ALU a 4-bit

by Antonio Bernardini

Table of contents

1 Homework

1.1 Introduzione

Per questo homework è stato richiesto di progettare una semplice ALU che sia in grado di lavorare con operandi a 4 bit. La ALU è formata da una *rete iterativa* e non è necessario implementare ottimizzazioni basate sul parallelismo.

1.2 Operazioni supportate

La ALU lavora su uno o due operandi ($a = \langle a_3 a_2 a_1 a_0 \rangle$ e $b = \langle b_3 b_2 b_1 b_0 \rangle$), a seconda dell'operazione che viene richiesta, utilizzando un *opcode*. L'*opcode* è a 4 bit, decomposto nei bit $\mathrm{op}_0, \mathrm{op}_1, \mathrm{op}_2, \mathrm{op}_3$. Le operazioni da supportare sono riportate nella seguente tabella:

op_0	op_1	op_2	op_3	Operazione
0	0	0	0	$a + b$ (somma senza segno)
0	0	0	1	Shift a sinistra di una posizione di a
0	0	1	0	a AND b
0	1	0	0	a OR b
0	1	1	0	$a \oplus b$
1	0	0	0	$a \odot b$
1	0	1	0	$a \mid b$
1	1	0	0	$a \downarrow b$
1	1	1	0	$\bar{a}$

Table 1. Tabella delle operazioni supportate dall'ALU a 4 bit

Le operazioni logiche sono operazioni bit a bit. Nel caso dell'operazione shift, il valore in input di b è una *don't care condition*. Nel caso di $\bar{a}$, b può essere assunto forzato a zero. Il circuito restituisce in c_{out} il carry risultante dalla somma tra a e b ed il bit più significativo estratto dall'operazione di shift a sinistra.

Warning. Non saranno considerate valide soluzioni che utilizzano sommatori o shifter per supportare queste operazioni. In generale, saranno valutate zero punti tutte le soluzioni che non sono iterative.

Tip. Si consiglia di affrontare il problema per fasi. Ogni operazione richiesta può essere risolta con un sotto-circuito dedicato. Riconoscere delle ricorrenze nei circuiti può aiutare nella minimizzazione. L'utilizzo di ROM o PLA può aiutare *molto* nell'individuazione di una soluzione minima.

2 Soluzione

2.1 Introduzione

Poichè l'ALU è una *rete iterativa* occorre ragionare sulla modellazione di una singola ALU per 1 bit per poi copiare e incollare ogni singola "mini" ALU per ottenerne infine una a 4 bit, 5 bit, eccetera. Come risulterà più chiaro in seguito, è normale che il primo modulo ALU ad 1 bit sia leggermente meno complesso dei successivi e questo va bene finchè si mantiene la logica di una *rete iterativa*.

2.2 Scelte progettuali

Il modo più semplice e veloce per progettare un'ALU a 4 bit è utilizzare una struttura simile a quella mostrata in Fig. 1.

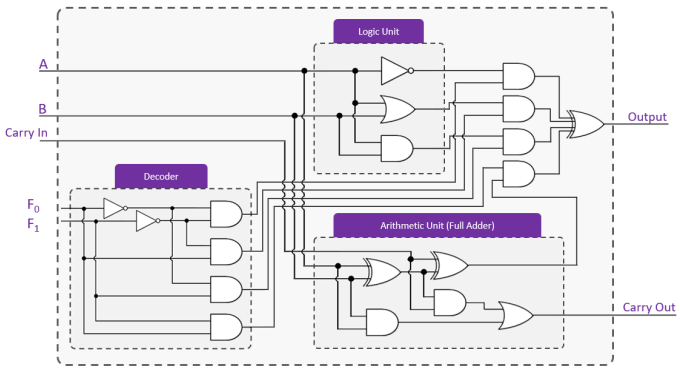

Figure 1. ALU ad 1 bit

Tuttavia occorre osservare che se per una singola ALU ad 1 bit abbiamo bisogno di 19 porte logiche, per la nostra ALU a 4 bit si arriverebbe a $19 \cdot 4 = 76$ porte logiche. Questo risulta problematico per due motivi: l'homework assegnato ci permette di realizzare un'ALU a 4 bit utilizzando un massimo di 25 componenti, quindi con 76 porte logiche circa (perchè il primo modulo ne ha meno di 19) viene superato il range, e inoltre occorre tenere presente l'elevato numero di transistor che verrebbero impiegati e di conseguenza anche eventuali ritardi nel caso in cui si volesse prototipare tale ALU su breadboard o PCB.

