Parte Prima: guida allo studio e integrazioni

La Prima Parte del corso, che riguarda principalmente la strutturazione di sistemi di elaborazione a livello firmware:

- inizia con una breve **introduzione sulla strutturazione in generale a livelli ed a moduli**:
  - Cap. I della Dispensa: sez. 1 e 2.1 + brevi integrazioni e spiegazioni in queste note (sez. 1);
- contiene elementi di **rappresentazione binaria delle informazioni e sua utilizzazione**:
  - in queste note (sez. 2, 3, 4);
- include la trattazione del **livello hardware (reti logiche combinatorie e sequenziali**):
  - Cap. II della Dispensa, sez. 1, 2.1, 2.2, 2.3 + integrazioni in queste note (sez. 4, 6);
- si concentra sulla **definizione, progettazione e valutazione di unità di elaborazione**:
  - Cap. III della Dispensa + integrazioni in queste note (sez. 5, 6, 7, 8);
- è accompagnata dalla **prima Esercitazione di Verifica Intermedia**.

1. Sulla strutturazione di sistemi a livelli e a moduli 3
1.1. Strutturazione a livelli: compilazione e interpretazione 3
1.2. Strutturazione a moduli: unità di elaborazione e processi 5
2. Rappresentazione binaria delle informazioni 5
2.1. Conversione decimale-binario e binario-decimale 6
2.2. Numeri binari con segno 7
2.3. Nozioni di aritmetica binaria 7
3. Sull'utilizzazione della rappresentazione binaria delle informazioni 8
3.1. Proprietà fondamentali 8
3.2. Alcune proprietà utili 8
4. Strutture di calcolo, tipi di dato e rappresentazione binaria delle informazioni 10
4.1. Parole di informazione 10
4.2. Registri 11
4.3. Funzioni e reti combinatorie 12
4.4. Memorie 14
5. Guida allo studio della strutturazione firmware 15
6. Elementi di algebra booleana della commutazione 15
7. Reti sequenziali e strutturazione firmware: una rilettura per agevolarne la comprensione 18
7.1. Parte Controllo e Parte Operativa come reti sequenziali 18
7.1.1. Stato interno e funzione di transizione dello stato interno 18
7.1.2. Funzione delle uscite 19
7.2. Un esempio 21
7.2.1. Rete sequenziale Parte Operativa 22
7.2.2. Rete sequenziale Parte Controllo 25
7.2.3. Ciclo di clock 27
7.2.4. Ancora sulla funzione delle uscite della Parte Operativa 27
8. Comunicazioni a livello firmware 29
8.1. Comunicazione tra unità 29
8.2. Interfaccia a transizione di livello 30
8.3. Protocollo mediante segnali di RDY e ACK 30
8.4. Implementazione delle interfacce 31
8.5. Utilizzo del protocollo di comunicazione 33
8.6. Esempio 1 34
8.7. Esempio 2 34
8.8. Comunicazione sincrona 35
8.9. Comunicazione asincrona con grado di asincronia k > 1 35
8.10. Comunicazione a domanda e risposta 35
9. Esercizi sulla strutturazione firmware 36

1. Sulla strutturazione di sistemi a livelli e a moduli

I concetti e le tecniche sulla strutturazione di sistemi a livelli ed a moduli sono di primaria importanza per questo corso, e saranno oggetto di approfondimento durante tutte le parti successive.

La prima trattazione (Cap. I, sez. 1, 2.1) rappresenta una introduzione all'argomento, avente lo scopo di avere un inquadramento iniziale di massima.

1.1. Strutturazione a livelli: compilazione e interpretazione

Per quanto riguarda la strutturazione a livelli, nella Dispensa è messo in evidenza come il concetto caratterizzante sia quello di *supporto a tempo di esecuzione* di un linguaggio. Questo concetto è legato alla comprensione di concetti fondamentali di *compilazione* e di *interpretazione* di linguaggi: anche se questi aspetti sono studiati in diversi altri corsi, si ritiene conveniente riportare, in questa sede, alcune considerazioni che aiutino ad acquisire la necessaria sensibilità.

Il fine ultimo di un calcolatore è quello di rendere possibile l'esecuzione di programmi. Cominciamo con il caratterizzare questa generica affermazione:

1. i programmi vengono progettati mediante linguaggi ad alto livello. Per ragioni di efficienza e di generalità, occorre prevedere una *traduzione* da linguaggio ad alto livello in una forma intermedia, detta linguaggio macchina o linguaggio *assembler*. Le caratteristiche di questo linguaggio emergeranno dall'analisi successiva;
2. i calcolatori a cui ci riferiamo devono permettere l'esecuzione di *qualunque* programma: devono cioè essere *general-purpose*. Come sempre accade, coniugare generalità ed efficienza è un problema complesso: uno dei principali obiettivi dell'Architettura degli Elaboratori è dare soluzioni ragionevoli a questo problema;
3. la traduzione dal programma *sorgente* (in gergo, "codice sorgente") scritto in linguaggio ad alto livello, nel programma oggetto o *eseguibile* (in gergo, "codice oggetto" o "codice eseguibile"), espresso in linguaggio assembler, può essere effettuata secondo una delle due ben note tecniche, la *compilazione* e l'*interpretazione*, o loro combinazioni. In entrambi i casi, il punto di partenza è una sequenza di comandi, costituenti il programma scritto nel linguaggio ad alto livello, operanti su determinati insiemi di dati. Un compilatore o un interprete sono essi stessi dei programmi, già disponibili in forma eseguibile, che accettano come dato d'ingresso il programma sorgente (una rappresentazione opportuna della sequenza di comandi e rispettivi dati) e producono come dato di uscita il programma eseguibile;
4. un traduttore di tipo interprete scandisce tale sequenza sostituendo *ogni singolo* comando con una sequenza di istruzioni assembler nota, detta *l'interprete* di quel comando. Questa forma di traduzione è effettuata *dinamicamente*, cioè *a tempo di esecuzione*, da cui la dizione, equivalente a quella di interprete di un linguaggio, di *supporto a tempo di esecuzione* di quel linguaggio;
5. un traduttore di tipo compilatore sostituisce *l'intera* sequenza del programma sorgente con una sequenza di istruzioni assembler. Questa forma di traduzione è effettuata *staticamente*, cioè in fasi di preparazione prima che il programma passi in esecuzione. Anche il compilatore deve disporre di regole per sostituire ogni comando del programma sorgente con il rispettivo supporto a tempo di esecuzione, ma, a differenza dell'interpretazione:
   - la compilazione prende in esame tutto il programma sorgente,
   - per ogni comando del linguaggio sorgente, esistono più sequenze eseguibili per tenere conto efficientemente del contesto in cui il comando si trova ad essere inserito.

Per esemplificare la differenza tra compilazione e interpretazione, si considerino i due seguenti frammenti di programmi:

/programma 1/

```
int A[N], B[N];
…
for (i = 0; i < N; i++)
        A[i] = A[i] + B[i];
```

/programma 2/

```
int a; int B[N];
…
for (i = 0; i < N; i++)
        a = a + B[i];
```

In entrambi i casi si tratta di tradurre un comando di tipo *for*, contenente un comando di assegnamento su dati con tipi opportuni. Mentre nel primo caso ad ogni passo del *for* viene calcolato il nuovo valore di un elemento del vettore A, nel secondo caso ad ogni passo del *for* viene calcolato un nuovo valore di una stessa variabile scalare *a*. Un compilatore utilizzerà modalità diverse per produrre il codice oggetto del comando di assegnamento: nel programma 1 ad ogni passo del *for* provocherà la modifica della rappresentazione eseguibile del vettore A; nel programma 2, utilizzerà una rappresentazione temporanea dello scalare *a*, al quale verranno assegnati tutti i successivi valori, finché, all'uscita del *for*, la variabile temporanea verrà scritta nella rappresentazione finale della variabile *a*, disponibile per successivi usi da parte del seguito del programma. La rappresentazione temporanea di *a* è "più efficiente" della rappresentazione finale (come vedremo ampiamente in seguito, tipicamente il temporaneo risiede in un registro, mentre la rappresentazione finale risiede in una locazione di memoria principale, in quanto l'accesso in memoria comporta ritardi decisamente maggiori rispetto all'accesso ad un registro).

Un interprete, invece, traduce il comando di assegnamento del programma 1 e del programma 2 nello stesso modo: nel secondo caso non viene utilizzato un temporaneo, ma esiste un'unica rappresentazione della variabile *a* il cui valore viene ripetutamente modificato (viene letto *N* volte e scritto *N* volte il valore di *a* direttamente in memoria, effettuando $2*(N-1)$ accessi in memoria in più rispetto alla versione compilata).

In generale nella traduzione da parte di un interprete, poiché questa avviene passo passo, non viene considerato il contesto in cui si trova il comando da interpretare a ogni passo, applicando sempre *la stessa regola di traduzione* per uno stesso comando; invece, la traduzione da parte di un compilatore prende in considerazione una sequenza, più o meno ampia, in cui si trova inserito un comando e, per uno stesso comando, applica *regole di traduzione diverse* allo scopo di ottimizzare le prestazioni (ad esempio, il tempo di elaborazione del programma e/o l'occupazione di memoria). La fase di *ottimizzazione* è in effetti quella che caratterizza più pesantemente un compilatore: allo stato attuale della tecnologia, nel confronto tra due calcolatori di costruttori diversi, quello con il compilatore migliore può talvolta riuscire a colmare il gap prestazionale dovuto ad un processore peggiore.

Sempre per rimanere all'esempio precedente, si può vedere un'ulteriore occasione di ottimizzazione passando dalla versione interpretata alla versione compilata: a seconda del valore di *N*, l'esecuzione del *for* può consistere in una o più iterazioni (esecuzione tipo *do while*), oppure non aver luogo nessuna iterazione. Un interprete effettuerà sempre, all'inizio di un *for*, un controllo su *N* per distinguere le due situazioni, e, per $N \geq 0$, la gestione del loop verrà effettuata sempre allo stesso modo per qualunque valore di *N*. Invece, un compilatore sfrutterà la conoscenza sul valore di *N* disponibile staticamente: se $N > 0$ verrà generato codice assembler per la gestione di un loop ripetuto più di una volta (nell'esempio, essendo *N* la dimensione di un array, questo è l'unico caso), se $N = 0$ verrà generato codice per il solo corpo del *for* senza generare codice per la gestione del loop, se $N < 0$, non verrà generato codice, passando direttamente alla parte successiva del programma.

Una classe importante di ottimizzazioni a tempo di compilazione è quella relativa la miglior sfruttamento delle *caratteristiche architetturali*, ad esempio la memoria cache e/o l'architettura internamente parallela di processori.

Sulle caratteristiche di compilatori e interpreti torneremo frequentemente durante il corso. Vale fin da ora la pena di rimarcare che:

a) anche nel caso della pura interpretazione, è sempre presente "un minimo di compilazione", consistente nelle rappresentazione del programma sorgente in un formato facilmente comprensibile all'interprete;

b) raramente siamo in presenza di linguaggi soggetti a compilazione pura o a interpretazione pura. Ad esempio, anche in linguaggi che facciano soprattutto uso di compilazione, il supporto di eventuali

comandi per esprimere *strutture dati dinamiche* in modo primitivo (come *new* o *malloc)* è necessariamente implementato mediante interpretazione;

c) *a livello assembler ed a livello firmware l'unica forma di traduzione possibile è l'interpretazione*. La dimostrazione di questa importante affermazione emergerà dai concetti studiati durante il corso.

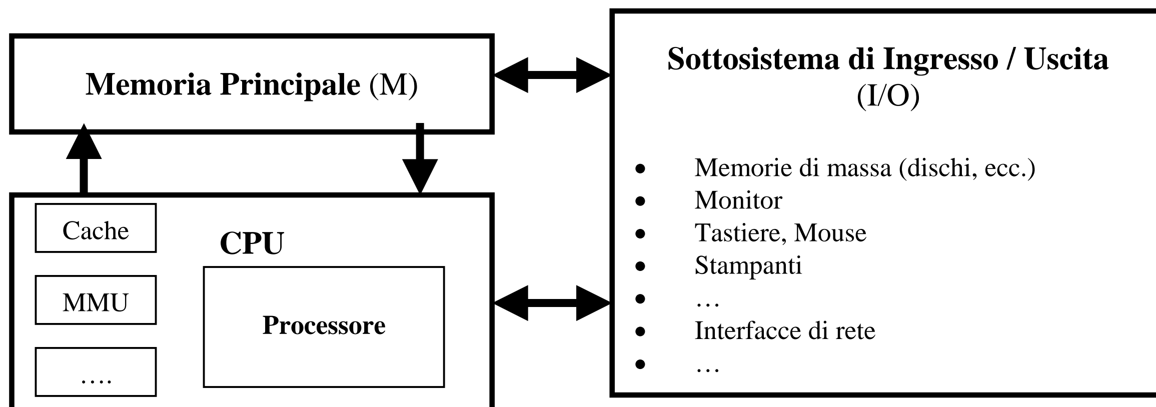
1.2. Strutturazione a moduli: unità di elaborazione e processi

La strutturazione a moduli sarà quella su cui ci baseremo per caratterizzare l'architettura di un qualunque sistema *a livello firmware*, tanto che sia un sistema specializzato, quanto un calcolatore general-purpose.

La strutturazione a moduli sarà usata in parti successive per caratterizzare le computazioni *a livello di applicazioni e di sistema operativo*.

Si ricorda che a livello firmware i moduli sono *unità di elaborazione*, mentre a livello di applicazioni e di sistema operativo i moduli sono *processi*. Questi due tipi di entità, che in maniera non scientifica sono ritenute due cose molto diverse e per certi aspetti conflittuali ("hardware" e "software" !), in realtà, da un punto di vista scientifico, sono due accezioni implementative di uno stesso concetto: questo permette di sviluppare una solida metodologia per la conoscenza e la progettazione di sistemi.

È importante aver presente che in un sistema a qualunque livello *una funzionalità complessa è ottenuta mediante composizione di funzionalità più semplici*: ognuna di queste, affidata ad un modulo, è specializzata verso determinati obiettivi, ma *la cooperazione* tra moduli rende possibile l'implementazione della funzionalità complessiva del sistema. Questa è la ragione per cui, già a livello firmware, la trattazione è basata sulla ed finzione e progetto di *unità specializzate*; un calcolatore general-purpose altro non è che una opportuna composizione di unità specializzate (processori, memorie, unità di I/O, unità di interfacciamento, unità di tradizione indirizzi, ecc), come mostrato schematicamente in figura:



2. Rappresentazione binaria delle informazioni

Nei sistemi di elaborazione, i numeri e, in generale, tutte le informazioni sono rappresentate come stringhe di bit mediante il *sistema binario*.

I normali numeri decimali consistono di una stringa di cifre decimali e, a volte, di una virgola per la separazione dei decimali. Il sistema di numerazione è basato sulla *notazione posizionale*, ed il valore di un numero è ottenuto con uno *sviluppo polinomiale*; ad esempio, per un numero naturale in *base* 10

$$5738 = 5 \times 10^3 + 7\times 10^2 + 3\times 10^1 + 8\times 10^0$$

Lavorando con i calcolatori, è spesso conveniente usare basi diverse da 10 e quelle più frequentemente usate sono 2 ed anche 8 e 16. I sistemi di numerazione basati su questi numeri sono rispettivamente chiamati binario, ottale ed esadecimale.

Un sistema a *base k* richiede *k* simboli differenti per rappresentare le cifre da 0 a *k - 1*. Così i numeri decimali sono scritti usando le 10 cifre decimali: 0 1 2 3 4 5 6 7 8 9. Invece i numeri binari sono costruiti

esclusivamente con le due cifre binarie: 0 1 (binary digit: *bit*). I numeri ottali sono costruiti con le otto cifre ottali: 0 1 2 3 4 5 6 7. Infine, per i numeri esadecimali, sono necessarie 16 cifre; così si richiedono sei nuovi simboli e, convenzionalmente, vengono usate le lettere maiuscole dalla A alla F per i sei simboli che seguono il simbolo 9: 0 1 2 3 4 5 6 7 8 9 A B C D E F

Il numero binario equivalente al numero decimale 5738 (in breve nel seguito useremo l'espressione "5738 in binario" o il simbolo 5738_2) è:

$$1011001101010 = 1 \times 10^{12} + 0 \times 10^{11} + 1 \times 10^{10} + 1 \times 10^{9} + 0 \times 10^{8} + 0 \times 10^{7} + 1 \times 10^{6} + 1 \times 10^{5} + 0 \times 10^{4} + 1 \times 10^{3} + 0 \times 10^{2} + 1 \times 10^{1} + 0 \times 10^{0} = 5738$$

2.1. Conversione decimale-binario e binario-decimale

Questi aspetti saranno di continua applicazione durante il corso, ragion per cui è necessario che siano del tutto familiari e che vengano utilizzati con disinvoltura e rapidità.

