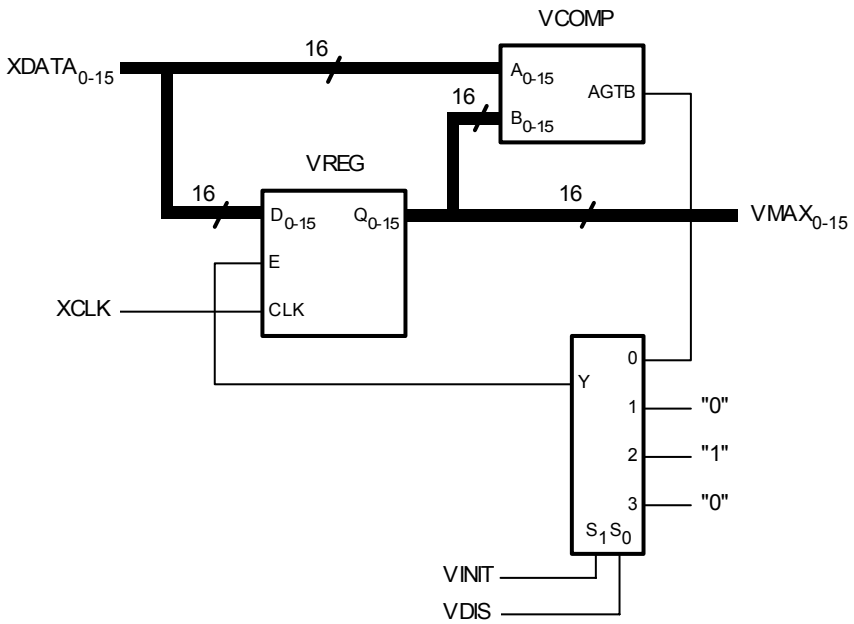


Fig. 19: Circuito di controllo dell'abilitazione al caricamento di VREG.

da fare è che, oltre alla necessità di generare il segnale `COMPLETE` per l'interfaccia di I/O (operazione che avverrà a cura del SCO), dovremo anche disabilitare ogni possibile caricamento del registro VREG fino al successivo riavviamento dei circuiti da parte del SCO; possiamo peraltro supporre che tale inibizione sia controllata da un segnale `VDIS` appositamente generato dal SCO. In definitiva, il segnale di abilitazione `E` al caricamento di VREG deve essere generato come segue:

caricamento di VREG	condizione
forzato	inizializzazione (`VINIT` = 1)
dipendente da `AGTB`	normale operazione (`VINIT` = `VDIS` = 0)
inibito	operazione completata (`VDIS` = 1)

e un possibile circuito che realizza tale funzione facendo uso di un multiplexer è illustrato in Fig. 19.

4 Sottosistema di controllo

Il SCO è generalmente costituito da una *macchina sequenziale a stati finiti* (*Finite State Machine*, FSM), sincronizzata al clock del SCA (nel nostro caso `XCLK`), che deve prevedere:

- uno **stato di riposo**, entro il quale essa deve ciclare fino all'arrivo di un comando di inizio operazioni proveniente dalla CPU;
- uno o più **stati di normale operazione**, entro i quali il SCA esegue nell'ordine dovuto le operazioni per cui è stato progettato;
- uno **stato di completamento**, che rappresenta l'avvenuta fine delle operazioni.

Tipicamente, lo stato di completamento si protrae per almeno un periodo di clock, dopo di che la FSM torna allo stato di riposo (Fig. 20); il Reset generale di sistema (`RESET`) solitamente forza la FSM

nello stato di riposo, o immediatamente ovvero facendola transitare per uno o più stati di inizializzazione speciale. A seconda della sua complessità, poi, la FSM di controllo può essere implementata o come un classico circuito sequenziale, con un array di flip-flop in cui vengono mantenute le variabili di stato necessarie, ovvero come struttura microprogrammata.

Nel caso delle periferiche PD32 in generale — e di IFMAX in particolare — tuttavia, le peculiarità e le caratteristiche dei segnali richiesti per l'interconnessione col blocco di interfaccia I/O suggeriscono una soluzione generale semplice e, nel contempo, efficiente al problema dell'avvio e del completamento delle operazioni.

4.1 Avvio e arresto del SCO

Il SCO deve essere avviato quando l'interfaccia di I/O emette un segnale `STARD`; tale segnale è di tipo impulsivo, con larghezza legata al periodo del clock di sistema `SYSCLK`, e di conseguenza *non ha alcuna relazione* col clock `XCLK`, proveniente dal mondo esterno, a cui è sincronizzato il SCO. In particolare, se `XCLK` ha periodo sufficientemente maggiore di `SYSCLK`, vi è la possibilità che l'impulso `STARD` sia completamente contenuto entro un periodo di clock di `XCLK` (Fig. 21); se il SCO campiona `STARD` ad ogni clock `XCLK`, allora esso non ha in tal caso *nessun* modo di accorgersi dell'avvenuta emissione di `STARD` stesso. D'altra parte, può anche accadere che `XCLK` abbia frequenza molto maggiore di `SYSCLK`, e di conseguenza la larghezza di `STARD` si estenda per parecchi cicli di `XCLK` (Fig. 22a): in tal caso, il SCO riconosce la presenza di `STARD` per *più volte* consecutive, e la FSM di controllo, se esce dallo stato di riposo non appena rivelato l'arrivo di `STARD`, deve di conseguenza entrare in uno stato speciale dal quale uscire solo quando `STARD` ritorna inattivo (Fig. 22b); se così non fosse, e se il SCA completasse rapidamente le proprie operazioni, vi sarebbe addirittura il rischio di emettere un segnale di `COMPLETE` *prima* che `STARD` sia tornato a 0, con conseguente indeterminazione dell'uscita del flip-flop SR che nell'interfaccia di I/O genera il segnale `READY`[5].