Pertanto la soluzione più ovvia è implementare una ROM (Read Only Memory), grazie alla quale sarà possibile scrivere in degli appositi indirizzi di memoria dei valori (ossia i risultati delle operazioni). Infatti la ROM funziona come un puntatore in linguaggio C: il puntatore "punta" ad un indirizzo di memoria e se voglio conoscerne il contenuto lo dereferenzio.

```c
#include <stdio.h>

int main(void) {
    int value = 69;
    int *p = &value;
    printf("Address: %#lX\n", (unsigned long)p);
    printf("Value: %d\n", *p);
    return 0;
}
```

Infine, per la realizzazione dell'ALU a 4 bit occorre ragionare su quali valori conviene usare come input e quali conviene usare come output, per non dare nulla per scontato. Pertanto ogni bit che forma i numeri $a = \langle a_3 a_2 a_1 a_0 \rangle$ e $b = \langle b_3 b_2 b_1 b_0 \rangle$ sarà un input per l'i-esima ALU, con $i \in [0, 3]$, così come i 4 bit di *opcode* saranno in input, per capire quale operazione eseguire. Quest'ultima selezione dell'operazione poteva essere svolta da un Multiplexer, tuttavia quest'ultimo è formato da un insieme di porte logiche e questo comporta un maggiore uso di componenti. Inoltre non bisogna dimenticarsi che la propagazione del carry c_i, con $i \in [0, 3]$, è sia un'operazione di input che di output. Infine come ulteriore output avremmo il risultato $r = \langle r_3 r_2 r_1 r_0 \rangle$.

2.3 Operazioni per il modulo ALU ad 1 bit

2.3.1 Somma senza segno

La somma senza segno $a_0 + b_0$ è l'unica operazione, tra quelle richieste, che si può svolgere utilizzando proprio la *rete iterativa* di un Full Adder.

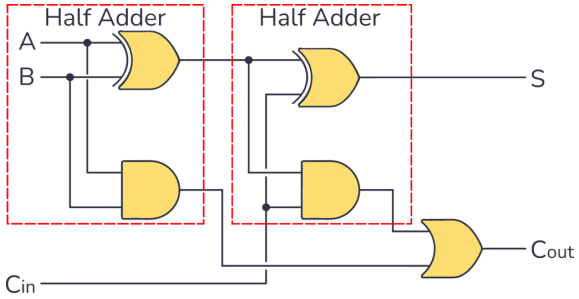

Figure 2. Full Adder

Tuttavia, come accennato pocanzi, andremo ad implementare una *rete iterativa* basata su una ROM (Read Only Memory). Pertanto in questa fase ci serve capire come cambierà la somma senza segno tra il primo modulo ALU ad 1 bit e i successivi. Risulta quindi abbastanza evidente che per il primo modulo ALU abbiamo un carry $c_{\text{in}} = 0$ e questo implica che il Full Adder di Fig. 2 si riduce ad un Half Adder. Pertanto iniziamo a costruire la tabella di verità che dovrà essere mappata, sotto forma di indirizzi di memoria, nella ROM. Come detto in precedenza in input avremo l'*opcode* per individuare l'operazione da svolgere e, solo per il primo modulo ALU, possiamo omettere c_{in} nella tabella. Inoltre inseriremo a_0, b_0 come input e r_0, c_0 come output:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0
0	0	0	0	1	0	1	0
0	0	0	0	1	1	0	1

Table 2. Tabella di verità dell'operazione $a_0 + b_0$ (senza segno) per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x00	0
0x01	2
0x02	2
0x03	1

Table 3. Tabella di verità dell'operazione $a_0 + b_0$ (senza segno) per il modulo ALU ad 1 bit in esadecimale

2.3.2 Shift a sinistra di una posizione di a

Per implementare uno shift a sinistra di una posizione di $a = \langle a_3 a_2 a_1 a_0 \rangle$ possiamo usare il risultato r_0 e il carry c_0, infatti lo shift a sinistra implica l'aggiunta di uno 0 a destra, quindi per il primo modulo ALU ad 1 bit, risulta sempre $r_0 = 0$ e $c_0 = a_0$. Per comprendere meglio questo concetto immaginiamo di effettuare tale operazione sul numero $a = (1)_{10} = (0001)_2$. L'operazione di shift trasforma il numero a nel risultato $r = (0010)_2 = (2)_{10}$, quindi come si può notare $r_0 = 0$ e $c_0 = a_0 = 1$. Sottolineo che questo vale solo per il primo caso e quindi per il primo modulo ALU ad 1 bit. Ora costruiamo la tabella di verità utilizzando gli stessi input e gli stessi output dell'operazione precedente:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0
0	0	0	1	1	0	0	1
0	0	0	1	1	1	0	1

Table 4. Tabella di verità dell'operazione shift a sinistra per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x04	0
0x05	0
0x06	1
0x07	1

Table 5. Tabella di verità dell'operazione shift a sinistra per il modulo ALU ad 1 bit in esadecimale