Anzitutto, è necessario avere presente la tabella delle potenze di 2 (funzione 2^j, con j numero naturale):

j	**2^j**		**j**	**2^j**
0	1		11	$2^1 \times 2^{10}$ = 2K
1	2		12	4K
2	4		20	$2^{10} \times 2^{10}$ = 1K × 1K = 1M (Mega)
3	8		21	2M
4	16		30	$2^{10} \times 2^{20}$ = 1K × 1M = 1G (Giga)
5	32		31	2G
6	64		32	4G
7	128		40	$2^{10} \times 2^{30}$ = 1K × 1G = 1T (Tera)
8	256		41	2T
9	512		ecc.	ecc.
10	1024 = 1K (Kilo)		ecc.	ecc.

È sufficiente ricordarsi le prime dieci righe e le potenze notevoli (Kilo, Mega, Giga, Tera, …), dopo di che qualunque numero si ottiene mediante facili prodotti di potenze note.

La **conversione da binario in decimale** si effettua, nel modo più semplice, applicando lo sviluppo polinomiale, come nell'esempio precedente di 5738_2.

La conversione da decimale in binario può essere fatta in due modi.

Il primo (del tutto generale) deriva direttamente dalla definizione di numero binario: dal numero decimale viene sottratta la massima potenza di 2 minore del numero stesso e lo stesso processo è ripetuto sulla differenza risultante. Una volta decomposto il numero in potenze di 2, si ottiene il numero binario ponendo 1 nelle posizioni di bit corrispondenti alla potenza di 2 usata nella decomposizione e 0 nelle restanti posizioni.

L'altro metodo (valido per i soli numeri interi) consiste nel dividere ripetutamente il numero decimale per 2, considerando i resti, finché si ottiene come quoziente il numero 0. Il numero binario è dato dalla stringa dei resti in ordine inverso rispetto a come sono stati ottenuti.

2.2. Numeri binari con segno

Per la rappresentazione dei numeri con segno sono usati principalmente tre metodi. In tutti, *il bit più significativo è il bit del segno*: 0 = +, 1 = –.

a) Modulo e segno: i bit restanti, tolto il bit del segno, rappresentano il valore assoluto (modulo) del numero. Ad esempio:

+ 25 = 011001

– 25 = 111001

Per quanto molto intuitivo, questo metodo non è sempre il più efficiente per l'aritmetica binaria. Più spesso si usano gli altri due metodi, ed in particolare il metodo in complemento a due.

b) **Complemento a uno**: se il numero è positivo, il numero in complemento a uno è uguale a quello modulo e segno, mentre, se il numero è negativo, si ottiene complementando bit a bit la rappresentazione modulo e segno. Ad esempio:

+ 25 = 011001

– 25 = 100110

c) Complemento a due: se il numero è positivo, il numero in complemento a due è uguale a quello modulo e segno, mentre, se il numero è negativo, si ottiene sommando uno alla rappresentazione in complemento a uno. Ad esempio:

+ 25 = 011001

– 25 = 100110 + 1 = 100111

L'eventuale riporto sul bit più a sinistra è semplicemente ignorato.

Sia il sistema modulo e segno che quello del complemento ad uno hanno due rappresentazioni possibili per il valore zero, cioè +0 e -0. Questa situazione è altamente indesiderata. *Il sistema del complemento a due non ha questo problema* perché il complemento a due di +0 è ancora +0, ma anch'esso rappresenta una singolarità. Infatti il numero costituito da un 1 seguito da tutti 0 è il complemento di sé stesso e questo fatto rende l'intervallo dei numeri positivi asimmetrico rispetto a quello dei negativi: esiste cioè un numero negativo senza il corrispondente controvalore positivo.

Qualsiasi insieme di numeri reali con tanti numeri negativi quanti sono i positivi ed un solo zero è costituito da un numero dispari di elementi, mentre *m* bit consentono di rappresentare un numero pari di elementi. Perciò ci sarà sempre o una configurazione di troppo o una di meno, indipendentemente dal tipo di rappresentazione scelto. Questa configurazione in più potrà essere usata per il valore -0, o per il massimo valore negativo, o per qualsiasi altra cosa, ma questo sarà sempre e solo una convenzione.

2.3. Nozioni di aritmetica binaria

La seguente tabella fornisce, in modo analogo rispetto a quanto siamo abituati nella numerazione decimale, la definizione della funzione addizione di due bit:

addendo	addendo	somma	riporto
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Con questa regola, due numeri binari possono essere sommati, a partire dal bit più a destra, sommando i corrispondenti bit nei due addendi. L'eventuale riporto è portato a sinistra di una posizione, proprio come nella somma decimale.

Nell'aritmetica in complemento ad uno il riporto generato dall'addizione sul bit più a sinistra è aggiunto al bit più a destra; questo processo e' chiamato di riporto circolare. Nell'aritmetica in complemento a due il riporto generato dall'addizione sul bit più a sinistra è semplicemente scartato.

Se due addendi sono di segno opposto non può capitare un errore di traboccamento (*overflow*), mentre se sono di ugual segno e il risultato è di segno opposto significa che si è avuto un errore di overflow e il risultato non è corretto. Nell'aritmetica sia del complemento ad uno che del complemento a due l'errore di overflow può capitare se e solo se il riporto che arriva al bit di segno differisce dal riporto che esce dal bit di segno. La maggior parte dei sistemi mantiene il riporto dal bit di segno ma dalla risposta non è visibile quello che entra nel bit di segno: per questa ragione e' presente normalmente uno speciale bit di overflow.

Queste nozioni verranno integrate durante il corso quando necessario o come esempio.

3. Sull'utilizzazione della rappresentazione binaria delle informazioni

In questa sezione applicheremo le nozioni di rappresentazione binaria ad alcuni casi che verranno usati con frequenza durante il corso.

3.1. Proprietà fondamentali

Una proprietà fondamentale dei numeri binari è la seguente:

- Con k bit si possono rappresentare tutti i numeri naturali che vanno da 0 a $2^k - 1$, estremi inclusi (cioè, il massimo numero naturale esprimibile con k bit è uguale a $2^k - 1$).

Altrettanto fondamentale è la seguente proprietà:

- Sia un insieme di m oggetti dello stesso tipo. Vogliamo denotare ogni oggetto con un numero naturale distinto, cioè un *identificatore* unico dell'oggetto, rappresentato in binario. Il minimo numero di bit necessario per rappresentare un qualunque identificatore è

$$n = \lceil \lg_2 m \rceil$$

Queste proprietà saranno applicate, ad esempio, con riferimento alle memorie e agli indirizzi di memoria, come vedremo in una prossima sez. di queste note.

Ad esempio, per un insieme di 1K oggetti, occorrono almeno 10 bit per identificarli univocamente tutti; per un insieme di 3000 oggetti, occorrono 12 bit.

Si osservi che $\lceil \lg_2 m \rceil$ è il *minimo* numero di bit necessari per identificare un oggetto. In alcuni casi, potrà capitare che il numero di bit per rappresentare gli identificatori sia un vincolo del problema e che sia ridondante rispetto al numero di oggetti: in tal caso, un qualunque identificatore avrà un certo numero di bit più significativi uguale a zero, oppure alcune configurazioni di bit non saranno significative.

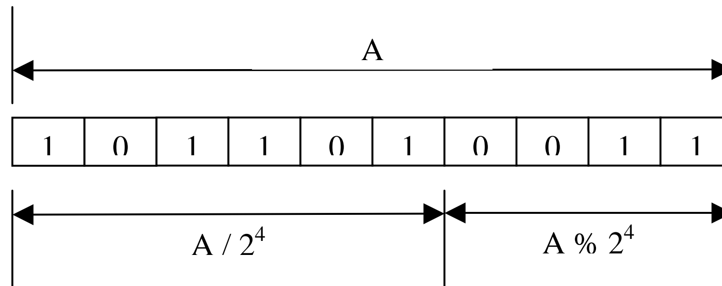
3.2. Alcune proprietà utili

a) Quoziente e resto della divisione per potenze di 2

Siano A e B due numeri binari corrispondenti a numeri naturali, e sia $B = 2^h$. In questa ipotesi, è molto facile il calcolo del quoziente della divisione intera e del resto (modulo):

- il valore di A%B (A *mod* b) è dato dagli h bit più significativi di A,
- il valore di A/B (A *div* B) è dato dai rimanenti bit più significativi di A.

Ad esempio sia A = 723 (A_2 = 1011010011), e B = 16 = 2^4, per cui A%B = 3 ⇒ 0011 (rappresentato dai 4 bit meno significativi di A) e A/B = 45 ⇒ 101101 (rappresentato dai rimanenti 6 bit più significativi di A):



Come caso particolare notevole:

- un numero binario *pari* (*dispari*) ha il bit meno significativo uguale a 0 (1).

b) Operazioni di traslazione (shift) e rotazione

Dato un numero binario naturale A, la sua traslazione destra (*shift destro*) di 1 bit

$$sh_R^1(A)$$

è il numero naturale che si ottiene spostando tutti i suoi bit a destra di una posizione e inserendo il valore 0 nel bit più significativo; il vecchio bit meno significativo viene perso.

Dato un numero binario naturale A, la sua traslazione sinistra (*shift sinistro*) di 1 bit

$$sh_L^1(A)$$

è il numero naturale che si ottiene spostando tutti i suoi bit a sinistra di una posizione e inserendo il valore 0 nel bit meno significativo; il vecchio bit più significativo viene perso.

Valgono le seguenti proprietà, facilmente dimostrabili (ad esempio, a partire dalla proprietà di *mod* e *div*):

- $sh_R^1(A) = A/2$
- $sh_L^1(A) = A * 2 \bmod 2^n$, dove n è il numero di bit di A.

Le proprietà sono generalizzabili alla traslazione di *k* posizioni, a destra $sh_R^k(A)$ ed a sinistra $sh_L^k(A)$:

- $sh_R^k(A) = A/2^k$
- $sh_L^k(A) = A * 2^k \bmod 2^n$

Quello finora definito è lo shift così detto *logico*, applicato di regola a numeri naturali. Se i numeri sono con segno, lo shift *aritmetico* è quello che si applica a tutti i bit tranne il bit del segno.

L'operazione di *rotazione* (destra o sinistra) di *k* posizioni è definita spostando circolarmente di *k* posizioni tutti i bit del numero naturale (rotazione logica; oppure rotazione aritmetica se non viene interessato il bit del segno). L'operazione di rotazione non comporta perdita di bit né inserzione di nuovi bit

c) Concatenamento di numeri binari

Sia A un numero naturale di *n* bit e B un numero naturale di *m* bit. La concatenazione di A e B

$$A \circ B$$

è il numero naturale di $n + m$ bit, che ha gli *n* bit più significativi uguali a quelli di A e gli *m* bit meno significativi uguali a quelli di B.

Ad esempio: A = 100110, B = 01, $A \circ B$ = 10011001.

Per ricavare il valore di $A \circ B$ a partire dai valori di A e B, consideriamo le rappresentazioni di A e di B su $n + m$ bit: quella di A (A') ha gli m bit più significativi tutti uguali a 0, quella di B (B') ha gli n bit più significativi tutti uguali a zero:

A' = 00 100110

B' = 000000 01

Trasliamo A' a sinistra di m posizioni e sommiamo il risultato a B':

$sh_L^m(A')$ = 100110 00 +

B' = 000000 01

ottenendo proprio $A \circ B$. Dunque, in generale vale la seguente proprietà:

$$A \circ B = A \times 2^m + B$$

d) Contatori

Nell'implementazione di loop a livello firmware, ci troveremo spesso nella situazione di valutare la condizione di uscita da un *for* ripetuto per un numero di volte *N*.

Consideriamo prima il caso che $N = 2^n$. Per scandire i passi del *for*, consideriamo una variabile *I* inizializzata al valore 0 e incrementata ad ogni passo. I viene rappresentata su $n + 1$ bit. Quando I assume il valore 2^n significa che sono già stati effettuati gli *N* passi. Per cercare un predicato il più possibile semplice, abbiamo che:

- la condizione di uscita dal *for* è che il bit più significativo di I assuma il valore 1.

Se non vogliamo fare ipotesi su N, consideriamo I rappresentata su

$$\lceil \lg_2 N \rceil \text{ bit}$$

inizializziamo I al valore *N-1*, e decrementiamola ad ogni passo. La condizione di uscita dal *for* è che I assuma il valore –1. Per cercare un predicato il più possibile semplice, abbiamo ancora che:

- la condizione di uscita dal *for* è che il bit più significativo di I assuma il valore 1 (ora si tratta del bit del segno).

4. Strutture di calcolo, tipi di dato e rappresentazione binaria delle informazioni

È importante acquisire sensibilità con il modo con cui la rappresentazione binaria verrà utilizzata ai livelli firmware e assembler.

4.1. Parole di informazione

Le informazioni a livello di macchina firmware e di macchina assembler sono rappresentate come *stringhe di bit*, che possono avere il significato più vario a seconda dell'uso che intendiamo farne:

- numeri naturali, o relativi, o reali,
- caratteri,
- codici per identificare oggetti (identificatori, indirizzi)
- ecc.

In generale, useremo il termine *parola* per denotare una stringa di bit significativa. Ad esempio, la parola può consistere nei bit che compongono una locazione (cella) di memoria, cioè i bit che possono essere letti o scritti contemporaneamente in una certa posizione di una memoria. In alcuni casi saremo interessati a concatenare più informazioni in una stessa parola, e in questo caso riconosceremo altrettanti *campi* della parola.

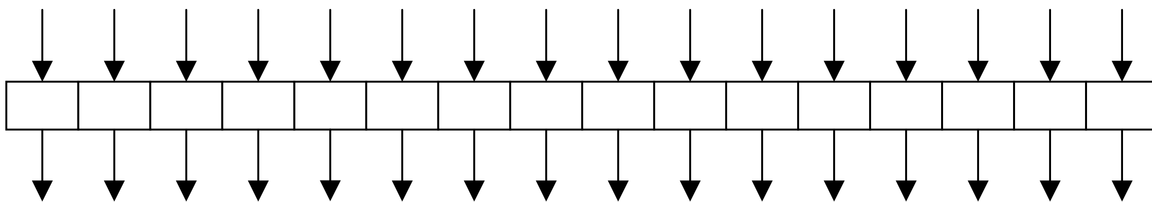
Ognuna di queste informazioni avrà una certa *lunghezza*, data dal numero di bit che la compongono. Ad esempio, un calcolatore general-purpose sarà caratterizzata da un parola di 32 bit o di 64 bit (a seconda della tecnologia): cosa rappresentare con una sola parola, o con un sottoinsieme di byte (1 byte = 8 bit) della parola, o con più parole, sarà oggetto di opportune scelte architetturali. Ad esempio, in un calcolatore con parola di 32 bit, le istruzioni possono essere rappresentate su una parola, gli interi double su una parola, gli interi short su mezza parola, i caratteri su un byte.

In tutta la trattazione che faremo ai diversi livelli, sarà sempre *nota la lunghezza* delle informazioni di interesse, o sarà *nostro compito determinarla a partire* da altre informazioni note.

Ad esempio, nella progettazione di una certa unità di elaborazione, supponiamo che sia un dato del problema il fatto che tale unità applica una certa funzionalità a dati di 32 bit. Supponiamo anche che, per implementare l'algoritmo corrispondente a tale funzionalità, sia necessario eseguire un *for* ripetuto esattamente 32 volte. Se ne deduce che, per il controllo del *for*, dovremo fare uso di una variabile contatore di 6 bit, applicando una delle due tecniche descritte nella sez. 2.2.d).