Problemi analoghi si presentano nella generazione del segnale `COMPLETE`, anch'esso di tipo impulsivo. La tecnica più immediata per produrlo consiste nel fare in modo che la FSM transiti per un dato stato, in corrispondenza al quale viene generato `COMPLETE`, vi permanga per un ciclo di `XCLK`, e poi ne esca, riportando così `COMPLETE` inattivo. Tuttavia, se il periodo `XCLK` è eccessivamente largo rispetto al periodo di `SYSCLK`, e se di conseguenza `COMPLETE` rimane attivo per un tempo eccessivamente lungo, vi è il rischio che il PD32 emetta un nuovo segnale di `START` quando ancora `COMPLETE` non è esaurito (Fig. 23), con conseguente indeterminazione dell'uscita del flip-flop di `READY`.

Un modo per risolvere tutti questi problemi consiste nell'utilizzare nell'ambito della FSM un circuito generatore di *Reset Interno* (*Internal Reset*, `IRES`) con le seguenti funzioni:

- `IRES` forza (in modo *asincrono*) la FSM nello stato di riposo;
- `IRES` diventa inattivo in corrispondenza al fronte di *caduta* di `STARD`, in modo che il primo fronte di `XCLK` successivo porti la FSM fuori dallo stato di riposo;
- lo stato generatore di `COMPLETE` forza (in modo *asincrono*) `IRES` a ritornare attivo e a portare la FSM nuovamente nello stato di riposo — in tal modo, lo stato di `COMPLETE` diventa *instabile*, mentre, come vedremo tra breve, il segnale `COMPLETE` avrà caratteristiche impulsive con larghezza tuttavia sufficiente ad agire correttamente e in maniera sicura sul flip-flop di `READY`.

La corrispondente temporizzazione è illustrata in Fig. 24. Quando `IRES` è attivo, la FSM è forzata nello stato di riposo; quando, all'istante t_1, arriva il fronte di caduta di `STARD`, `IRES` diventa inattivo, e, al prossimo fronte di `XCLK` (istante t_2), la FSM esce dallo stato di riposo iniziando le normali operazioni. All'istante t_3 la FSM entra nello stato di completamento (tratteggiato in Fig. 24), la cui decodifica

[5] Ricordiamo che `STARD` viene generato a partire dal segnale di `START`, che viene applicato all'ingresso `R` del flip-flop che produce `READY`, mentre `COMPLETE` viene applicato all'ingresso `S` del medesimo flip-flop: se un flip-flop SR vede contemporaneamente attivi entrambi i suoi ingressi `S`, `R`, la sua uscita diventa indeterminata.

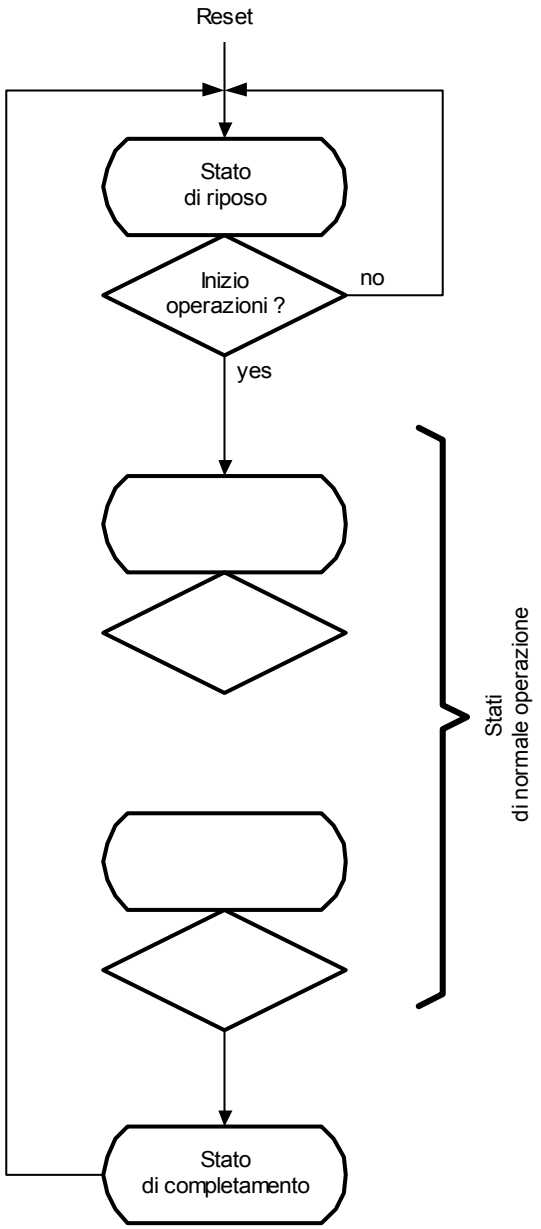


Fig. 20: Struttura generale di un SCO.

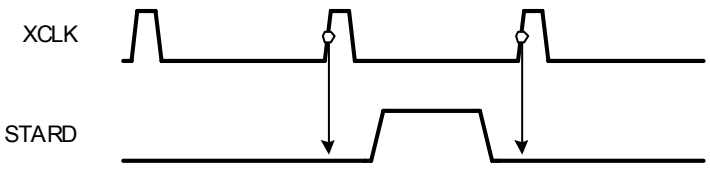
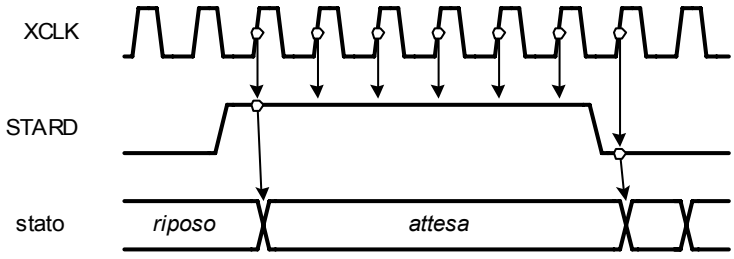
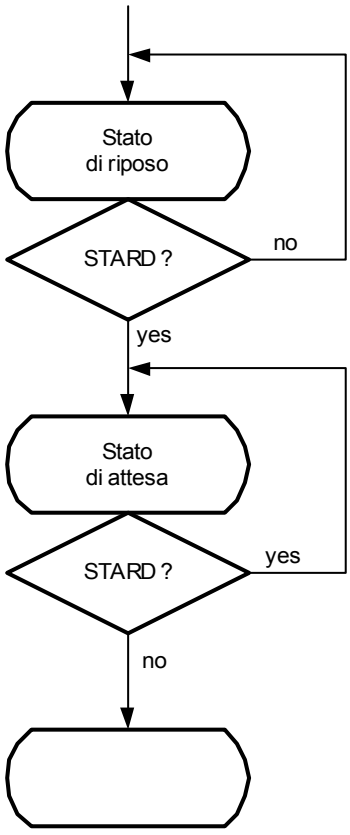


Fig. 21: Esempio di segnale `STARD` con larghezza insufficiente per essere rivelato dal SCO.