Come altro esempio, se la capacità (massima) C di una memoria è un dato del problema, è immediato dedurre quanti bit sono necessari per rappresentare i suoi indirizzi (vedi sez. 3.4).

Una parola di informazione



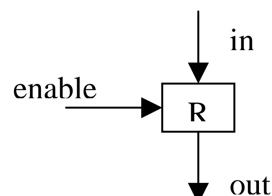
potrà essere manipolabile secondo certi vincoli noti. Ad esempio:

- è possibile leggere (testare) ogni bit singolarmente,
- è possibile modificare tutta la parola, ma non i singoli bit separatamente,
- ecc.

4.2. Registri

Fondamentale per la strutturazione a livello firmware ed a livello assembler è il registro, che funge da *componente con memoria (con stato)*.

Un *registro di un bit*, rappresentato convenzionalmente come:



ha il compito di memorizzare, per tutto il tempo che si ritiene necessario, l'informazione (1 bit) presente sull'ingresso *in* e di rendere disponibile tale informazione (*contenuto*) sull'uscita (1 bit) *out*. Per scrivere (memorizzare) il valore di *in*, è necessario che sia presente un *segnale di controllo* (1 bit, indicato in figura

con *enable*): la scrittura avviene se *enable = 1*. Finché *enable = 0*, successivi valori che si presentino su *in* vengono perduti (non memorizzati), mentre appena *enable = 1* il valore che in quel momento è presente sull'ingresso *in* viene memorizzato (diviene il *contenuto* del registro) e comparirà sull'uscita *out*. Il valore su *out* può sempre essere letto ed elaborato (un valore è *sempre* presente, non occorre abilitazione per utilizzarlo in lettura).

Nella trattazione delle reti logiche e delle unità di elaborazione, faremo uso di *registri impulsati*, nei quali il segnale *enable* è messo in AND con un segnale, detto *clock*, impulsivo e periodico. Il periodo, cioè l'intervallo tra l'inizio di due impulsi di clock consecutivi, è detto *ciclo di clock*. La scrittura nel registro avviene in corrispondenza di un impulso di clock a condizione che contemporaneamente *enable = 1*. Questa caratteristica permetterà un funzionamento sincrono della reti (Parte Operativa e Parte Controllo) costituenti ogni unità di elaborazione.

Questo componente, ed il suo funzionamento, sono assolutamente fondamentali per il corso. *Qualunque memoria* (svariati tipi di memoria verranno studiati nelle varie parti del corso) è, in ultima analisi, costituita dalla composizione di componenti elementari come il registro di un bit.

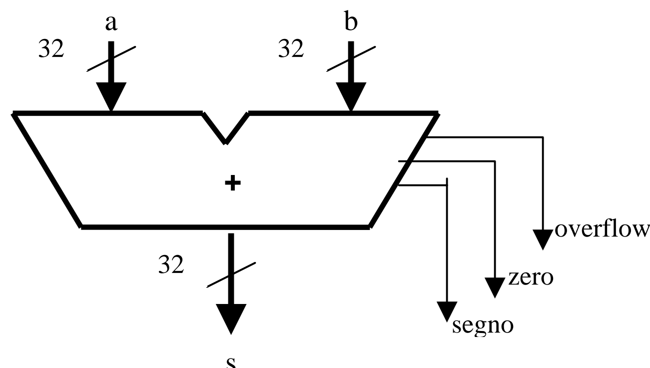
Ad esempio *un registro a 16 bit* può essere schematizzato come nella figura alla fine della sez. precedente, stante a significare che il registro a 16 bit è ottenuto come *composizione in parallelo* di 16 registri ognuno di 1 bit. Tutti i registri di 1 bit hanno lo stesso segnale *enable*; ciò significa che, con tale registro a 16 bit, possiamo:

- *scrivere una parola di 16 bit*: ciò si ottiene presentando i 16 bit della parola sull'ingresso e ponendo *enable = 1*. Poiché il segnale di controllo è lo stesso per tutti i bit, non è invece possibile scrivere singolarmente in alcuni bit e lasciare inalterati gli altri;
- usare tutta la parola di 16 bit in uscita (il contenuto del registro), ad esempio applicarvi una funzione definita sul tipo di dato corrispondente a quella parola;
- usare uno o più campi in uscita separatamente, ad esempio applicarvi una funzione definita sul tipo di dato corrispondente a tali campi;
- usare uno o più bit in uscita separatamente, ad esempio per effettuare un test sul contenuto attuale, come:
  - test del bit del segno di un numero,
  - test del bit meno significativo di un numero per sapere se tale numero è pari o dispari,
  - test del bit *h*-esimo di un numero per sapere se tale numero è maggiore o uguale a 2^h.

4.3. Funzioni e reti combinatorie

A livello hardware implementeremo funzioni "pure" (computazioni *senza* stato: per ogni valore dell'ingresso si ha sempre uno ed un solo valore dell'uscita) mediante opportune strutture dette *reti combinatorie*, ottenute a loro volta come composizioni di componenti più elementari.

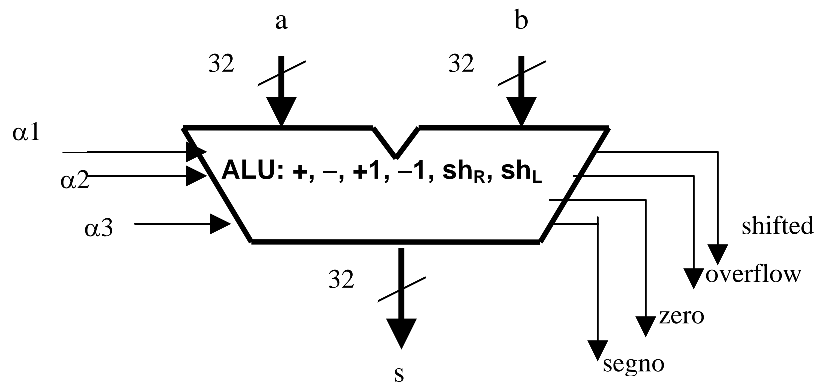
Ad esempio, una funzione molto utilizzata sarà la somma di numeri con segno. In questo caso, vedremo come realizzare una rete combinatoria *addizionatore*, avente (vedi figura):



- due ingressi, *a* e *b*, che rappresentano numeri con segno (ad esempio su 32 bit). Tali valori potrebbero essere uscite di registri;
- una uscita primaria, *s*, che, istante per istante, rappresenta la somma dei valori presenti sugli ingressi;
- tre uscite secondarie, ognuna di un bit ("flag"), che forniscono rispettivamente il valore del segno del risultato, un booleano che indica se il risultato della somma è uguale a zero, l'indicazione dell'eventuale traboccamento (overflow) della somma.

Un caso più completo è quello delle così dette **ALU**: si tratta di **reti di calcolo multifunzione**. A seconda del valore di alcune variabili d'ingresso secondarie (*variabili di controllo*), il valore dell'uscita è dato dall'applicazione agli ingressi primari di una ben specifica funzione: addizione, sottrazione, confronto, shift, AND bit a bit, OR bit a bit, ecc. a seconda di quali operazioni interessano di volta in volta nella progettazione di una unità di elaborazione.

Nella figura seguente è rappresentata una ALU a 32 bit capace di eseguire le operazioni di addizione, sottrazione, incremento di uno sull'ingresso sinistro, decremento di uno sull'ingresso sinistro, shift destro di una posizione sull'ingresso sinistro, shift sinistro di una posizione sull'ingresso sinistro:



Sono presenti tre variabili di controllo binarie, α1, α2, α3, per identificare l'operazione che vogliamo applicare agli ingressi primari. La semantica, riferita all'uscita primaria, è:

s = *case* α1 α2 α3

000: a + b;

001: a – b;

010: a + 1

011: a – 1;

10–: sh_R(a)

11–: sh_L(a)

Si noti che, dovendo identificare 6 operazioni, abbiamo bisogno di $\lceil \lg_2 6 \rceil = 3$ variabili di controllo. Le codifiche scelte per gli identificatori (ad esempio, con quale *codice* rappresentare l'addizione) sono arbitrarie, purché *utilizzino in modo consistente e completo le variabili di controllo*. Nel nostro caso, abbiamo scelto di rappresentare le prime quattro operazioni (+,–, +1, -1) con i codici 0, 1, 2, 3 in binario (000, 001, 010, 011).

A questo punto, per rappresentare le due operazioni di shift abbiamo a disposizione i quattto codici 4, 5, 6, 7: abbiamo scelto che lo shift destro sia identificato dal valore 4 (100) *oppure* dal valore 5 (101), e lo shift sinistro dal valore 6 (110) *oppure* dal valore 7 (111). Nelle scritture "10–" e "11–" il simbolo "–" sta a significare "*non specificato*": qualunque valore della variabile di controllo α3 va bene per identificare una di queste due operazioni.

Dalla sez. 3 sappiamo che, in generale, dovendo identificare un numero n di oggetti che non sia una potenza di 2 (nel nostro caso, $n = 6$ operazioni), non è corretto (non è possibile) usare esattamente n codici distinti, bensì che è necessario usare

$$2^{\lceil \lg_2 n \rceil}$$

codici: di questi, alcuni corrispondono allo stesso oggetto, quindi c'è una certa *ridondanza* nella codifica. In talune situazioni, potrà anche accadere che qualcuno dei codici ridondanti non possano mai presentarsi. Questo aspetto non ha però alcuna importanza ai nostri effetti, perché un "risparmio" nel numero di codici ammessi non comporterebbe alcun "risparmio" nel numero delle variabili di controllo.

Sempre nell'esempio della ALU precedente, è stata introdotta una ulteriore uscita secondaria che assume il valore del bit "espulso" (perso) nelle operazioni di shift.

Le quattro uscite secondarie assumono sempre un certo valore, qualunque sia l'operazione eseguita dalla ALU; questo valore potrà essere significativo e meno a seconda dei casi. Ad esempio, per qualunque operazione può essere significativo conoscere il segno del risultato, oppure se il risultato è uguale a zero; il valore del bit di overflow è significativo solo in seguito ad una operazione di addizione o di incremento (ma "esiste" anche eseguendo una delle altre operazioni); il valore del bit "shifted" è significativo solo in seguito ad una operazione di shift (ma "esiste" anche eseguendo una delle altre operazioni).

In effetti, la semantica completa della ALU deve fornire l'espressione del valore delle *cinque uscite (s, segno, zero, overlow, shifted)* in corrispondenza delle configurazioni binarie delle variabili di controllo. Si lascia per esercizio scrivere la semantica completa.

Ogni unità di elaborazione (modulo a livello firmware) conterrà opportune combinazioni di registri impulsati e di reti combinatorie, allo scopo di realizzare le funzionalità (anche complesse) richiestele.

4.4. Memorie

Come introdotto informalmente nella sez. 3.2, un registro (di n bit) rappresenta la base per implementare il concetto di memoria.

Normalmente, quella che va sotto il nome di memoria è costituita da più registri, detti *locazioni* o *celle*. Il numero (massimo) di locazioni che una memoria può contenere è detto la sua *capacità*. Ad esempio, nel corso vedremo l'uso di memorie aventi capacità le più varie: da poche decine di locazioni (ad esempio 64), a poche migliaia (ad esempio, 4K), a pochi milioni (ad esempio, 8M), a qualche miliardo (ad esempio, 1G), ed oltre (capacità dell'ordine dei Terabyte per dischi recenti).

Formalmente, le memorie sono l'implementazione firmware del tipo di dato *array*: una memoria M di capacità C, è di fatto un array di C elementi:

tipo M[C]

dove ogni elemento (locazione di n bit) è un *tipo* elementare rappresentabile su singola parola.

Ogni locazione di memoria è accedibile solo specificandone l'indice. Tale indice prende il nome di **indirizzo** della memoria. La notazione familiare

M[I]

sta esattamente a significare il contenuto della locazione di indirizzo I. La semantica di tale scrittura è:

M[I] = *case* I

0: M[0];

1: M[1];

…

C-1: M[C-1]

Per quanto apparentemente banale, questa semantica si rivelerà di estrema utilità in varie parti del corso.

Per rappresentare un qualunque indirizzo di (per indirizzare) una memoria di capacità C, occorrono (almeno):

$$\lceil \lg_2 C \rceil \text{ bit}$$

Di fatto, in tutti i casi reali si lavora con memorie aventi capacità uguali a potenze di 2.

Ad esempio,

- una memoria di capacità 64K parole necessita di indirizzi di 16 bit;
- se è noto a priori che l'indirizzo di una memoria è ampio 32 bit, tale memoria può avere una capacità massima di 4G parole.

Il concetto di memoria e di indirizzo è fondamentale per questo corso, e verrà approfondito in diverse occasioni.

Fin da ora deve essere chiaro che:

- *una locazione di memoria può essere acceduta (letta o scritta) solo specificandone l'indirizzo.*

Nessun'altra modalità è consentita in maniera primitiva.

Ciò non toglie che, come siamo abituati nella programmazione, *tipi di dato più complessi* non possano essere implementati mediante array ed altri tipi elementari. Ad esempio, anche a livello firmware si può implementare

- una coda o una pila, usando un vettore (una memoria) e qualche registro per conoscere la posizione a cui accedere (l'indirizzo della memoria da leggere o scrivere),
- una tabella accedibile per contenuto, implementando un algoritmo di ricerca oppure realizzando opportunamente la trasformazione della chiave in un indirizzo hash,
- ecc.

5. Guida allo studio della strutturazione firmware

Sulla base delle sez. precedenti, lo studio della prima parte del corso procede come segue:

- **Reti combinatorie**: Cap. II, sez. 1 della Dispensa + integrazione sull'algebra booleana in queste note, sez. 6,
- **Reti sequenziali**: Cap. II, sez. 2.1, 2.2, 2.3 della Dispensa,
- **Firmware e Modello di Unità di Elaborazione**:
    - o Cap. III della Dispensa,
    - o integrazioni in queste note, sez. 7,
- **Comunicazioni tra unità**: materiale presente in queste note, sez. 8, in sostituzione del Cap. IV della Dispensa.

La sez. 9 contiene testi di quesiti ed esercizi.

6. Elementi di algebra booleana della commutazione

Questa sezione sostituisce il riferimento a pag.2 del Cap. I della Dispensa: S. Antonelli "Rappresentazione dell'Informazione e Algebre Booleane".

L'algebra della commutazione è un sistema algebrico in cui ogni variabile può assumere uno solo tra due valori, 0 e 1, nel quale sono applicate alle variabili le operazioni binarie *di moltiplicazione logica* e *somma logica* e l'operazione unaria di *complementazione* o *negazione*. Queste operazioni, anche dette **funzioni logiche base**, sono chiamate AND (∧), OR (∨) e NOT (¯) e sono definite come indicato di seguito. Quando non sorga ambiguità, al posto di A ∧ B (A ∨ B) verrà usata la notazione semplificata AB (A+B).

A	B	A ∧ B = AB
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A ∨ B = A+B
0	0	0
0	1	1
1	0	1
1	1	1

A	$\overline{A}$
0	1
1	0

Queste definzioni costituiscono i *postulati di base dell'algebra della commutazione*. Altre proprietà notevoli, tutte ricavabili da questi postulati, possono essere enunciate mediante i seguenti *teoremi*:

Complementazione:

$$A+\overline{A}=1 \qquad A\overline{A}=0$$

Involuzione:

$$\overline{\overline{A}}=A$$

Potenza identica:

$$A+A=A \qquad AA=A$$

Unione e intersezione

$$A+0=A \qquad A1=A$$

$$A+1=1 \qquad A0=0$$

Proprietà commutativa:

$$A+B=B+A \qquad AB=BA$$

Proprietà associativa:

$$A+(B+C)=(A+B)+C \qquad A(BC)=(AB)C$$

Proprietà distributiva:

$$A(B+C)=AB+AC \qquad A+(BC)=(A+B)(A+C)$$

Teorema di De Morgan:

$$\overline{A+B}=\overline{A}\,\overline{B} \qquad \overline{AB}=\overline{A}+\overline{B}$$

Il metodo più semplice per dimostrare i teoremi T_1-T_8 è quello cosiddetto della "perfetta induzione" che

consiste nel verificare l'uguaglianza delle relazioni per tutte le combinazioni dei valori delle variabili. Questo metodo si può facilmente applicare perché ogni variabile può assumere solo due valori.

Dalle relazioni che enunciano i teoremi precedenti notiamo che ciascuna di queste (tranne la seconda) è affiancata da un'altra relazione che si ottiene dalla prima sostituendo AND con OR e viceversa, ed ogni 0 con 1 e viceversa. L'algebra della commutazione gode infatti della proprietà, insita nella simmetria delle operazioni di AND e OR, di poter trasformare, con la regola ora enunciata, una data relazione in un'altra relazione chiamata *duale*.

Una funzione logica può essere rappresentata sia da una **tabella di verità**, sia da una **espressione algebrica**. Tuttavia, *mentre c'è una sola tabella di verità per ogni funzione, vi sono molte espressioni che possono rappresentare una stessa funzione*.

Per esempio, le due espressioni:

$$\overline{x}y\overline{z} + x\overline{y} + z$$

$$\overline{x}z + \overline{x}y + xz + x\overline{y}$$

rappresentano la stessa funzione, definita dalla seguente tabella di verità:

x y z	*f*
0 0 0	0
0 1 0	1
1 0 0	1
1 1 0	0
0 0 1	1
0 1 1	1
1 0 1	1
1 1 1	1

Due espressioni che rappresentano la stessa funzione sono chiamate *equivalenti*.

Tra tutte le espressioni equivalenti di una stessa funzione ve ne sono due di particolare interesse che hanno forme ben definite. Chiamiamo *lettera* una variabile affermata o complementata e *termine* un prodotto di lettere (termine prodotto) o una somma di lettere (termine somma) che compare in un'espressione. Chiamiamo *mintermine* un termine prodotto e *maxtermine* un termine somma che contengono un numero di lettere di variabili distinte uguale al numero *n* delle variabili della funzione rappresentata dall'espressione.

Indichiamo con SP e PS un'espressione in Somma di termini Prodotto (o Somma di Prodotti) e Prodotti di termini Somma (o Prodotti di Somma), rispettivamente. Chiamiamo infine *forma normale* la forma di un'espressione SP.

Chiameremo **forma canonica** di una funzione in *n* variabili l'espressione in SP o PS in cui ogni termine è un mintermine o un maxtermine. Per esempio, l'espressione

$$\overline{x}yz + x\overline{y}\overline{z} + \overline{x}\overline{y}z + \overline{x}yz + x\overline{y}z + xyz$$

è la forma canonica della funzione definita nella tabella precedente. Si può facilmente dimostrare che c'è una sola forma canonica in SP ed una sola forma canonica in PS per ciascuna funzione.

Gli altri elementi di algebre della commutazione, per lo studio delle reti combinatorie, sono dati direttamente nel Cap. I, sez. 1, della Dispensa.

7. Reti sequenziali e strutturazione firmware: una rilettura per agevolarne la comprensione

Questa sezione intende ripercorrere tutta la trattazione dei livelli hardware-firmware, ed in particolare la relazione tra strutturazione firmware e reti sequenziali, allo scopo di agevolare la comprensione dei concetti che vi sono contenuti. Da questo punto di vista, questa sez. non contiene niente di nuovo rispetto ai capitoli della Dispensa, ma espone i concetti legati alle reti sequenziali in un modo più intuitivo e connesso alla microprogrammazione.

7.1. Parte Controllo e Parte Operativa come reti sequenziali

7.1.1. Stato interno e funzione di transizione dello stato interno

La Parte Controllo (PC) e la Parte Operativa (PO) di una unità di elaborazione sono reti sequenziali, cioè implementazioni hardware di *automi a stati finiti*.

L'unità di elaborazione è quindi costituita dalle due reti sequenziali, PC e PO, tra loro interconnesse e sincronizzate dallo stesso clock.

Sia PC che PO hanno infatti un loro *stato interno*.

Dato il microprogramma che descrive l'unità,

- lo *stato interno di PC* è in corrispondenza biunivoca con le etichette delle microistruzioni, ed è rappresentato dal contenuto di un apposito registro di stato del controllo (RC) [1];
- lo *stato interno di PO* è rappresentato dai contenuti di tutti i registri di PO, inclusi i registri di ingresso[2].

Il funzionamento di PC e di PO è scandito da un segnale impulsivo (*clock*) applicato contemporaneamente a tutti i registri dell'unità. La scrittura in un qualunque registro corrisponde ad una *variazione di stato interno*, ed avviene solo se, e quando, al registro perviene l'impulso di clock. Per ogni registro di PO, tranne quelli d'ingresso (vedi nota a fondo pagina), l'impulso di clock è messo in AND con una variabile di controllo (β) per l'abilitazione alla scrittura, mentre la scrittura nel registro RC di PC e nei registri d'ingresso di PO è abilitata ad ogni impulso di clock (assenza di variabile β, oppure β sempre uguale ad uno).

Sia per PC che per PO, *durante un ciclo di clock, l'uscita* dei registri (il valore assunto dall'insieme dei contenuti di tutti i bit di tutti i registri) rappresenta lo stato interno *presente*. Esso rimane *stabile*, per tutta la durata del ciclo di clock, ed uguale al valore assunto all'inizio del ciclo di clock ("fronte di discesa" dell'impulso).

Prima dell'inizio ("fronte di salita") del prossimo impulso di clock, gli ingressi di tutti i registri devono assumere di nuovo valori stabili. In questa situazione, l'insieme dei valori di tutti i bit *all'ingresso* di tutti i registri rappresenta lo stato interno *successivo*, cioè lo stato interno *che diventerà automaticamente* (grazie al meccanismo del clock) *stato interno presente all'inizio del prossimo ciclo di clock*.

Quindi, sia PC che PO devono implementare una loro *funzione di transizione dello stato interno*, cioè una funzione che, operando sullo stato interno presente e sui valori delle variabili d'ingresso (variabili di condizionamento per PC, variabili di controllo per PO), produce il valore dello stato interno successivo.

La funzione di transizione dello stato interno sarà implementata da *una rete combinatoria in PC*, detta σ_{PC}, e da *una rete combinatoria in PO*, detta σ_{PO}.

La specifica di tali reti combinatorie è data dal microprogramma stesso, quindi la loro definizione e implementazione deriva dai metodi per implementare funzioni a livello hardware come reti combinatorie.

Ad esempio, si consideri la microistruzione a struttura di frase:

---

[1] Solo nel caso particolare in cui il microprogramma consti di una sola microistruzione, PC non ha stato interno, ed è quindi implementata da una rete combinatoria.

[2] Lo scopo dei registri d'ingresso è solo quello di assicurare la stabilizzazione dei valori d'ingresso durante un ciclo di clock. La parte di stato interno che riguarda questi registri può essere modificata solo con valori provenienti dall'esterno.

2. *$(A_0 = 0)$ A – B → A, 0; $(A_0 = 1)$ sh_{R1} (A) → A, 1*

All'inizio del ciclo di clock in cui essa viene eseguita, lo stato interno presente di PC (uscita di RC) vale 2. Se la variabile di condizionamento A_0 vale zero, allora la funzione di transizione dello stato interno di PC dovrà produrre, a partire dai valori RC = 2 e $A_0 = 0$, il valore $in_{RC} = 0$ (ingresso di RC) in forma stabile prima che pervenga il prossimo impulso di clock: questo valore rappresenta lo stato successivo di PC, che diventerà stato presente di PC all'inizio del prossimo ciclo di clock; in tal modo, nel prossimo ciclo di clock venga eseguita la microistruzione 0. Altrimenti, se $A_0 = 1$, PC transirà dallo stato interno 0 allo stato interno 1.

Per la stessa microistruzione, lo stato presente di PO è dato dal contenuto corrente di A e B e di tutti gli altri eventuali registri di PO. Se viene eseguita la microperazione A – B → A, il valore stabile di A – B, presente sull'ingresso di A prima della fine del ciclo di clock, rappresenta (una parte de) lo stato interno successivo di PO, che diventerà stato presente all'inizio del prossimo ciclo di clock. *La transizione dello stato interno di PO è dunque il risultato dell'esecuzione della microperazione*.

Finora abbiamo visto, attraverso gli esempi di PC e PO, due aspetti fondamentali delle reti sequenziali:

- il concetto di *stato interno*
- il concetto di *funzione di transizione dello stato interno*.

Tali concetti sono del tutto generali, in quanto caratterizzano qualunque automa a stati finiti. Se un automa è implementato a livello hardware da una rete sequenziale, l'implementazione dello stato interno si ottiene mediante registri impulsati e quella della funzione di transizione dello stato interno mediante una *rete combinatoria*: di quest'ultima, gli ingressi sono costituiti dalle variabili dello stato interno presente (uscite dei registri) e dalle variabili di ingresso, e le uscite sono i nuovi valori che verranno assunti dagli ingressi dei registri stessi.

7.1.2. Funzione delle uscite

Oltre che dalla funzione di transizione dello stato interno, un automa è caratterizzato anche dalla

- *funzione delle uscite*.

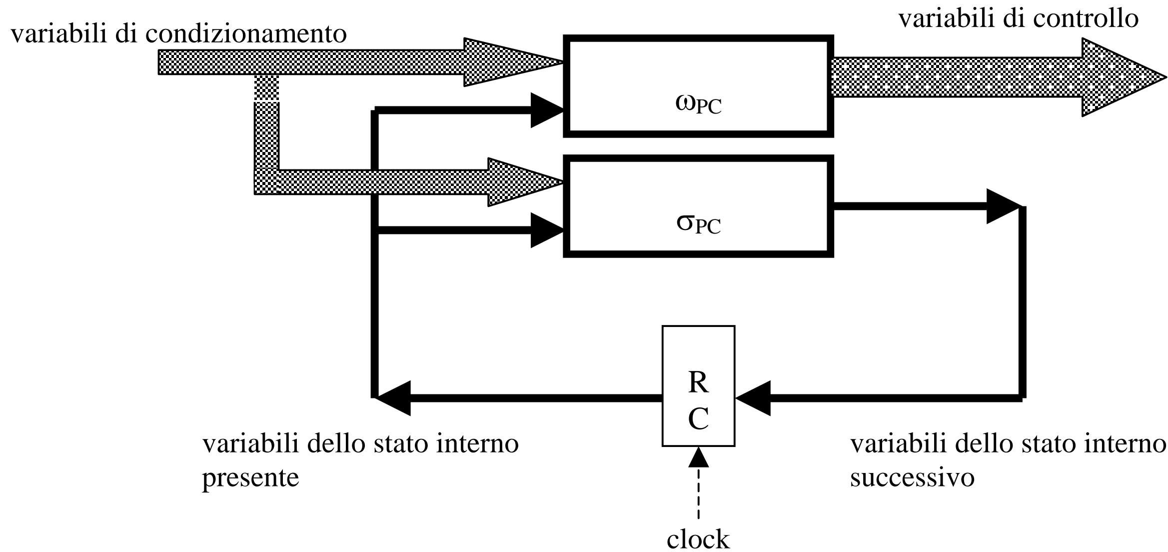
Consideriamo il caso di PC dell'unità dell'esempio precedente. Partendo dallo stato presente RC = 2 e con stato d'ingresso definito univocamente da $A_0 = 0$, PC deve produrre i valori delle variabili di controllo che permettono l'esecuzione della microperazione A – B → A nella PO: si tratta dei valori di eventuali (in numero di zero o più) variabili $\{\alpha\}$, per controllare ALU e commutatori, ed il valore della variabile β_A che abilita la scrittura nel registro A. Questi valori sono uscite di una rete combinatoria , detta ω_{PC}, che implementa la *funzione delle uscite* dell'automa PC. Tale rete ha *come ingressi le variabili dello stato interno presente di PC (uscite di RC) e le variabili di condizionamento*, e *come uscite le variabili di controllo*.

Come si nota riprendendo la spiegazione della sez. precedente, *la rete combinatoria* ω_{PC} *opera sugli stessi ingressi della rete combinatoria* σ_{PC}: queste due reti iniziano quindi a stabilizzarsi contemporaneamente, continuano a stabilizzarsi in parallelo, e devono concludere la stabilizzazione entro il prossimo impulso di clock. Spesso il ritardo di stabilizzazione delle due reti di PC è uguale, in generale una delle due potrà stabilizzarsi prima dell'altra.

Complessivamente, lo schema della rete sequenziale PC è quindi del tipo mostrato in figura a pagina seguente.

Si consideri ora la funzione delle uscite di PO, che verrà implementata da una rete combinatoria detta ω_{PO}.

Alcune uscite di PO sono ovvie: si tratta delle uscite esterne dell'unità che, come sappiamo, sono sempre uscite di appositi registri. Per tali uscite, la funzione ω_{PO} è quindi la funzione *identità* rispetto ai contenuti dei registri di uscita. È importante notare che, per quanto riguarda le uscite esterne, *la rete combinatoria* ω_{PO} *non ha in ingresso nessun altra variabile tranne le uscite di detti registri*: in altre parole, *il risultato della funzione* ω_{PO} *dipende, ad ogni ciclo di clock, solo dallo stato interno di PO e non dallo stato d'ingresso costituito dalla configurazione delle variabili di controllo*.



Agli effetti della funzione delle uscite di PO, assai più significativa è la parte relativa alle *variabili di condizionamento*. Nell'esempio precedente, A_0 è una variabile di condizionamento, e quindi è una variabile di uscita di PO. Come le precedenti, anche il valore di questa variabile di uscita *dipende, ad ogni ciclo di clock, solo dallo stato interno di PO (uscita del registro A_0) e non dallo stato d'ingresso costituito dalla configurazione delle variabili di controllo* [3].

Nell'esempio, la funzione ω_{PO} è la funzione identità anche per quanto riguarda l'uscita A_0. Mentre, come detto, la funzione identità è la regola per le uscite esterne, non è detto che ciò avvenga sempre per le variabili di condizionamento. Ad esempio, il microprogramma può prevedere l'uso di variabili di condizionamento più complesse come:

- segno (A + B)
- zero (M – N)
- A[J]
- or (C)
- and (J)
- or (E ⊕ F)

In tali casi, ω_{PO} è data dalle funzioni, rispettivamente: segno o zero del risultato di una operazione di una ALU, commutazione del bit J-esimo del registro A con J uscita di registro, OR o AND bit-a-bit dei bit di un registro, OR bit-a-bit applicato al risultato di una operazione aritmetico-logica o di commutazione (nell'esempio, OR esclusivo). *Le reti combinatorie ω_{PO} corrispondenti devono operare, ad ogni ciclo di clock, solo su uscite di registri.*

La condizione:

> "*lo stato di uscita di PO dipende, ad ogni ciclo di clock, solo dallo stato interno presente e non dallo stato d'ingresso*"

deve valere in generale, qualunque sia la funzione che produce i valori delle variabili di condizionamento.

Al contrario, si ricordi che questa condizione non vale per ω_{PC}.

---

[3] Si noti che il valore dell'uscita di A_0 *non* dipende, ad ogni ciclo di clock, dalla variabile di controllo β_A: questa ha avuto impatto solo sulla *funzione di transizione dello stato interno al ciclo di clock immediatamente precedente*.

Nella teoria generale degli automi e reti sequenziali (Cap. II, sez. 2) la differenza tra i comportamenti che abbiamo visto per ω_{PC} e per ω_{PO} è alla base della differenza tra due modelli matematici di automa, detti modello di Mealy e modello di Moore. Nel modello di *Mealy*, la funzione delle uscite ad ogni istante (ad ogni ciclo di clock) dipende dallo stato interno presente e dallo stato d'ingresso in quell'istante (in quel ciclo di clock), mentre nel modello di *Moore* la funzione delle uscite ad ogni istante (ad ogni ciclo di clock) dipende solo dallo stato interno presente in quell'istante (in quel ciclo di clock). In entrambi i modelli, la funzione di transizione dello stato interno dipende dallo stato interno presente e dallo stato d'ingresso.