(a)



(b)

Fig. 22: Esempio di segnale `STARD` con larghezza eccessiva (a), e relativa modifica al diagramma di stato (b).

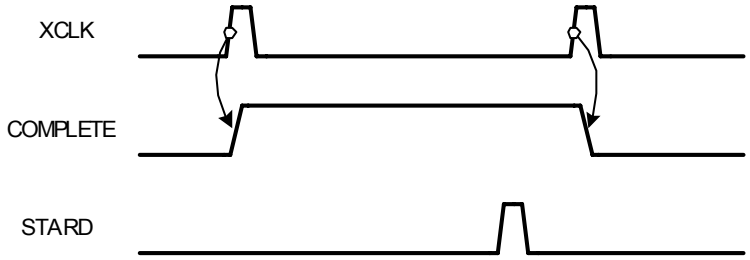


Fig. 23: Arrivo di STARD con COMPLETE ancora attivo.

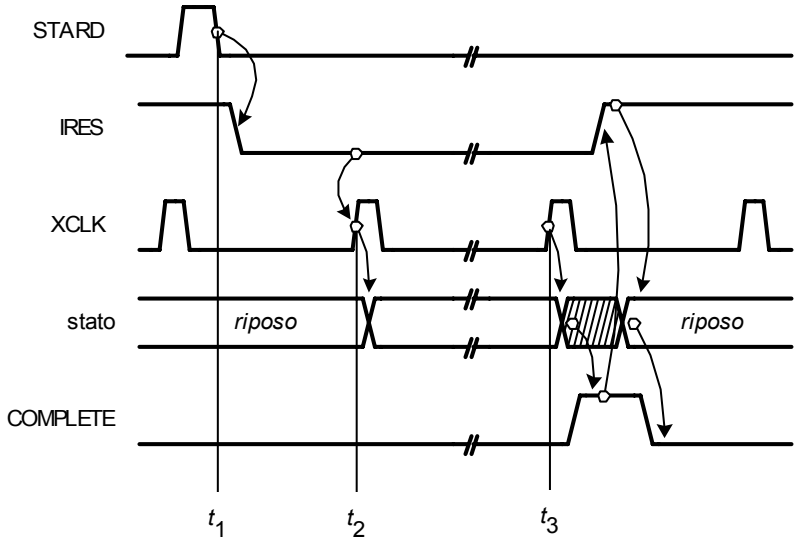


Fig. 24: Comportamento del circuito generatore di Internal Reset.

genera COMPLETE; questo segnale, d'altra parte, provoca l'attività di IRES che riporta immediatamente la FSM nello stato di riposo, che, a sua volta, rende COMPLETE inattivo. Il ciclo ricomincerà solo con l'arrivo di un nuovo STARD.

Un circuito che realizza le funzioni descritte appare in Fig. 25, dove è anche illustrato come IRES agisca sulla FSM di controllo semplicemente resettando i flip-flop di stato (abbiamo qui fatto l'ipotesi che lo stato di riposo sia codificato con 00...0, e che il segnale COMPLETE venga prodotto decodificando una specifica configurazione delle variabili di stato con un AND connesso alle uscite dirette o negate dei relativi flip-flop); la struttura consente anche una facile connessione al Reset di sistema (RESET), il quale, come si può osservare, forza semplicemente IRES attivo e di conseguenza porta la FSM nello stato di riposo. La larghezza del segnale COMPLETE può essere determinata prendendo in considerazione i tempi necessari al verificarsi della seguente sequenza di eventi:

- COMPLETE diventa attivo quando la FSM entra nello stato di completamento;
- COMPLETE deve poi attraversare la porta OR di Fig. 25 e resettare il flip-flop D che genera IRES;
- IRES attivo deve resettare le variabili di stato della FSM in modo da riportarla allo stato di riposo;
- solo quando la FSM è entrata nello stato di riposo, la decodifica di stato riporta COMPLETE inattivo.

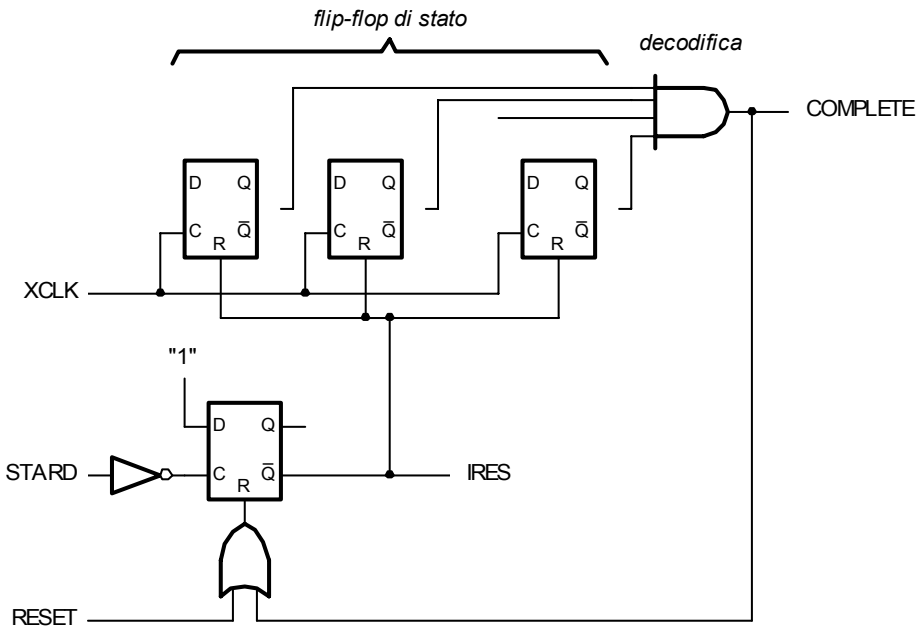


Fig. 25: Circuito generatore di Internal Reset, e sua azione sulla FSM di controllo.

`COMPLETE` permane dunque attivo per almeno il doppio del tempo necessario a resettare un flip-flop, e di conseguenza la sua larghezza sarà senz'altro sufficiente a settare in maniera sicura ed affidabile il flip-flop di READY.

4.2 Controllo di IFMAX

Una volta risolti i problemi di avvio e di completamento delle operazioni, il controllo di IFMAX è estremamente semplice: a meno degli stati speciali (R, riposo; C, completamento) esaminati sopra, avremo bisogno soltanto di:

- uno stato I per l'**inizializzazione** del SCA, nel quale la FSM deve permanere per un solo clock, e di
- uno stato N di **normale operazione**, dal quale la FSM deve uscire al completamento delle operazioni, ossia quando il contatore VCOUNT produce un Terminal Count ($\texttt{TC} = 1$).

Il diagramma di flusso che ne risulta è illustrato in Fig. 26, dove è anche illustrata la codifica degli stati, che, salvo per $R = \texttt{00}$, è stata scelta in maniera del tutto arbitraria[6]; il tratteggio sullo stato C di completamento ci rammenta che tale stato è instabile.

Se la FSM di controllo viene realizzata con flip-flop D, la tavola di eccitazione risulta la seguente[7]:

[6] Abbiamo qui volutamente tralasciato di occuparci della codifica ottimale degli stati; sarebbe facile mostrare, difatti, come una diversa codifica potrebbe comportare ulteriori semplificazioni in vari circuiti.

[7] Ulteriori semplificazioni dei circuiti di eccitazione si sarebbero potute ottenere ricorrendo a flip-flop di tipo JK anziché di tipo D.

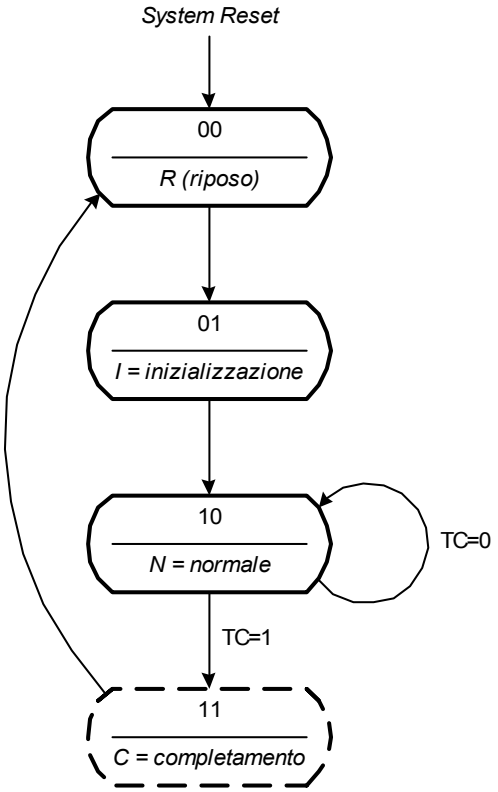


Fig. 26: Diagramma di stato della FSM di controllo.

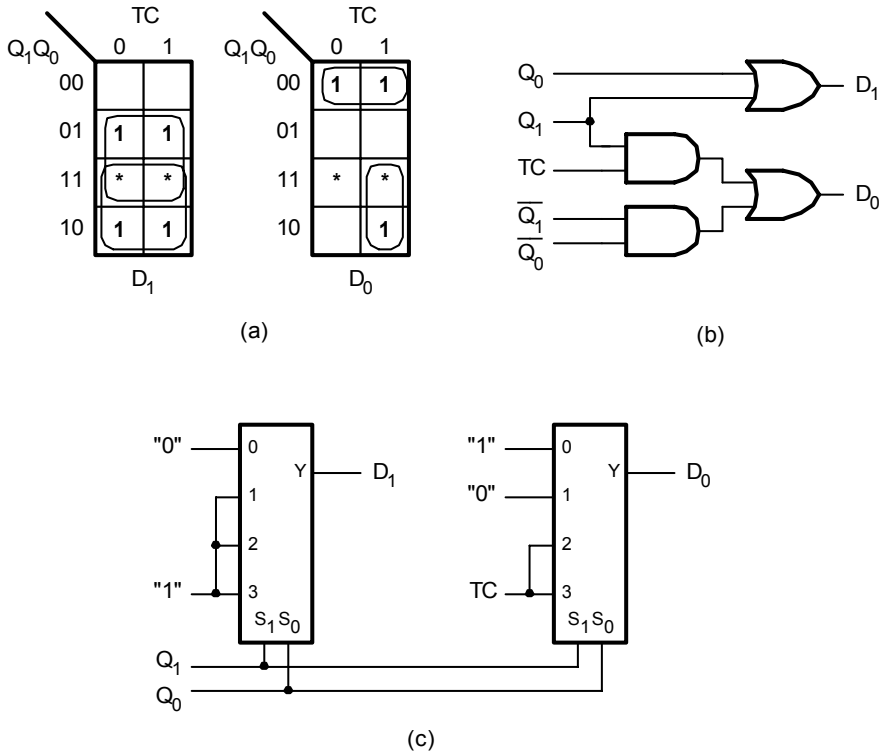


Fig. 27: Circuito di eccitazione della FSM di controllo.