Questa teoria non verrà approfondita, ma ne verranno semplicemente utilizzati alcuni risultati senza dimostrazione formale. In pratica, i termini "modello di Mealy" e "modello di Moore" sono delle semplici etichette per denotare le due caratterizzazioni finora discusse per la funzione delle uscite.

Il seguente teorema non verrà dimostrato formalmente:

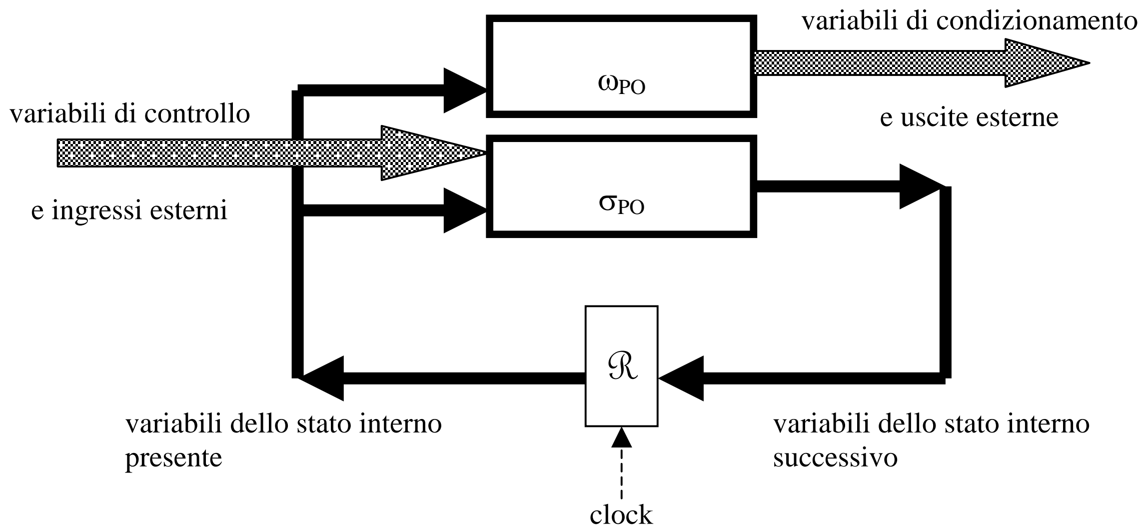
in una unità di elaborazione descritta da un microprogramma con microistruzioni a struttura di frase

1) *PC è un automa di Mealy,*
2) *è necessario che PO sia realizzata come un automa di Moore.*

Il punto 1) è una conseguenza della struttura di frase delle microistruzioni. Il punto 2) è una condizione necessaria per una realizzazione corretta di PO.

Questo teorema ha quindi implicazioni sulla progettazione di PO. Nel seguito vedremo delle regole in base alle quali costruire sempre una PO corretta, senza bisogno di addentrarsi nella trattazione dei modelli matematici di automi.

Concettualmente, lo schema della rete sequenziale PO è sempre del tipo:



dove ℜ denota l'insieme di tutti i registri di PO (A, B, …), inclusi i registri d'ingresso. Come detto, tranne che per i registri di ingresso, l'impulso di clock è messo in AND con i segnali (β) di abilitazione alla scrittura. Le reti combinatorie ω_{PO} e σ_{PO} sono costruite a partire da componenti standard come ALU, commutatori, memorie, eventuali selezionatori, porte logiche.

7.2. Un esempio

Consideriamo una unità di elaborazione che, ricevendo in ingresso i valori di due interi A e B, restituisce in uscita i valori interi DIV = A/B (quoziente della divisione intera) e MOD = A%B (resto della divisione intera).

Adottando un semplice algoritmo a sottrazioni successive, il microprogramma (senza sincronizzazioni degli ingressi e delle uscite, che dovranno essere aggiunte dallo studente) può essere il seguente:

0. $A - B \rightarrow M$, $B \rightarrow N$, $0 \rightarrow Q$, 1

1. $(M_0 = 0)$ $M - N \rightarrow M$, $Q + 1 \rightarrow Q$, 1 ;

   $(M_0 = 1)$ $Q \rightarrow DIV$, $M + N \rightarrow MOD$, 0

Il registro Q è un temporaneo per il quoziente, M per il resto, e N per conservare il valore di B stabile durante l'esecuzione del microprogramma. Tutti i registri sono di una parola, supponiamo di 32 bit.

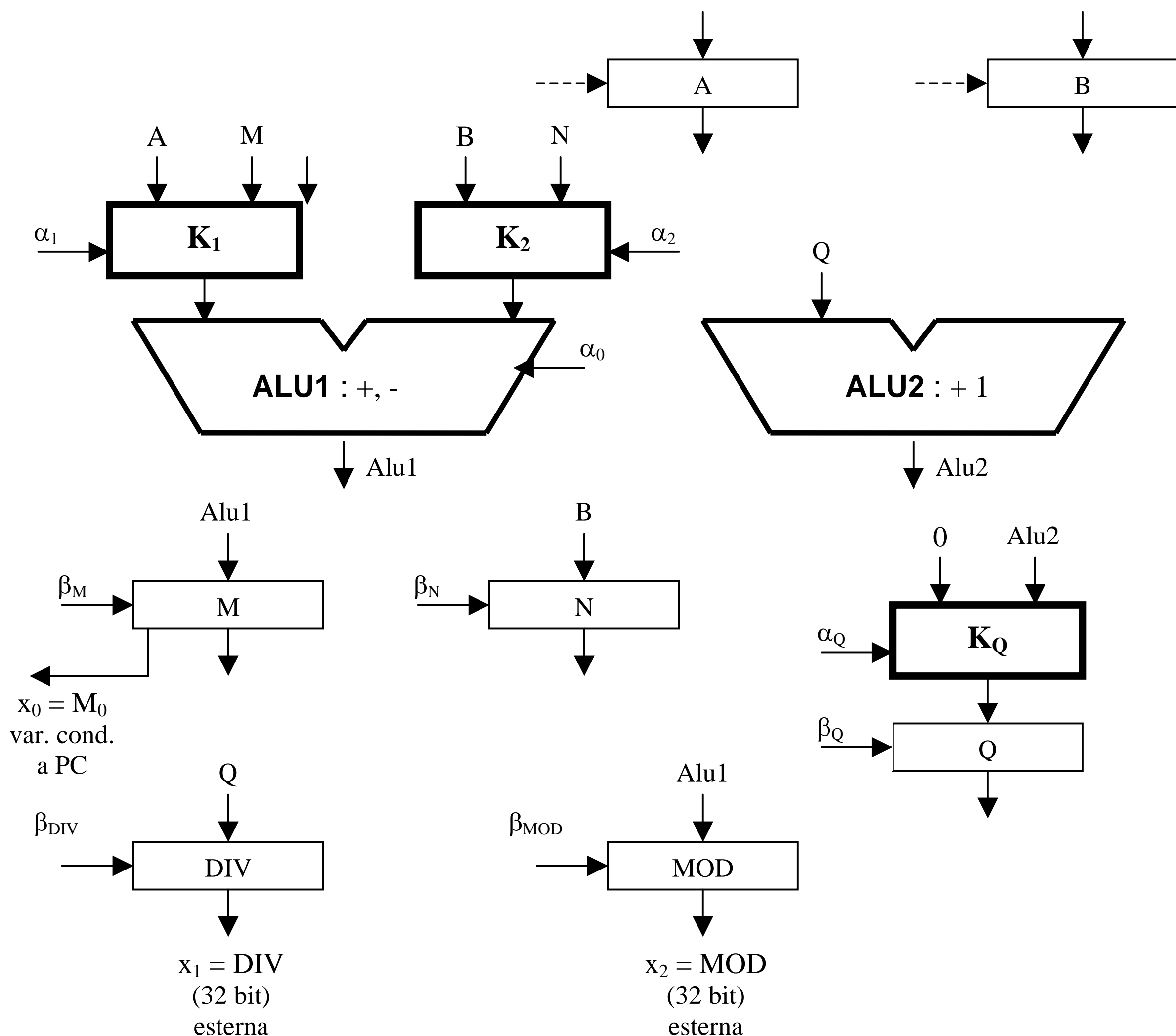
7.2.1. Rete sequenziale Parte Operativa

La struttura di PO si ottiene applicando un procedimento formale di sintesi avente complessità lineare nel numero dei registri e delle operazioni aritmetico-logiche e indipendente dal numero degli stati, a partire dalle microperazioni del microprogramma e utilizzando componenti standard.

Nel caso dell'esempio:

- A, B sono registri d'ingresso "impulsati sempre",
- occorrono due ALU per eseguire in parallelo M – N (ALU1) e Q + 1 (ALU2); ad ALU1 vengono fatte eseguire anche A – B e M + N,
- il registro M riceve valori solo da ALU1, N solo da B, DIV solo da Q, MOD solo da ALU1, e in Q possono essere scritti la costante 0 (su 32 bit) oppure il risultato di ALU2,

ottenendo:



Lo stato interno di PO è rappresentato dai contenuti di (A, B, M, N, Q, DIV, MOD): complessivamente si tratta di 224 variabili binarie dello stato interno, quindi PO ha 2^{224} stati interni possibili.

Lo stato d'ingresso di PO è rappresentato dai valori di (in_A, in_B, α_0, α_1, α_2, α_Q, β_Q, β_M, β_N, β_{DIV}, β_{MOD}): in tutto 73 variabili d'ingresso per un totale di 2^{73} stati d'ingresso.

Lo stato di uscita di PO è dato dai valori di (DIV, MOD, M_0): in tutto 65 variabili di uscita per un totale di 2^{65} stati di uscita. Come detto, poiché il modo di produrre le uscite esterne è standard, particolarmente significativa è la variabile di uscita data dalla variabile di condizionamento M_0.

Dall'analisi di PO si ricava che la funzione delle uscite ω_{PO} è *definita* come segue:

$x_0 = M_0$

x_1 = DIV (32 bit)

x_2 = MOD (32 bit)

Anche nel caso della variabile di condizionamento la definizione è data dalla *funzione identità rispetto al bit più significativo del registro M*. La corrispondente rete combinatoria si riduce quindi ad un semplice collegamento (rete "a zero livelli di logica"). È verificata la condizione necessaria per la correttezza della realizzazione di PO, è cioè che, ad ogni ciclo di clock, *lo stato di uscita è funzione solo dello stato interno presente*.

La funzione di transizione dello stato interno di PO, σ_{PO}, deve esprimere il modo di ottenere i valori degli ingressi di ogni registro (stato interno successivo) a partire dalle uscite di tutti i registri (stato interno presente) e dalle variabili di controllo (stato d'ingresso) ad ogni ciclo di clock. Ovviamente, poiché la PO è già stata realizzata con un procedimento a bassa complessità (nonostante il numero pazzesco di stati), si tratta di ricavare la *definizione* della funzione partendo dall'implementazione stessa (si tratta di un procedimento di *analisi*, in quanto la sintesi è stata appunto ottenuta direttamente a partire dal microprogramma).

Dallo schema di PO, la funzione σ_{PO} può essere analizzata con un procedimento di navigazione sul grafo di PO.

Ad esempio, la parte di stato interno rappresentata dal registro Q si ottiene posizionandosi sull'ingresso in_Q di Q e procedendo a ritroso sul grafo di PO percorrendo tutti i possibili cammini fino a raggiungere esclusivamente variabili di controllo, registri o costanti. Inizialmente si incontra il commutatore K_Q avente in ingresso la variabile di controllo α_Q, la costante zero e l'uscita di ALU2: il primo cammino si esaurisce con α_Q, il secondo con la costante zero, mentre il terzo prosegue con l'uscita di ALU2. ALU2 ha in ingresso solo il registro Q: anche questo terzo ramo del cammino si è dunque esaurito, e non esistono altre parti del grafo di PO che interessando la parte di stato interno rappresentata dal registro Q. In conclusione, *tutto il sottografo di PO costituito da ALU2 e KQ, con ingressi* α_Q, *zero e Q, ed uscita* in_Q, *rappresenta la rete combinatoria che implementa la parte della funzione* σ_{PO} *relativa al registro Q.*

In maniera sintetica, si può definire la funzione utilizzando un formalismo di programmazione come segue:

$in_Q = \sigma_{PO/Q}$ (Q, 0, α_Q, β_Q) = **when** $\beta_Q = 1$ **do**

if $\alpha_Q = 0$

then 0

else Q + 1

Il "corpo del **do**" è scritto utilizzando il metodo, visto a proposito della definizione delle reti combinatorie, per esprimere strutture costituite da commutatori ed ALU.

La scrittura "**when** $\beta_Q = 1$" sta a significare che il valore, risultante dalla funzione "corpo del **do**", rappresenta effettivamente lo stato interno successivo (cioè, rappresenterà lo stato interno presente al prossimo ciclo di clock) solo a condizione che venga abilitata la scrittura nel registro Q.

Per gli altri registri si ha:

in_A = valore primo ingresso esterno

in_B = valore secondo ingresso esterno

$in_N = \sigma_{PO/N}$ (B, β_N) = **when** $\beta_N = 1$ **do** B

$in_{DIV} = \sigma_{PO/DIV}$ (Q, β_{DIV}) = **when** $\beta_{DIV} = 1$ **do** Q

$in_M = \sigma_{PO/M}$ (A, B, M, N, α_0, α_1, α_2, β_M) = **when** $\beta_M = 1$ **do**

case α_0, α_1, α_2 **of**

0 0 0 : A + B

0 0 1 : A + N

0 1 0 : M + B

0 1 1 : M + N

1 0 0 : A – B

1 0 1 : A – N

1 1 0 : M – B

1 1 1 : M – N

$in_{MOD} = \sigma_{PO/MOD}$ (A, B, M, N, α_0, α_1, α_2, β_{MOD}) = **when** $\beta_{MOD} = 1$ **do**

case α_0, α_1, α_2 **of**

0 0 0 : A + B

0 0 1 : A + N

0 1 0 : M + B

0 1 1 : M + N

1 0 0 : A – B

1 0 1 : A – N

1 1 0 : M – B

1 1 1 : M – N

Si noti che, di fatto, alcuni dei risultati possibili della funzione di transizione dello stato interno relativamente ai registri M e MOD non si verificheranno mai durante l'esecuzione del microprogramma (ad esempio, A + N). Questo fatto è dovuto all'esigenza di mantenere di bassa complessità il procedimento di sintesi di PO.

Allo scopo di chiarire il più possibile il significato delle funzioni delle uscite e di transizione dello stato interno di PO, si consideri le due funzioni relativamente ad M_0. La funzione di transizione dello stato interno:

$in_M = \sigma_{PO/M0}$ (A, B, M, N, α_0, α_1, α_2, β_M) = **when** $\beta_M = 1$ **do**

case α_0, α_1, α_2 **of**

0 0 0 : $(A + B)_0$

0 0 1 : $(A + N)_0$

0 1 0 : $(M + B)_0$

0 1 1 : $(M + N)_0$

1 0 0 : $(A – B)_0$

1 0 1 : $(A – N)_0$

1 1 0 : $(M – B)_0$

1 1 1 : $(M – N)_0$

esprime il modo in cui viene ottenuto il valore dell'ingresso di M_0 entro la fine di un certo ciclo di clock.

Invece, la funzione delle uscite:

$x_0 = M_0$

esprime il valore dell'uscita di M_0 all'inizio di, e durante, un certo ciclo di clock. Il risultato della prima ($\sigma_{PO/M0}$) dipende necessariamente sia dallo stato interno presente (A, B, M, N) che dallo stato d'ingresso (combinazioni di α_0, α_1, α_2, β_M), mentre il risultato della seconda ($\omega_{PO/M0}$) dipende solo dallo stato interno preseente (M). Per avere un certo risultato della ω_{PO}, in un certo ciclo di clock, occorre che, in un ciclo di clock precedente, sia stata eseguita una opportuna istanza della σ_{PO}.

Come ogni rete sequenziale, PO è quindi completamente definita dalla quintupla:

- insieme degli stati d'ingesso,
- insieme degli stati di uscita,
- insieme degli stati interni,
- funzione delle uscite $\omega_{PO,}$
- funzione di transizione dello stato interno σ_{PO}.