Stato presente Q_1Q_0		TC	Stato futuro	Ingressi D_1D_0
R	00	0	I	01
	00	1	I	01
I	01	0	N	10
	01	1	N	10
N	10	0	N	10
	10	1	C	11
C	11	0	?	**
	11	1	?	**

(Si noti come, essendo lo stato C instabile ed essendo forzata da `IRES` la transizione da esso allo stato R di riposo, lo stato futuro di C può essere lasciato non specificato, il che contribuisce tra l'altro a semplificare i circuiti di eccitazione della FSM.)

Le relative mappe di Karnaugh appaiono in Fig. 27*a*, e il circuito di eccitazione corrispondente è illustrato in Fig. 27*b*; naturalmente, quest'ultimo può anche realizzato mediante multiplexer, come è riportato a titolo di esempio in Fig. 27*c*.

Le uscite di controllo dalla FSM al SCA, come discusso più sopra, sono:

- ingresso `RES` di reset per VCOUNT,
- abilitazione al conteggio `CE` per VCOUNT,
- abilitazione e inibizione al caricamento di VREG (segnali `VINIT` e `VDIS`).

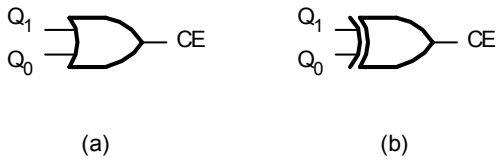


Fig. 28: Realizzazioni alternative del segnale di controllo `CE`: (a) mediante OR, (b) mediante XOR.

Il contatore VCOUNT deve restare resettato quando la FSM si trova nello stato di riposo, e dunque basta connettere il reset interno `IRES` all'ingresso `RES` di reset del contatore stesso, in maniera che questo rimanga libero di contare non appena la FSM esce dallo stato di riposo. Tuttavia, per ottenere il conteggio corretto, in modo che all'arrivo di `TC` da VCOUNT il SCA stia processando esattamente il 256-esimo valore della sequenza, dobbiamo fare in modo che durante lo stato I di inizializzazione il contatore parta dal conteggio 0. A tal fine, è sufficiente abilitare VCOUNT al conteggio, ossia porre `CE` $= 1$, solo negli stati I ed N, e mantenerlo disabilitato in tutti gli altri stati. La tavola di verità di `CE` in funzione dello stato presente è allora:

stato presente	Q_1Q_0	`CE`
R	00	0
I	01	1
N	10	1
C	11	*

Osserviamo come anche in questo caso, essendo lo stato C instabile, possiamo assegnare a `CE` una condizione non specificata. Senza necessità di ricorrere alla mappa di Karnaugh, è immediato vedere che il segnale `CE` può essere sintetizzato come $Q_1 + Q_0$ o anche come $Q_1 \oplus Q_0$ (Fig. 28).

Quanto al caricamento di VREG, come si ricorderà dalla discussione fatta sopra, dobbiamo generare due segnali `VINIT` e `VDIS` in modo che VREG:

- venga forzato al caricamento (`VINIT` $= 1$, `VDIS` $= 0$) quando la FSM si trova nello stato I di inizializzazione, in modo che il primo valore della sequenza possa essere scritto incondizionatamente nel registro,
- venga caricato o meno, a seconda del valore assunto dall'uscita `AGTB` del comparatore (`VINIT` $= 0$, `VDIS` $= 0$), quando la FSM si trova nello stato N di normale operazione,
- venga inibito al caricamento (`VINIT` $= 0$, `VDIS` $= 1$) in tutti gli altri stati.

La tavola di verità dei segnali `VINIT` e `VDIS` è allora:

stato presente	Q_1Q_0	`VINIT`	`VDIS`
R	00	0	1
I	01	1	0
N	10	0	0
C	11	*	*

Senza necessità di ricorrere alle mappe di Karnaugh, si vede facilmente che possiamo porre `VINIT` $= Q_0$ e $VDIS = \overline{Q}_1\overline{Q}_0 = \overline{Q_1 + Q_0}$. I circuiti corrispondenti appaiono in Fig. 29.

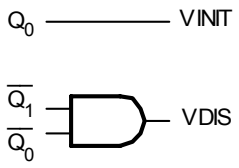


Fig. 29: Generazione dei segnali di controllo per il caricamento di VREG.

5 Schema generale

Il progetto di IFMAX è a questo punto completo: il suo schema logico generale è riportato in Fig. 30, mentre in Fig. 31 sono illustrate le temporizzazioni relative a un generico ciclo di operazioni. A riposo, si ha `IRES` = 1 che forza la FSM di controllo nello stato R (`00`) e VCOUNT al conteggio 0; inoltre, lo stato R comporta `VINIT` = 0, `VDIS` = 1, il che disabilita il caricamento del registro VREG, e di conseguenza quest'ultimo mantiene ancora memoria del valore calcolato nel precedente ciclo di operazioni. In queste condizioni, i clock `XCLK` non hanno alcun effetto (uno di questi appare in figura all'istante t_{-1}), poiché la FSM è forzata nello stato R = `00`, il contatore è forzato al conteggio 0, e VREG è inibito al caricamento.

Il nuovo processo inizia con l'arrivo di un impulso `STARD`, il cui fronte di caduta rende `IRES` inattivo e di conseguenza la FSM è libera di evolvere allo stato di inizializzazione I (`01`) all'arrivo del successivo fronte di `XCLK` (istante t_0). Lo stato I implica `VINIT` = 1, `VDIS` = 0, il che forza incondizionatamente l'abilitazione di VREG al caricamento, e di conseguenza il valore che transita sulle linee `XDATA` quando la FSM si trova nello stato I (valore che chiameremo convenzionalmente V_0) verrà copiato in VREG al prossimo fronte attivo di `XCLK` (istante t_1). Il contatore è adesso libero di contare, ma nell'intervallo (t_0, t_1) permane nel conteggio 0 poiché la sua abilitazione `CE` è stata attivata *dopo* il clock t_0.

Al clock t_1 la FSM passa allo stato di normale operazione N (`10`): in questo stato è `VINIT` = `VDIS` = 0, e l'abilitazione al caricamento di VREG dipende dunque dall'esito del confronto tra il valore V_1 presente su `XDATA` e il valore contenuto in VREG; se il confronto dà esito favorevole, `AGTB` si attiva e predispone VREG al caricamento di V_1 in corrispondenza al clock successivo che si presenterà all'istante t_2, diversamente il contenuto di VREG rimane invariato. Il contatore VCOUNT, che è abilitato, passa al conteggio 1.

L'interfaccia continua ad operare in maniera analoga fino all'istante t_{255}, in corrispondenza al quale il contatore raggiunge il conteggio terminale 255 ed emette `TC` = 1. Al successivo istante t_{256} questa condizione provoca la transizione della FSM allo stato di completamento C (`11`), che, come sottolineato in precedenza, è *instabile* e provoca non solo la generazione dell'impulso di `COMPLETE`, ma anche il reset della FSM allo stato di riposo R (`00`). Si noti come, perdurando lo stato C per meno di un periodo di `XCLK`, la presenza temporanea di `VINIT` = 1 non provoca alcuna alterazione del contenuto di VREG: essendo legata ad una abilitazione `E` sincrona, questa può avvenire solo all'arrivo di un successivo `XCLK`, ma per quell'istante la FSM sarà già tornata allo stato R (`00`) con conseguente forzamento di `VINIT` = 0, `VDIS` = 1, condizioni che, come abbimo visto, implicano `E` = 0.

In Fig. 32, infine, appare lo schema logico dell'interfaccia di I/O, dove le etichette in grassetto corsivo corrispondono ai segnali di bus del PD32.

6 Software di pilotaggio

Il software necessario a controllare l'interfaccia IFMAX è piuttosto semplice. La variabile `VMINMAX` risiede in memoria e va inizializzata col massimo valore positivo esprimibile su 16 bit in complemento a 2:

```
VMINMAX   DL  32767   ; inizializzazione di VMINMAX
```

L'indirizzo della periferica è una costante `IFMAX` che può essere assegnata in maniera arbitraria:

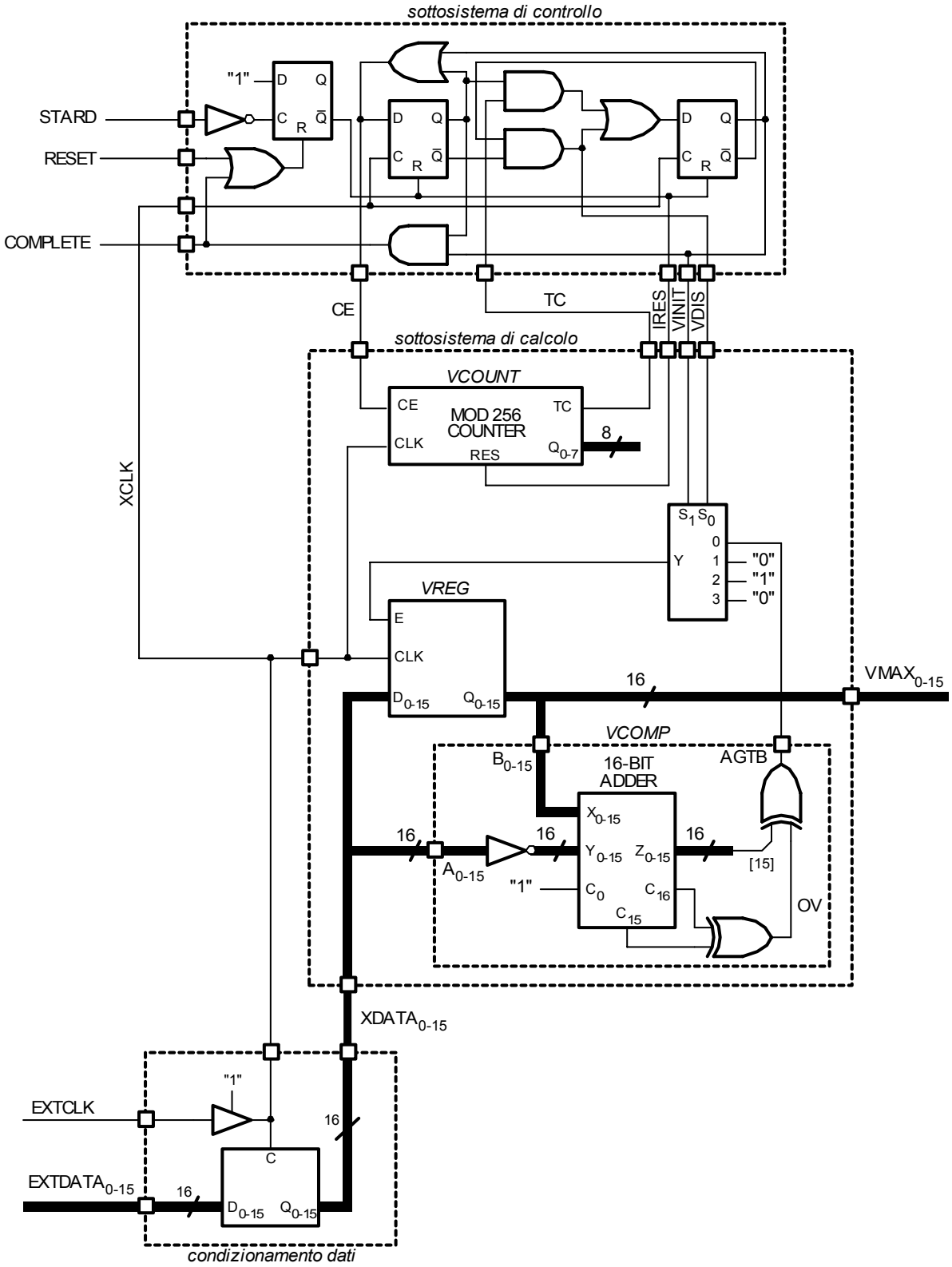


Fig. 30: Schema logico del blocco di condizionamento, del sottosistema di calcolo e del sottosistema di controllo.