In generale, una rete sequenziale è caratterizzata anche da un sesto elemento: lo stato interno iniziale. In questo esempio lo stato iniziale non è definito, in quanto non rilevante (l'operazione esterna è, concettualmente una funzione pura).

7.2.2. Rete sequenziale Parte Controllo

Lo stato interno di PC è in corrispondenza biunivoca con le etichette del microprogramma. Nell'esempio, PC ha due stati interni, corrispondenti alle microistruzioni di etichetta 0 e 1.

Per rappresentare gli stati interni è sufficiente una *variabile dello stato interno (in generale in numero di* $\lceil lg_2 n \rceil$*, con n numero delle microistruzioni)*, la cui codifica può essere banalmente:

stato interno = etichetta di microistruzione	variabile dello stato interno *y*
0	0
1	1

Il valore della variabile *y* rappresenta lo stato interno *presente*, memorizzato nel registro di stato del controllo RC; indicheremo con *Y* la variabile dello stato interno *successivo* (= in_{RC}). In questo esempio c'è un'unica variabile d'ingresso di PC: la variabile di condizionamento $x_0 = M_0$; quindi PC ha due stati d'ingresso. Le variabili di uscita di PC sono le nove variabili di controllo α_0, α_1, α_2, α_Q, β_Q, β_M, β_N, β_{DIV}, β_{MOD}; il numero degli stati di uscita di PC è dunque 512, anche se solo tre corrispondono a microistruzioni distinte.

Come spiegato in precedenza, la funzione di transizione dello stato interno di PC, σ_{PC}, si ricava dal microprogramma, considerando tutti i possibili valori di *Y* in corrispondenza delle combinazioni significative di *y* e M_0. Utilizzando una tabella di verità si ha:

y	M_0	Y
0	-	1
1	0	1
1	1	0

La funzione σ_{PC} è completamente definita da questa tabella. La sua implementazione si ottiene, come per ogni rete combinatoria, ricavando l'espressione logica:

$$Y = \overline{y} + y\overline{M_0}$$

e quindi lo schema della rete combinatoria σ_{PC} a due livelli di logica. Si noti che la stessa espressione si sarebbe potuto facilmente ricavare direttamente dal microprogramma, senza bisogno di costruire una tabella, considerando solo le etichette del microprogramma stesso:

0. …. , 1
1. (M_0 = 0) …. , 1 ;
   (M_0 = 1) …. , 0

Le due rappresentazioni sono un diverso zucchero sintattico dello stesso concetto.

Dal microprogramma si ricava anche la definizione della funzione delle uscite ω_{PC}. Usando una tabella di verità:

y	M_0	α_0	α_1	α_2	α_Q	β_Q	β_M	β_N	β_{DIV}	β_{MOD}
0	-	1	0	0	0	1	1	1	0	0
1	0	1	1	1	1	1	1	0	0	0
1	1	0	1	1	0	0	0	0	1	1

si ottengono le espressioni logiche:

$$\alpha_0 = \overline{y} + y\overline{M_0}$$
$$\alpha_1 = y$$
$$\alpha_2 = y$$
$$\alpha_Q = y\overline{M_0}$$
$$\beta_Q = \overline{y} + y\overline{M_0}$$
$$\beta_M = \overline{y} + y\overline{M_0}$$
$$\beta_N = \overline{y}$$
$$\beta_{DIV} = yM_0$$
$$\beta_{MOD} = yM_0$$

che permettono di realizzare la rete combinatoria ω_{PC} a due livelli di logica (solo un livello di logica per alcune variabili).

Si noti come, in generale, le variabili di uscita siano funzione sia delle variabili dello stato interno che delle variabili d'ingresso (caratteristica di PC descritta da un microprogramma a struttura di frase).

Si noti anche che alcune variabili hanno la stessa espressione logica: ciò permette di inserire una sola rete combinatoria per tutte quelle che sono identicamente uguali. Sarebbe stata una fatica inutile provare ad accorgersene durante la costruzione di PO, allo scopo di ridurre il numero delle variabili; ad esempio, accorgersi che $\alpha_1 = \alpha_2$, o che $\beta_Q = \alpha_0$, o che $\beta_N = \overline{\alpha_1}$, o che α_0 = Y, ecc. Semplicemente, tutte le relazioni tra variabili di controllo e/o dello stato interno sono ricavate *in modo automatico* nel procedimento di sintesi di PC.

7.2.3. Ciclo di clock

Durante ogni ciclo di clock PC e PO cooperano per permettere l'esecuzione della microistruzione la cui etichetta è rappresentata dallo stato interno di PC.

Il ciclo di clock dell'unità di elaborazione deve, quindi, avere *ampiezza sufficiente a permettere la stabilizzazione di tutte e quattro le funzioni delle due reti sequenziali*: ω_{PO}, σ_{PO}, ω_{PC}, σ_{PC}.

La chiave del discorso risiede nel fatto che le uscite di una rete sequenziale (variabili di condizionamento, variabili di controllo) sono gli ingressi dell'altra rete sequenziale. Il ragionamento da fare è il seguente:

- il fatto che, di tutte le quattro funzioni, *la funzione ω_{PO} sia l'unica il cui risultato non dipende dallo stato d'ingresso* permette di affermare con certezza che essa sarà *la prima a stabilizzarsi*, e la sua stabilizzazione inizierà all'inizio del ciclo di clock quando sarà disponibile il nuovo stato presente di PO;
- non appena ω_{PO} ha completato la stabilizzazione, inizia in parallelo la stabilizzazione finale di ω_{PC} e σ_{PC}: si ricordi infatti che il risultato di tali funzioni dipende anche dallo stato d'ingresso di PC (valore delle variabili di condizionamento prodotte da PO con la ω_{PO});
- non appena ω_{PC} ha completato la stabilizzazione, inizia la stabilizzazione finale di σ_{PO}, il cui risultato è ovviamente funzione anche del valore delle variabili di controllo prodotto appunto da ω_{PC};
- nel caso più generale (anche se raro) è possibile che una parte della stabilizzazione di σ_{PC} si sovrapponga a quella di σ_{PO}.

Da questo ragionamento deriva la formula per la valutazione del ciclo di clock riportata sulla Dispensa (Cap. III, sez. 3.4):

$$\tau = T_{\omega PO} + \max (T_{\omega PC} + T_{\sigma PO}, T_{\sigma PC}) + \delta$$

dove con la notazione T_f si indica il massimo ritardo di stabilizzazione di una generica funzione *f*, e con il simbolo δ l'ampiezza dell'impulso di clock.

Indicando con t_p il massimo ritardo di stabilizzazione di una porta logica, con T_K il ritardo di stabilizzazione di un commutatore, e supponendo $T_{ALU} = 5t_p$, nell'esempio si ha:

$T_{\omega PO} = 0$ (funzione identità sull'uscita di M_0)

$T_{\omega PC} = T_{\sigma PC} = 2t_p$ (vedere le espressioni logiche delle variabili di controllo e di *Y*)

$T_{\sigma PO} = T_K + T_{ALU} = 2t_p + 5t_p = 7t_p$

Il ritardo della σ_{PO} è quello della "più lenta" operazione di trasferimento tra registri, cioè di quella che utilizza il cammino di ritardo massimo nel grafo di PO per la stabilizzazione degli ingressi dei registri. Nel nostro caso questo ritardo si ha per "A - B → M", per "M - N → M", "Q + 1 → Q", e per "M + N → MOD".

Supponendo $\delta = t_p$, si ha quindi

$$\tau = 10t_p.$$

Dal ragionamento fatto per ricavare il ciclo di clock, ci si può rendere conto, pur informalmente, della dimostrazione del teorema "poiché PC è di Mealy, allora PO deve essere di Moore": infatti, se anche PO fosse di Mealy, nessuna delle quattro funzioni potrebbe iniziare a stabilizzarsi senza che anche le altre si siano già stabilizzate; ciò provocherebbe un circolo vizioso che porterebbe (tranne casi particolarissimi) ad una ri-inizializzazione indefinita della stabilizzazione delle quattro funzioni, con l'impossibilità di determinare un valore finito per τ.

7.2.4. Ancora sulla funzione delle uscite della Parte Operativa

Per approfondire l'aspetto della funzione delle uscite di PO, scriviamo il microprogramma dell'esempio in una forma alternativa:

0. A → M, B → N, 0 → Q, 1

1. (segno(M – N) = 0) M – N → M, Q + 1 → Q, 1;

   (segno(M – N) = 1) Q → DIV, M + N → MOD, 0

Il valore della variabile di condizionamento "segno(M – N)" va ricavato come uscita secondaria di una ALU ("flag" del segno del risultato dell'operazione eseguita).

Proviamo a mantenere la struttura di PO vista in precedenza, prelevando il valore di "segno(M – N)" dalla ALU1. Questa scelta è scorretta: notiamo che, ad ogni ciclo di clock, il valore di tale variabile dipende non solo dal contenuto di A, B, M, N, ma anche dai valori delle variabili di controllo α_0, α_1, α_2. Non sarebbe quindi soddisfatta la condizione necessaria per il corretto funzionamento di PO (PO sarebbe un automa di Mealy).

Occorre allora realizzare PO in un modo diverso: nel nostro caso, occorre inserire una terza ALU, capace di eseguire solo l'operazione di sottrazione, avente sugli ingressi solo M e N, e con uscita il "flag segno". In questo modo, la variabile di condizionamento "segno(M – N)" ad ogni ciclo di clock è ora funzione solo dello stato interno presente di PO (contenuti di M, N).

Questo porta ad avere un ritardo maggiore per la funzione ω_{PO}, nel nostro caso:

$$T_{\omega PO} = 5t_p$$

In questo esempio, non c'è convenienza sul tempo medio di elaborazione rispetto alla soluzione con variabile di condizionamento M_0: la maggiore ampiezza del ciclo di clock non è compensata da un guadagno nel numero di cicli di clock.

Esistono, però, molti altri casi in cui, oltre che permettere una scrittura "più naturale" del microprogramma, l'adozione di variabili di condizionamento più complesse può essere conveniente anche dal punto di vista del tempo medio di elaborazione. Si vedano, ad esempio, alcuni esercizi della sez. 9.

Come caso ulteriore, la seguente scrittura del microprogramma:

0. A → M, B → N, 0 → Q, 1
1. segno(M – N) → S, 2
2. (S = 0) M – N → M, Q + 1 → Q, 1;

   (S = 1) Q → DIV, M + N → MOD, 0

porta ad usare una variabile di condizionamento (S) che è direttamente l'uscita di un registro, e quindi la PO è certamente corretta anche prelevando il valore di "segno(M – N)" come uscita ausiliaria di ALU1.

Per contro, questa soluzione comporta un tempo medio di elaborazione maggiore di tutte quelle viste finora. Come esercizio si consiglia di provare una soluzione che, usando l'uscita del registro S come variabile di condizionamento, eviti di introdurre un ciclo di clock in più per ogni iterazione.

I casi esemplificati in queste note possono essere generalizzati per la soluzione degli esercizi relativamente alla scelta delle variabili di condizionamento:

a) volendo, è sempre possibile utilizzare *variabili di condizionamento che siano direttamente uscite di bit di registri*. In questo modo, la funzione ω_{PO} è la funzione identità, la condizione necessaria sulla correttezza di PO è sicuramente soddisfatta, e $T_{\omega PO} = 0$. È possibile che il numero di cicli di clock non venga minimizzato, anche se l'ampiezza del ciclo di clock può essere minimizzata;

b) spesso è possibile adottare *variabili di condizionamento più complesse*, per le quali la funzione ω_{PO} non è la funzione identità e quindi $T_{\omega PO} > 0$. In questi casi occorre anzitutto fare attenzione a realizzare la PO in modo da soddisfare la condizione necessaria sulla sua correttezza, cioè a far sì che il risultato della ω_{PO} dipenda solo dallo stato interno di PO ad ogni ciclo di clock: *PO per ogni microprogramam, esiste sempre almeno una realizzazione corretta di PO*. Inoltre, alla maggiore ampiezza del ciclo di clock corrisponde spesso un minor numero di cicli di clock, così che, specie in presenza di microprogrammi con loop, il tempo medio di elaborazione può essere minimizzato.

8. Comunicazioni a livello firmware

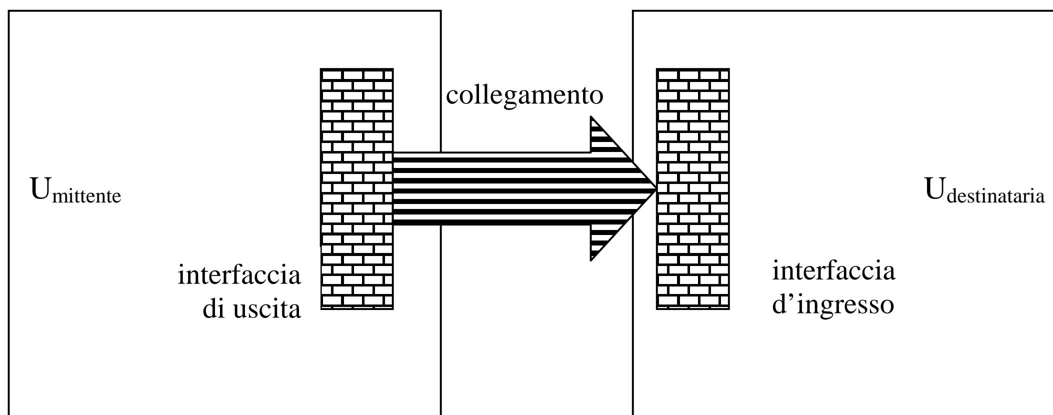
Alla parte di corso sulle comunicazioni tra unità di elaborazione è apportata una sensibile semplificazione rispetto a quanto contenuto nelle Dispense. Il materiale del Cap. I, sez. 3, ed il Cap. IV sono sostituiti dalle presenti note.

8.1. Comunicazione tra unità

In un sistema a livello firmware, la cooperazione tra unità di elaborazione avviene secondo il *modello a scambio di messaggi*. I *canali di comunicazione* sono implementati da supporti fisici (strutture di interconnessione) costituiti da interfacce presenti nella Parte Operativa delle unità e da collegamenti tra le unità stesse.

La situazione a cui occorre riferirsi è quella in cui le unità di elaborazione di un sistema operano tra loro in parallelo. Di conseguenza, occorre che le comunicazioni siano implementate mediante un preciso *protocollo* che tenga conto sia della *trasmissione* dei messaggi che della loro *sincronizzazione*. La sincronizzazione è necessaria per far sì che una unità U1, quando deve ricevere un messaggio da una unità U2, attenda che un nuovo messaggio gli sia stato esplicitamente inviato; oppure per far sì che, se una unità U2 intende inviare un messaggio ad U1, non rischi di sovrascrivere eventuali messaggi già inviati e non ancora ricevuti.

Il canale di comunicazione tra due unità viene implementato da tre componenti: *interfaccia di uscita*, *collegamento fisico* e *interfaccia d'ingresso*:



Sia le interfacce (dette, in gergo, anche "porte") che il collegamento fisico dispongono di strutture per la trasmissione e per la sincronizzazione. La trasmissione è dunque anche *unidirezionale.*

Per i collegamenti paralleli di *n* bit, caso che interessa particolarmente in questa corso ("porte parallele"), la trasmissione avviene *simultaneamente* per tutti gli *n* bit del messaggio.