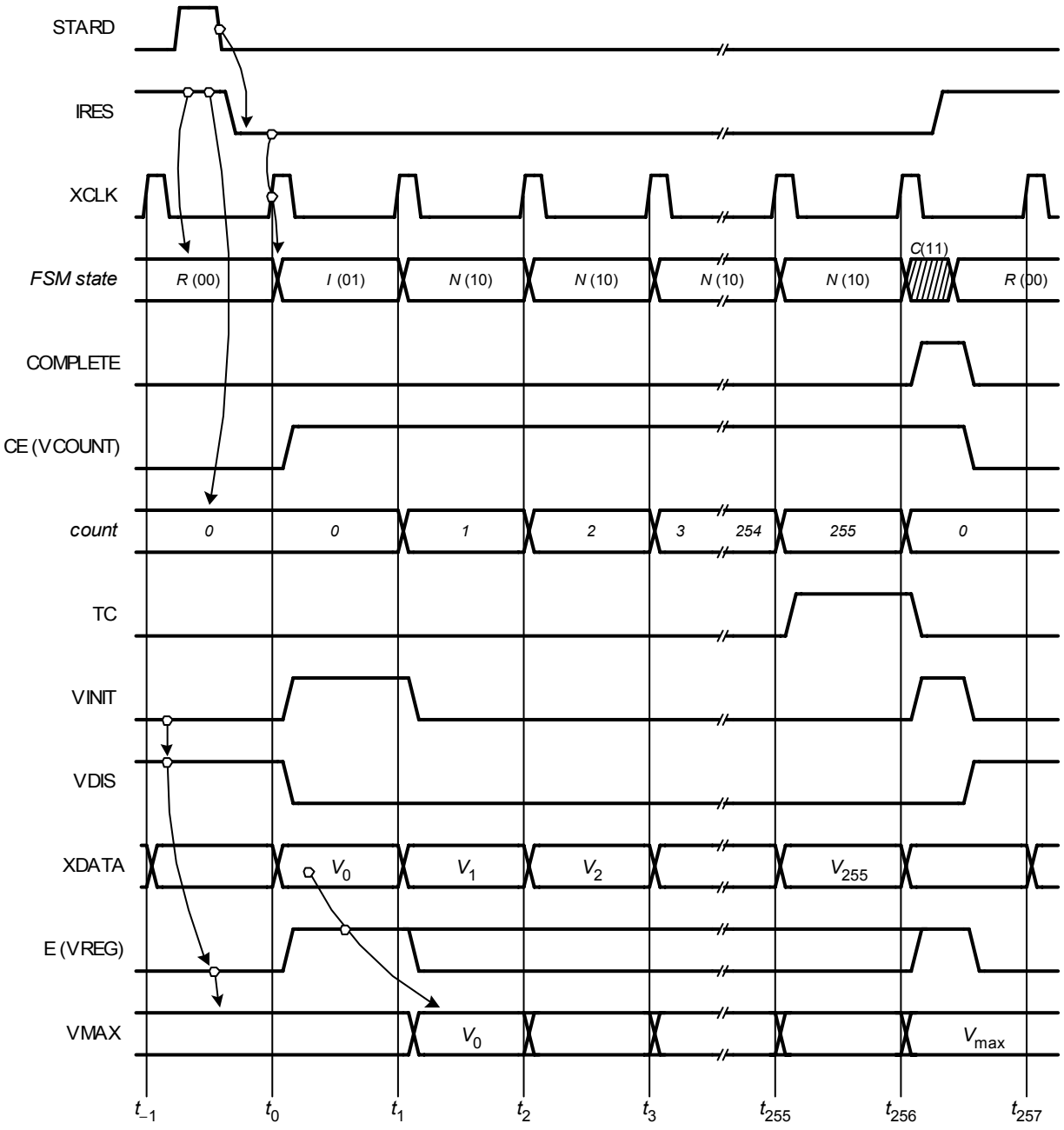


Fig. 31: Temporizzazioni di IFMAX.

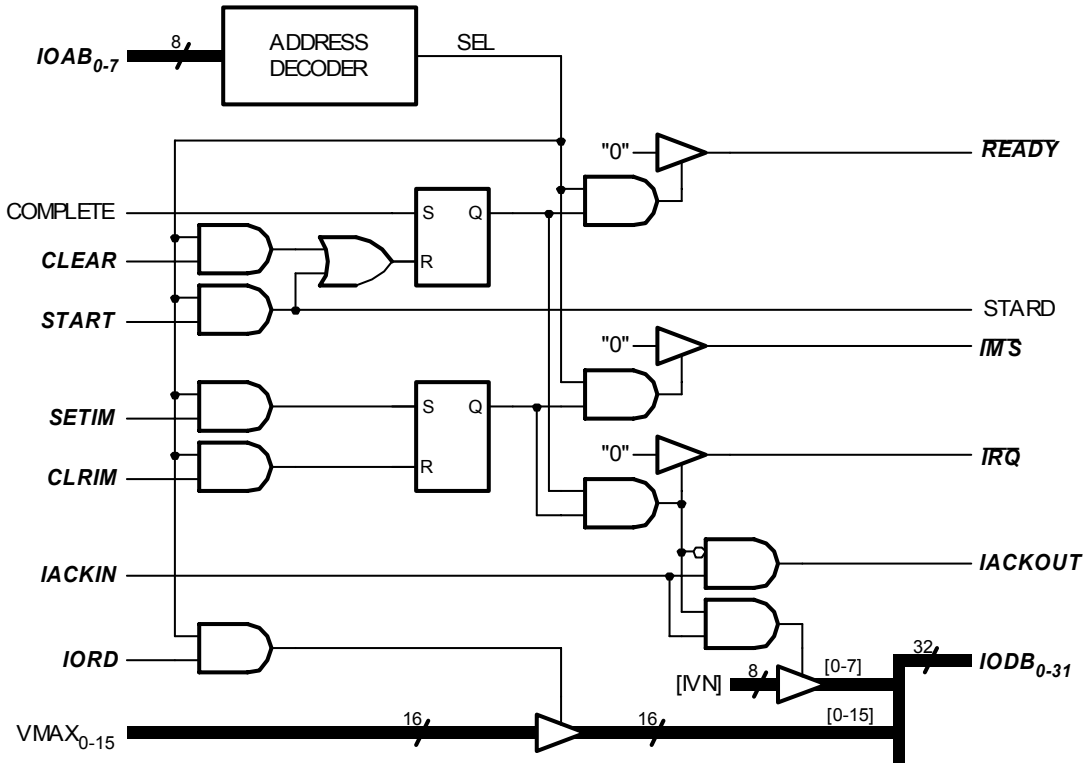


Fig. 32: Interfaccia di I/O (le etichette in grassetto corsivo corrispondono ai segnali di bus del PD32).