Le *forme di comunicazione* possibili sono:

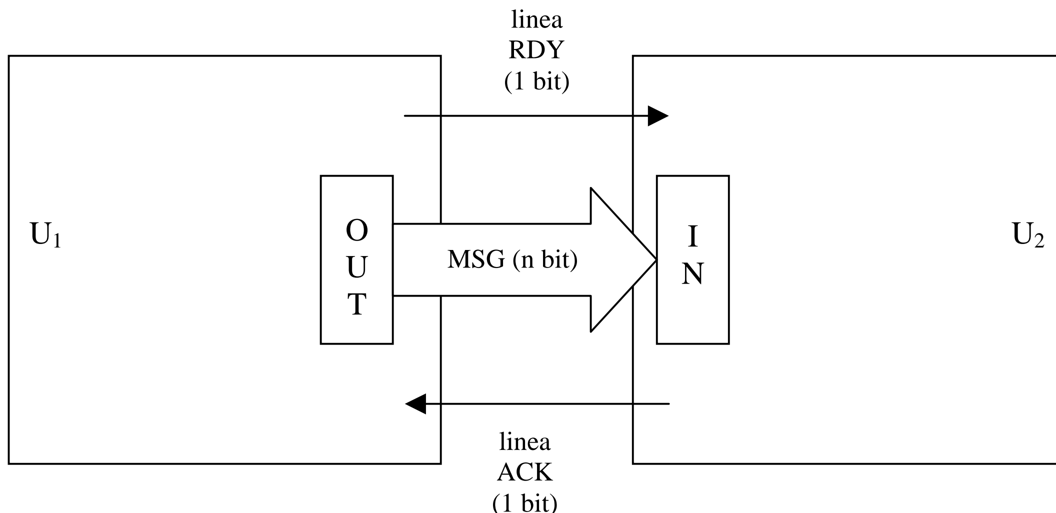
- *simmetrica vs asimmetrica*: nel caso *simmetrico* il canale di comunicazione ha un singolo mittente ed un singolo destinatario (comunicazione "uno-a-uno"), nel caso *asimmetrico in ingresso* il canale di comunicazione ha più mittenti ed un singolo destinatario (comunicazione "molti-a-uno"), nel caso *asimmetrico in uscita* il canale di comunicazione ha un singolo mittente e più destinatari (comunicazione "uno-a-molti"). In questo corso ci concentreremo sulla comunicazione **simmetrica**, mediante la quale possono essere emulate anche tutte le altre;
- *asincrona vs sincrona:* si dice che un canale (simmetrico) ha *grado di asincronia k*, con $k \geq 0$, se il mittente può inviare fino a *k* messaggi senza attendere che il destinatario ne abbia ricevuti uno o più. Nel caso che il mittente intenda inviare un (k+1)-esimo messaggio senza che il destinatario ne abbia ricevuto alcuno dei k precedenti, occorre che il mittente stesso attenda che il destinatario ne abbia ricevuto almeno uno. Nel caso che sia verificata la condizione di attesa del mittente, questo può proseguire nell'elaborazione. Il caso di $k = 0$ è quella della comunicazione *sincrona*: per ogni messaggio, il mittente, prima di poter proseguire, deve attendere che il destinatario lo abbia ricevuto.

Nel seguito realizzeremo interfacce capaci di implementare la comunicazione **asincrona con grado di asincronia uguale a uno**; questo stesso meccanismo si presta anche a implementare la comunicazione sincrona.

Siamo interessati a meccanismi di sincronizzazione che, quando possibile, assicurino la correttezza della comunicazione in modo **indipendente dal tempo**. Ciò significa che la semantica della comunicazione deve essere rispettata *indipendentemente dalla velocità relativa* delle unità mittenti e destinatarie (unità "partner"). A livello firmware l'indipendenza dei meccanismi dal tempo implica anche *non fare ipotesi sulla relazione tra i cicli di clock delle unità partner,* tanto in termini di lunghezza quanto di sfasamento relativo. Le soluzioni che qui verranno date risolvono implicitamente il problema anche in questi casi senza che ciò in generale comporti una perdita di efficienza. Esse sono inoltre più affidabili di soluzioni basate sulla rigida sincronizzazione delle unità, in quanto mascherano automaticamente variazioni che (a causa di guasti o "invecchiamento" dei componenti hardware) possono verificarsi nella relazione tra i cicli di clock.

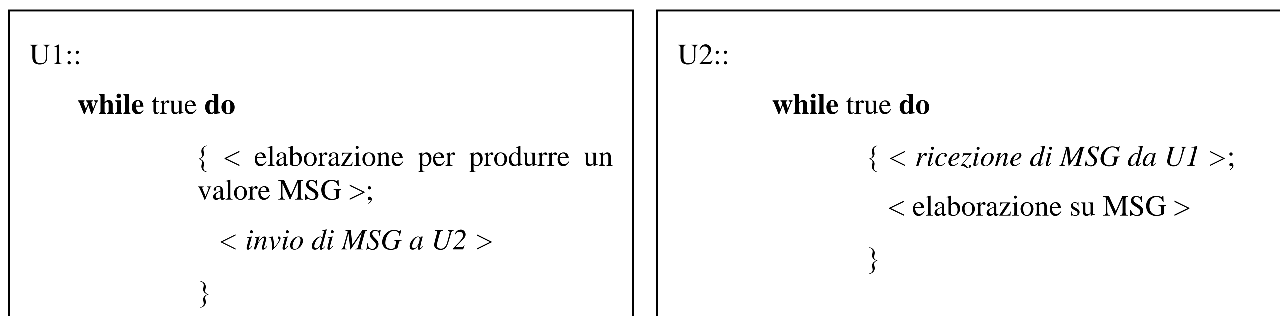
8.2. Interfaccia a transizione di livello

Consideriamo due unità, U1 (mittente) e U2 (destinataria), collegati da un canale di comunicazione **simmetrico e asincrono con grado di asincronia uguale a uno**. Un primo schema, ancora incompleto, delle interfacce e collegamenti è mostrato nella figura seguente.



8.3. Protocollo mediante segnali di RDY e ACK

Nel seguito, supporremo che il comportamento di U1 e U2 sia:



La *trasmissione* del messaggio (MSG) avviene attraverso un registro di uscita di U1 (OUT) ed un registro d'ingresso di U2 (IN) connessi da un collegamento fisico. Si tratta dei registri di ingresso e di uscita con le caratteristiche tipiche di una unità firmware. Una volta che U1 ha scritto il valore di MSG in OUT (abilitando la scrittura con il relativo β_{OUT}), questo valore si propaga, con il ritardo del collegamento, all'ingresso di IN di U2, e verrà quindi scritto in tale registro al primo impulso di clock di U2 (si ricordi che IN è "impulsato sempre"); il valore di MSG è ora disponibile a U2 come stato interno in microperazioni, uscita da PO come variabili di condizionamento, ecc.

Nel caso più generale, il messaggio è da una tupla di informazioni (ad esempio, un codice operativo OP, un valore su parola, un valore su *w* bit, …) , inviate contemporaneamente attraverso più registri di uscita e ricevute in altrettanti registri di ingresso, con lo stesso meccanismo di sincronizzazione.

La *sincronizzazione* è implementata facendo uso di due linee addizionali, di un bit ciascuna, dette *linea di Ready* (RDY) e *linea di Acknowledgement* (ACK):

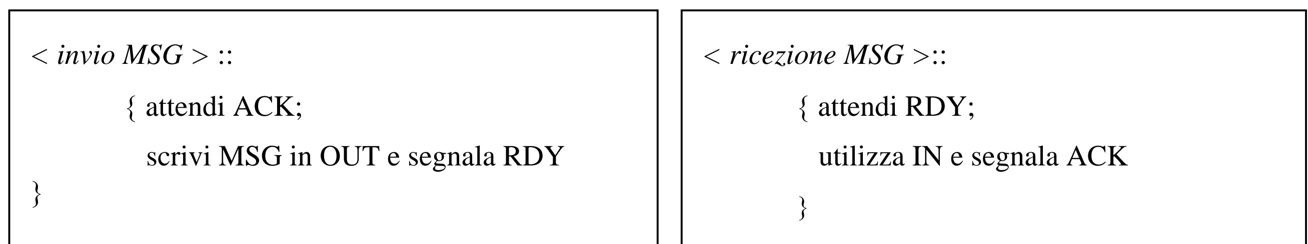
- mediante la linea RDY U1 fa sapere ad U2 che ha inviato un nuovo messaggio; essa ha dunque il significato di segnalare a U2 la *presenza* di un nuovo messaggio;
- mediante la linea ACK U2 fa sapere ad U1 che ha ricevuto un nuovo messaggio; essa ha dunque il significato di segnalare a U1 la ricezione di un messaggio.

Poiché intendiamo implementare un protocollo per la comunicazione asincrona con grado di asincronia $k = 1$, il significato della linea ACK è quello di far sapere a U1 che è possibile inviare un nuovo messaggio (nel caso U1 lo desideri).

Il protocollo di comunicazione è definito come le azioni, effettuate da U1 per eseguire < *invio MSG* > e da U2 per eseguire < *ricezione MSG* >, in modo da assicurare la *corretta sincronizzazione indipendente dal tempo*. "Corretta" significa che

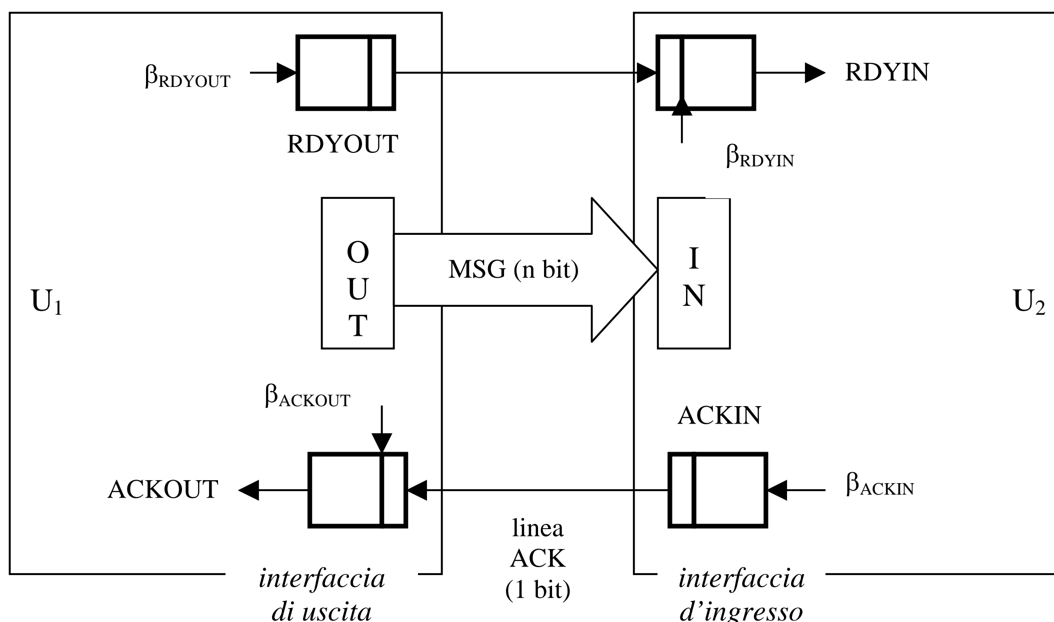
- nessun messaggio deve essere perso,
- se U1 invia un messaggio, prima o poi U2 è in grado di riceverlo,
- se U2 riceve un messaggio, prima o poi U1 è in grado di inviarne un altro.

In linea di massima, il protocollo contiene le seguenti azioni:



8.4. Implementazione delle interfacce

Si tratta ora di dare una implementazione di questa specifica di funzionamento. A questo scopo, le due linee di RDY e di ACK sono interfacciate da quattro elementi di memoria, appositamente definiti (non sono normali registri, come vedremo nel seguito), mostrati nella figura seguente.



I quattro elementi di memoria, RDYOUT, ACKOUT, RDYIN e ACKIN (contrassegnati dai simboli speciali indicati in figura), sono detti *indicatori di interfaccia* e sono così caratterizzati:

- sugli *indicatori di interfaccia di uscita* (RDYOUT e ACKIN) è definita l'unica operazione chiamata *set:* l'esecuzione di *set RDYOUT* da parte di U1 ha come effetto finale il fatto che l'indicatore di interfaccia d'ingresso di U2 RDYIN (quello a cui è collegato RDYOUT) assume l'uscita uguale a uno; analogamente, l'esecuzione di *set ACKIN* da parte di U2 ha come effetto finale il fatto che l'uscita di ACKOUT in U2 assume il valore uno;
- sugli *indicatori di interfaccia d'ingresso* (RDYIN, ACKOUT) sono definite due operazioni. Una è il test (*if*) del valore di uscita di tali indicatori. L'altra, chiamata *reset*, ha come effetto di portare l'uscita degli indicatori stessi a zero: *reset RDYIN* porta a zero l'uscita di RDYIN, *reset ACKOUT* porta a zero l'uscita di ACKOUT.

Con queste operazioni, il protocollo si esprime come segue:

< *invio MSG* > ::	< *ricezione MSG* >::
{ **if not** ACKOUT **then** *wait*;	{ **if not** RDYIN **then** *wait*;
MSG → OUT, *set* RDY, *reset* ACKOUT;	f(IN, …) … , *set* ACKIN, *reset* RDYIN;
}	}

La scrittura *wait* ha il semplice significato di ciclare sulla stessa microistruzione, senza eseguire operazioni (*nop*), in attesa che il predicato atteso si verifichi. La scrittura *f(IN, …)* vuole esprimere l'utilizzo immediato (nello stesso ciclo di clock) del valore nel registro IN (una operazione di trasferimento tra registri, oppure un test).

Per inviare un messaggio, U1 per prima cosa attende l'ACK testando ACKOUT finché non lo trova ad uno. A questo punto, scrive il messaggio nel registro OUT e contemporaneamente segnala la presenza a U2 (*set RDYOUT*) e rimette a zero ACKOUT (*reset ACKOUT*). È necessario rimettere ACKOUT a zero in quanto, altrimenti, al prossimo tentativo di inviare un messaggio non avremmo informazione certa sul fatto che il precedente è stato ricevuto.

Per ricevere un messaggio, U2 per prima cosa attende il RDY testando RDYIN finché non lo trova ad uno. A questo punto, utilizza subito il valore nel registro IN e contemporaneamente segnala la ricezione a U1 (*set ACKIN*) e rimette a zero RDYIN (*reset RDYIN*). È necessario rimettere RDYIN a zero in quanto, altrimenti, al prossimo tentativo di ricevere un messaggio non avremmo informazione certa sul fatto che il messaggio stesso è stato effettivamente inviato.

Si noti che, al ciclo di clock successivo alla ricezione con successo di un messaggio, i contenuti dei registri d'ingresso non sono più significativi (è stato eseguito *set ACK*, per cui, da questo momento in poi, il mittente potrà inviare un nuovo messaggio in un momento impredicibile); se tali valori dovessero servire successivamente, occorre quindi salvarli in registri interni.

La correttezza di funzionamento è assicurata dal verificarsi di tre condizioni:

1) l'uscita di ogni indicatore di interfaccia d'ingresso assume la sequenza di valori 0, 1, 0, 1, 0, ….;

2) quanto viene eseguito il set su un indicatore di uscita (come RDYOUT) l'indicatore di ingresso collegato (come RDYIN) ha già l'uscita a zero (è stato certamente eseguito il *reset*);

3) in condizioni iniziali ogni unità esegue *set* sui suoi indicatori di interfaccia di uscita di tipo ACKIN, in modo che i possibili mittenti trovino ACK = 1 per effettuare l'invio del primo messaggio (questa operazione non verrà indicata nel microprogramma, in quanto eseguita automaticamente all'accensione del sistema).

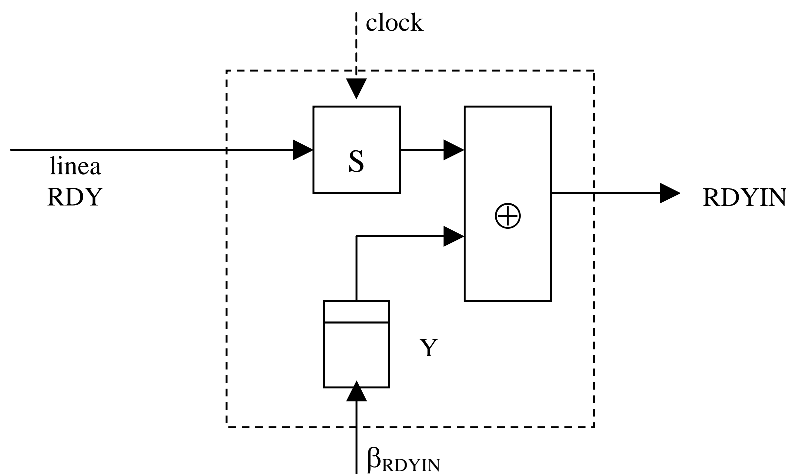
L'implementazione degli indicatori di interfaccia è la seguente:

- gli indicatori di interfaccia di uscita sono *contatori modulo 2*, aventi un solo ingresso (β_{RDYOUT}, β_{ACKIN}). La loro uscita conta modulo 2 il numero di volte che la variabile di controllo β assume il valore uno. In tal modo, è assicurata la condizione 1) di cui sopra. Quindi, per eseguire *set RDYOUT* nella PO, la PC di U1 produce $\beta_{RDYOUT} = 1$. Analogamente per ACKIN.