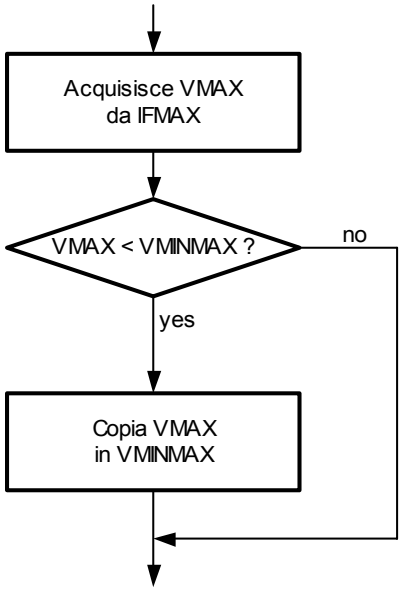


Fig. 33: Flowchart delle operazioni di acquisizione.

```
  IFMAX EQU   21        ; indirizzo (arbitrario) del device IFMAX
```

Le operazioni che la CPU deve eseguire in corrispondenza all'acquisizione di ciascun valore dall'interfaccia sono le seguenti (Fig. 33):

- leggere il valore $V_{\max}$ dall'interfaccia mediante un'operazione di input;
- confrontare $V_{\max}$ col valore corrente della variabile `VMINMAX`;
- se $V_{\max} <$ `VMINMAX`, aggiornare `VMINMAX` col valore di $V_{\max}$ appena ottenuto, altrimenti lasciare `VMINMAX` immutato.

Si noti che, così come accadeva nell'hardware, anche qui il confronto (che si riduce a un test sul segno della differenza tra i due valori) deve tener conto della possibilità di overflow (Fig. 34). Il sottoprogramma `SUB_IFMAX` che realizza le operazioni suddette è pertanto il seguente:

```
SUB_IFMAX:
  INW   IFMAX, R0       ; legge VMAX dall'interfaccia in R0
  CMPW  VMINMAX, R0     ; confronta VMAX con VMINMAX, eseguendo VMAX-VMINMAX
  JO    OVER            ; salta se overflow
  JNN   DONE            ; non overflow: salta se VMAX >= VMINMAX
  MOVW  R0, VMINMAX     ; altrimenti aggiorna VMINMAX
  J     DONE            ; salta a fine operazioni
OVER:
  JN    DONE            ; overflow: salta se VMAX > VMINMAX
  MOVW  R0, VMINMAX     ; altrimenti aggiorna VMINMAX
DONE:
  RET
```

Se il protocollo di I/O tra la CPU e l'interfaccia è di tipo busy-waiting, il frammento di programma che controlla le attività è il seguente:

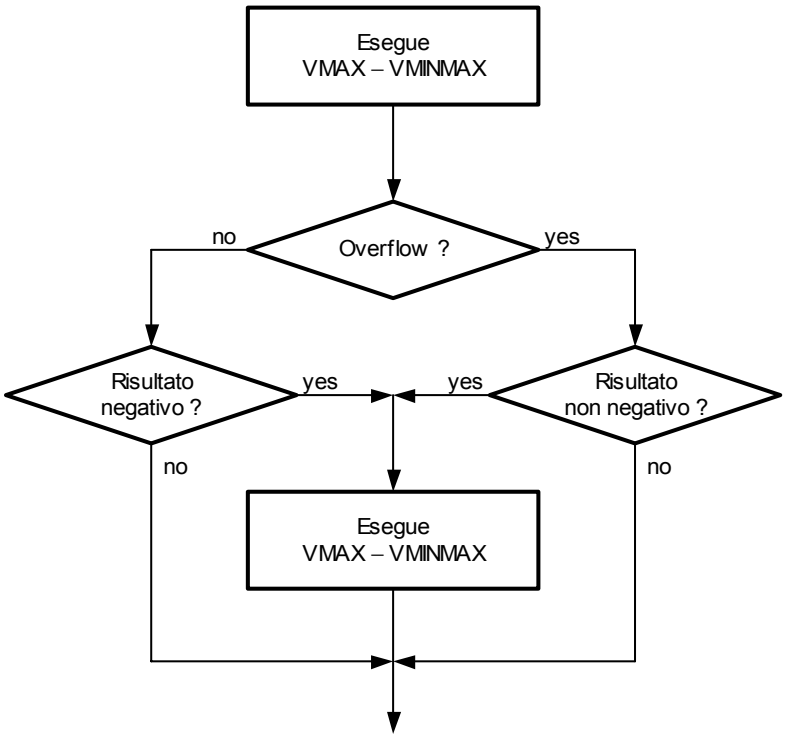


Fig. 34: Confronto con analisi dell'overflow.

```
BEGIN:
  START IFMAX           ; lancia le attivita' dell'interfaccia
  . . . . .             ; (esegue eventuali altre attivita')
NOTRDY:
  JNR   IFMAX, NOTRDY   ; aspetta che l'interfaccia sia pronta
  JSR   SUB_IFMAX       ; processa il risultato
  J     BEGIN           ; rientra in loop
```

Se invece il protocollo è basato su interrupt, la routine di servizio sarà così strutturata:

```
ISR_IFMAX:
  PUSH  R0              ; salva R0 nello stack
  JSR   SUB_IFMAX       ; processa il risultato
  POP   R0              ; ripristina R0
  START IFMAX           ; riavvia le attivita' dell'interfaccia
  RTI                   ; ritorna da interrupt
```