  Per segnalare la presenza di un messaggio (la ricezione di un messaggio), U1 (U2) provoca una transizione di livello della linea RDY (ACK): da qui il nome di *interfaccia a transizione di livello*.

  Si può dimostrare che facendo corrispondere valori di livelli, invece che transizioni di livello, agli eventi che si vogliono esprimere (ad esempio, sempre il valore RDY = 1 per segnalare presenza di messaggio) non sarebbe possibile implementare la comunicazione asincrona con k > 0;

- gli indicatori di interfaccia di ingresso sono *reti sequenziali che riconoscono ogni transizione di livello*. La loro implementazione è la seguente (è mostrato il caso di RDYIN; analoga è la struttura di ACKOUT):



S è un normale registro di ingresso di un bit; Y è un contatore modulo 2; S e Y sono ingressi di un confrontatore la cui uscita è proprio l'uscita dell'indicatore di interfaccia RDYIN.

È necessario che il valore iniziale di Y in U2 sia uguale al valore iniziale di RDYOUT in U1, e quindi uguale a S. In tale situazione, all'inizio l'uscita del confrontatore è uguale a zero. Quando RDYOUT subisce la prima transizione di livello (primo messaggio inviato), S cambia di valore e l'uscita del confrontatore passa a uno segnalando la transizione di livello e quindi la presenza del messaggio. A questo punto (condizione 2) il protocollo prescrive che sia eseguito *reset RDYIN* mediante $\beta_{RDYIN} = 1$; questo ha come effetto che l'uscita di Y cambia diventando di nuovo uguale a S e quindi l'uscita del confrontatore è di nuovo nelle condizioni iniziali. Da questo momento in poi, il ragionamento si applica a qualunque invio di messaggio:

- prima dell'invio di un messaggio si ha RDYOUT = S = Y, e quindi RDYIN = 0,
- una variazione di livello di RDYOUT, per segnalare l'invio del messaggio stesso, verrà rilevata in U2 in quanto RDYIN passerà a uno.

8.5. Utilizzo del protocollo di comunicazione

In quanto "protocollo" il modo di funzionare ora descritto deve essere usato come uno *standard*, senza apportare modifiche di sorta. Nei microprogrammi, il protocollo di invio e di ricezione potrà (dovrà) essere *parallelizzato con l'elaborazione interna*.

Nella realizzazione di PC e PO occorre inserire gli indicatori di interfaccia e le relative variabili di controllo.

8.6. Esempio 1

Si consideri una unità U che riceve dall'unità U_a messaggi costituiti dalla tripla (OP, A, B) dove OP è un codice operativo di un bit ed A e B sono interi a 32 bit. L'unità U invia all'unità U_b il valore OUT uguale al risultato dell'espressione (A + B – 5)/2 se OP = 0, oppure dell'espressione (A/2 + 2*B) se OP =1.

L'interfaccia d'ingresso è costituita dai tre registri d'ingresso OP, A, B (*considerati logicamente come uno solo agli effetti del protocollo di comunicazione*) e da due indicatori di interfaccia d'ingresso RDYIN, ACKIN. L'interfaccia di uscita è costituita dal registro di uscita OUT e dagli indicatori di interfaccia di uscita RDYOUT e ACKOUT.

Il microprogramma è il seguente:

0. (RDYIN, OP = 0 -) nop, 0;

   (= 1 0) A + B $\rightarrow$ M, reset RDYIN, set ACKIN, 1;

   (= 1 1) sh_{d1} (A) $\rightarrow$ M, sh_{s1} (B) $\rightarrow$ N, reset RDYIN, set ACKIN, 3

1. M – 5 $\rightarrow$ M, 2

2. (ACKOUT = 0) nop, 2;

   (= 1) sh_{d1} (M) $\rightarrow$ OUT, set RDYOUT, reset ACKOUT, 0

3. (ACKOUT = 0) nop, 3;

   (= 1) M + N $\rightarrow$ OUT, set RDYOUT, reset ACKOUT, 0

È molto importante notare che *l'adozione del protocollo di comunicazione non comporta alcuna penalità dal punto di vista del tempo di elaborazione,* in termini di numero di cicli di clock, grazie alla sua parallelizzazione con l'elaborazione interna. Quindi, la modifica di microprogrammi già scritti senza sincronizzazione è semplice e non comporta modifiche alla valutazione delle prestazioni.

Ovviamente, nel calcolo delle prestazioni *non* si devono valutare gli eventuali cicli di clock spesi in attesa di eventi di sincronizzazione: il tempo medio di elaborazione di una unità è quello impiegato dall'unità stessa ad eseguire le operazioni esterne; "nessuna responsabilità" ha l'unità se chi deve inviargli i dati, o chi deve ricevere i suoi risultati, ritarda. La valutazione che noi diamo è l'unica di interesse, cioè nelle condizioni di *massima sollecitazione* da parte delle altre unità (in tali condizioni, non vengono mai eseguite le *nop*).

Per quanto riguarda l'impatto del protocollo sulla struttura di PC e PO:

- gli indicatori di interfaccia fanno parte dello stato interno della PO;
- RDYIN e ACKOUT fanno parte delle variabili di condizionamento;
- consideriamo la configurazione di variabili di controllo per l'esecuzione di alcune microoperazioni:
  - A + B $\rightarrow$ M, reset RDYIN, set ACKIN : variabili α (da stabilire), $\beta_M = 1$, $\beta_{RDYIN} = 1$, $\beta_{ACKIN} = 1$;
  - sh_{d1} (M) $\rightarrow$ OUT, set RDYOUT, reset ACKOUT : variabili α (da stabilire), $\beta_{OUT} = 1$, $\beta_{RDYOUT} = 1$, $\beta_{ACKOUT} = 1$;
- consideriamo la funzione delle uscite di PC relativamente alla sola variabile di controllo β_{RDYOUT}:
  - $\beta_{RDYOUT} = y_0 \overline{y_1} ACKOUT + y_0 y_1 ACKOUT = y_0 ACKOUT$

8.7. Esempio 2

Consideriamo il microprogramma già scritto, senza sincronizzazione, nelle note "Reti sequenziali e strutturazione firmware" per l'unità che calcola il quoziente e il resto della divisione intera:

0. A – B $\rightarrow$ M, B $\rightarrow$ N, 0 $\rightarrow$ Q, 1

1. (M_0 = 0) M – N $\rightarrow$ M, Q + 1 $\rightarrow$ Q, 1 ;

   (M_0 = 1) Q $\rightarrow$ DIV, M + N $\rightarrow$ MOD, 0

Aggiungendo il protocollo di comunicazione, e chiamando gli indicatori di interfaccia con gli stessi simboli dell'Esempio 1, il microprogramma diviene:

0. (RDYIN = 0) nop, 0;

   (= 1) A – B → M, B → N, 0 → Q, reset RDYIN, set ACKIN, 1

1. (M_0, ACKOUT = 0 -) M – N → M, Q + 1 → Q, 1 ;

   (M_0, ACKOUT = 1 0) nop, 1;

   (M_0, ACKOUT = 1, 1) Q → DIV, M + N → MOD, set RDYOUT, reset ACKOUT, 0

Per esercizio studiare PC e PO di questa unità.

8.8. Comunicazione sincrona

Il protocollo che è stato studiato per l'interfaccia a transizione di livello vale per la comunicazione asincrona con grado di asincronia *k = 1*. Se vogliamo usare la stessa interfaccia a transizione di livello per la comunicazione sincrona (*k = 0*), il protocollo rimane invariato per la parte < *ricezione MSG* >, mentre le azioni della parte < *invio messaggio* > devono semplicemente essere invertite (testare ACK *dopo* aver inviato il messaggio).

Di fatto, l'uso della comunicazione sincrona a livello firmware non è quasi mai giustificato.

8.9. Comunicazione asincrona con grado di asincronia k > 1

In questo caso non esiste una soluzione basata su semplici interfacce nelle due unità comunicanti, ma è necessario interporre tra le due unità una terza unità (*Unità Buffer*) che implementi una coda FIFO a *k* posizioni.

La comunicazione tra l'unità mittente U1 e l'unità destinataria U2 si realizza quindi con la coppia di comunicazioni tra U1 e l'Unità Buffer e tra l'Unità Buffer e U2.

8.10. Comunicazione a domanda e risposta

Si consideri una unità U1 che invia un messaggio a U2 dopo di che attende un messaggio da U2 stessa; tipicamente, si tratta di una situazione cliente-servente, in cui il cliente U1 prima invia un messaggio al servente U2 per chiedere un servizio, e quindi attende dal servente un messaggio di risposta. Anche se le interfacce tra U1 e U2 contengono gli indicatori di ACK, questi non servono nella comunicazione a domanda e risposta (serviranno comunicazioni come quelle viste finora): U1 non invierà un nuovo messaggio prima di avere ricevuto la risposta da U2, quindi l'ACK è implicito nella risposta.

Ad esempio, si consideri l'interfaccia tra processore e memoria principale (in realtà, con altre unità interposte tra processore e memoria). In uscita al processore l'interfaccia contiene i registri:

- IND: indirizzo logico
- DATAOUT: dato da scrivere
- OP: operazione richiesta alla memoria (ad esempio, 0 = lettura, 1 = scrittura)
- RDYOUT: indicatore di interfaccia di uscita

L'interfaccia in ingresso al processore contiene:

- DATAIN: parola letta dalla memoria
- ESITO: esito dell'accesso in memoria (successo, fallimento 1, fallimento 2, ….)
- RDYIN: indicatore di interfaccia di ingresso.

Il microprogramma del processore per effettuare un accesso in memoria a domanda e risposta per lettura è del tipo:

i. … indirizzo → IND, 0 → OP, set RDYOUT, i+1

i+1. (RDYIN = 0) nop, i+1;

(= 1) reset RDYIN, f (DATAIN, …) … , ESITO → ESITO 1, … , …

Per un accesso a domanda e risposta in scrittura:

j. … indirizzo → IND, dato → DATAOUT, 1 → OP, set RDYOUT, j+1

j+1. (RDYIN = 0) nop, j+1;

(= 1) reset RDYIN, ESITO → ESITO 1, … , …

Ovviamente, “indirizzo” e “dato” sono risultati di operazioni dipendenti dall’istruzione interpretata.

9. Esercizi sulla strutturazione firmware

Alla fine della trattazione di questa parte del corso, lo studente deve risolvere qualunque esercizio *esplicitando il massimo parallelismo* nelle microistruzioni e *sincronizzando tutte le comunicazioni* in ingresso ed in uscita.

In tutti i casi in cui sia richiesto di progettare una unità di elaborazione si intende, nel caso più completo:

- scrivere il microprogramma, *spiegando le scelte più significative*; alcuni aspetti importanti (anche se non gli unici) relativamente alle spiegazioni potranno riguardare: *i)* l’algoritmo adottato per l’interprete (quando non sia banale), *ii)* la scelta delle variabili di condizionamento in relazione al problema di minimizzare il numero di cicli di clock oppure la lunghezza del ciclo di clock, oppure entrambi, *iii)* la scelta di registri e reti logiche affinché la Parte Operativa risulti effettivamente una rete sequenziale di Moore, *iv)* la scelta delle funzioni, e relative reti combinatorie, adottate per l’esecuzione delle microoperazioni o per ricavare variabili di condizionamento, nei casi in cui tali aspetti non siano banali;
- mostrare, e spiegare come è stata ottenuto, lo schema della Parte Operativa; ricavare la funzione delle uscite e la funzione di transizione dello stato interno, quest’ultima applicata ad un sottoinsieme piccolo, ma significativo, dei registri;
- ricavare, e spiegare, la *definizione* delle funzioni delle uscite e di transizione dello stato interno della Parte Controllo; ricavare *le espressioni logiche* di tali funzioni, limitandosi, per quella delle uscite, ad un sottoinsieme piccolo, ma significativo, delle variabili di controllo;
- ricavare il valore del ciclo di clock, spiegando come sono stati ottenuti i valori di tutti i ritardi utilizzati, nell’ipotesi che sia noto il ritardo t_p di porte logiche con al massimo N ingressi (ad esempio, N = 8), che la durata dell’impulso di clock sia uguale a t_p, e che siano noti, come multipli di t_p, il ritardo di una generica ALU ed il tempo di accesso di eventuali memorie di registri date;
- ricavare il tempo medio di elaborazione di ognuna delle operazioni esterne ed il tempo medio globale, assumendo equiprobabili le varie operazioni esterne, a meno che il testo non dica diversamente.

1) Progettare una unità di elaborazione U che

- riceve in ingresso messaggi (OP, DEST A, B), con OP e DEST di 1 bit, e A e B interi di 32 bit,
- se OP = 0 calcola (A + B – 5) / 2, altrimenti (A + B) % 1024,
- se DEST = 0 invia il risultato ad una certa unità U_a, altrimenti lo invia ad una unità U_b diversa da U_a.

2) Si consideri la stessa funzionalità svolta dall’unità dell’Esercizio 1. Dire se, oltre che come unità di elaborazione (cioè, come l’insieme di due reti sequenziali opportune, PC e PO), tale funzionalità può essere

concettualmente realizzata anche come una singola rete combinatoria. Nel caso sia vero, provare a darne una effettiva realizzazione come rete combinatoria. Si veda anche l'esercizio 6 del Cap III.

3) Spiegare se, in generale, il numero delle variabili di condizionamento, il numero delle variabili dello stato interno della Parte Controllo e il numero delle variabili di controllo hanno impatto sul ritardo di stabilizzazione della funzione delle uscite e della funzione di transizione dello stato interno della Parte Controllo di un'unità.

4) Esercizi 2 e 3 del Cap. III.

5) Progettare una unità di elaborazione così definita:

- ha tre ingressi: A di 32 bit, J di 5 bit, OP di 1 bit, e due uscite: COUNT di 6 bit e B di 32 bit;
- interpreta due operazioni esterne:
  - per OP = 0: se il bit J-esimo di A assume il valore 1, allora il valore di A viene trasferito in B;
  - per OP =1: il numero di "1" presenti in A viene trasferito in COUNT.

In questo esercizio non è permesso di far uso della funzione di shift.

6) Esercizio 5 del Cap. III.

7) Esercizio 7 del Cap. III.

8) Progettare una unità di elaborazione U è così definita:

- possiede al suo interno una memoria A di 256K locazioni di 32 bit;
- riceve in ingresso messaggi (J1, J2), dove J1 e J2 sono entrambi di 5 bit;
- interpreta una operazione esterna, consistente nell'inviare sull'uscita OUT il numero di locazioni della memoria A aventi il J1-esimo ed il J2-esimo bit uguali.

9) Esercizio 10 del Cap. III.

10) Una unità di elaborazione possiede al suo interno una memoria A di 1M locazioni di 32 bit, riceve messaggi IN di 32 bit e invia messaggi OUT di 21 bit. L'unità è descritta del seguente microprogramma, dove I e C sono registri di 21 bit:

0. $IN \rightarrow B,\ 0 \rightarrow I,\ 0 \rightarrow C,\ 1$

1. $(I_{20},\ segno(A[I] - B),\ B_5 = 0\ 0\ 0 -,\ 0\ 1\ 1\ )\ C + 1 \rightarrow C,\ I + 1 \rightarrow I,\ 1\ ;$

$(= 0\ 0\ 1\ ,\ 0\ 1\ 0\ )\ I + 1 \rightarrow I,\ 1\ ;$

$(= 1 - -\ )\ C \rightarrow OUT,\ 0$

Ricavare la PO, dimostrare che risponde al modello matematico di Moore, e valutare il ciclo di clock di U.