2.3.3 Operazione AND

Per questa operazione non c'è molto da dire tranne che il carry è sempre 0, quindi possiamo direttamente costruire la tabella di verità:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
0	0	1	0	0	0	0	0
0	0	1	0	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	0	1	1	1	0

Table 6. Tabella di verità dell'operazione a_0 AND b_0 per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
`0x08`	0
`0x09`	0
`0x0A`	0
`0x0B`	2

Table 7. Tabella di verità dell'operazione $a_0 \,\text{AND}\, b_0$ per il modulo ALU ad 1 bit in esadecimale

2.3.4 Operazione OR

Per questa operazione non c'è molto da dire tranne che il carry è sempre 0, quindi possiamo direttamente costruire la tabella di verità:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
0	1	0	0	0	0	0	0
0	1	0	0	0	1	1	0
0	1	0	0	1	0	1	0
0	1	0	0	1	1	1	0

Table 8. Tabella di verità dell'operazione $a_0 \,\text{OR}\, b_0$ per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
`0x10`	0
`0x11`	0
`0x12`	0
`0x13`	2

Table 9. Tabella di verità dell'operazione $a_0 \,\text{OR}\, b_0$ per il modulo ALU ad 1 bit in esadecimale

dove il valore $\langle \text{op}_0\text{op}_1\text{op}_2\text{op}_3 a_0 b_0 \rangle = (010000)_2 = (0001|0000)_2 = (10)_{16}$ e così via per gli altri.

2.3.5 Operazione XOR

Per questa operazione non c'è molto da dire tranne che il carry è sempre 0, quindi possiamo direttamente costruire la tabella di verità:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
0	1	1	0	0	0	0	0
0	1	1	0	0	1	1	0
0	1	1	0	1	0	1	0
0	1	1	0	1	1	0	0

Table 10. Tabella di verità dell'operazione $a_0 \oplus b_0$ per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x18	0
0x19	2
0x1A	2
0x1B	0

Table 11. Tabella di verità dell'operazione $a_0 \oplus b_0$ per il modulo ALU ad 1 bit in esadecimale

2.3.6 Operazione XNOR

Per questa operazione non c'è molto da dire tranne che il carry è sempre 0, quindi possiamo direttamente costruire la tabella di verità:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
1	0	0	0	0	0	1	0
1	0	0	0	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	0	1	1	1	0

Table 12. Tabella di verità dell'operazione $a_0 \odot b_0$ per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x20	2
0x21	0
0x22	0
0x23	2

Table 13. Tabella di verità dell'operazione $a_0 \odot b_0$ per il modulo ALU ad 1 bit in esadecimale

2.3.7 Operazione NAND

Per questa operazione non c'è molto da dire tranne che il carry è sempre 0, quindi possiamo direttamente costruire la tabella di verità:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
1	0	1	0	0	0	1	0
1	0	1	0	0	1	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	1	0	0

Table 14. Tabella di verità dell'operazione $a_0|b_0$ per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x28	2
0x29	2
0x2A	2
0x2B	0

Table 15. Tabella di verità dell'operazione $a_0|b_0$ per il modulo ALU ad 1 bit in esadecimale

2.3.8 Operazione NOR

Per questa operazione non c'è molto da dire tranne che il carry è sempre 0, quindi possiamo direttamente costruire la tabella di verità:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
1	1	0	0	0	0	1	0
1	1	0	0	0	1	0	0
1	1	0	0	1	0	0	0
1	1	0	0	1	1	0	0

Table 16. Tabella di verità dell'operazione $a_0 \downarrow b_0$ per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x30	2
0x31	0
0x32	0
0x33	0

Table 17. Tabella di verità dell'operazione $a_0 \downarrow b_0$ per il modulo ALU ad 1 bit in esadecimale

2.3.9 Operazione $\bar{a}$

Per questa operazione non c'è molto da dire tranne che il carry è sempre 0, quindi possiamo direttamente costruire la tabella di verità:

op_0	op_1	op_2	op_3	a_0	b_0	r_0	c_0
1	1	1	0	0	0	1	0
1	1	1	0	0	1	1	0
1	1	1	0	1	0	0	0
1	1	1	0	1	1	0	0

Table 18. Tabella di verità dell'operazione $\bar{a}$ per il modulo ALU ad 1 bit

Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x38	2
0x39	2
0x3A	0
0x3B	0

Table 19. Tabella di verità dell'operazione $\bar{a}$ per il modulo ALU ad 1 bit in esadecimale

2.3.10 Implementazione della ROM

Per usare la ROM con il software Digital occorre configurare gli indirizzi di memoria con i relativi valori di output precedentemente calcolati, come segue:

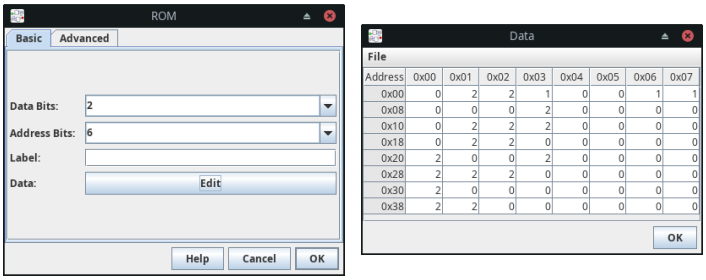

Figure 3. Configurazione della ROM su Digital

Inoltre come mostrato in Fig. 4, dove è riportato il modulo ROM che implementa un'ALU ad 1 bit, occorre prestare molta attenzione all'ordine in cui si mettono i bit in ingresso e in uscita, infatti l'ordine deve essere lo stesso delle tabelle di verità, nelle quali il bit in posizione 0 è b_0 mentre quello in posizione 1 è a_0, dal bit in posizione 2 al bit in posizione 5 c'è *opcode* ed infine per l'output abbiamo in posizione 1 il risultato r_0 e in posizione 0 il carry c_0 da propagare.

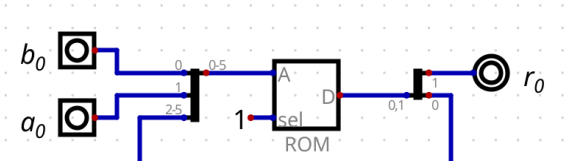

Figure 4. Modulo ALU ad 1 bit con ROM

Il modulo ALU ad 1 bit è completo quindi dobbiamo solo capire come modificarlo, per il modulo ALU successivo, in modo tale da poterlo copiare e incollare per ottenere un'ALU ad n bit.

2.4 Operazioni per il modulo ALU a 4 bit

2.4.1 Somma senza segno

Per come sono state realizzate le tabelle di verità precedenti la cosa più sensata da fare è inserire il carry c_{in} delle propagazioni "davanti" all'*opcode* modificando la tabella di verità come segue:

c_{in}	op_0	op_1	op_2	op_3	a_i	b_i	r_i	c_i
1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0	1
1	0	0	0	0	1	0	0	1
1	0	0	0	0	1	1	1	1

Table 20. Tabella di verità dell'operazione $a_i + b_i$ (senza segno) per l'ALU a 4 bit

dove $i \in [1, 3]$. Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x40	2
0x41	1
0x42	1
0x43	3

Table 21. Tabella di verità dell'operazione $a_i + b_i$ (senza segno) per l'ALU a 4 bit in esadecimale

2.4.2 Shift a sinistra di una posizione di a

Anche per questa operazione procediamo allo stesso modo:

c_{in}	op_0	op_1	op_2	op_3	a_i	b_i	r_i	c_i
1	0	0	0	1	0	0	1	0
1	0	0	0	1	0	1	1	0
1	0	0	0	1	1	0	1	1
1	0	0	0	1	1	1	1	1

Table 22. Tabella di verità dell'operazione shift a sinistra per l'ALU a 4 bit

dove $i \in [1, 3]$. Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
`0x44`	2
`0x45`	2
`0x46`	3
`0x47`	3

Table 23. Tabella di verità dell'operazione shift a sinistra per l'ALU a 4 bit in esadecimale

2.4.3 Le altre operazioni

Le operazioni rimaste rimangono identiche a meno del carry c_{in} pertanto le riassumo tutte in una singola tabella di verità:

c_{in}	op_0	op_1	op_2	op_3	a_i	b_i	r_i	c_i
1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	1	0	0
1	0	0	1	0	1	0	0	0
1	0	0	1	0	1	1	1	0
1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	1	0
1	0	1	0	0	1	0	1	0
1	0	1	0	0	1	1	1	0
1	0	1	1	0	0	0	0	0
1	0	1	1	0	0	1	1	0
1	0	1	1	0	1	0	1	0
1	0	1	1	0	1	1	0	0
1	1	0	0	0	0	0	1	0
1	1	0	0	0	0	1	0	0
1	1	0	0	0	1	0	0	0
1	1	0	0	0	1	1	1	0
1	1	0	1	0	0	0	1	0
1	1	0	1	0	0	1	1	0
1	1	0	1	0	1	0	1	0
1	1	0	1	0	1	1	0	0
1	1	1	0	0	0	0	1	0
1	1	1	0	0	0	1	0	0
1	1	1	0	0	1	0	0	0
1	1	1	0	0	1	1	0	0
1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	1	0
1	1	1	1	0	1	0	0	0
1	1	1	1	0	1	1	0	0

Table 24. Tabella di verità delle restanti operazioni per l'ALU a 4 bit

dove $i \in [1, 3]$. Tuttavia tale tabella presenta gli input e gli output in binario, quindi occorre convertire tali valori in esadecimale, per poterli inserire nella ROM, ottenendo:

input	output
0x48	0
0x49	0
0x4A	0
0x4B	2
0x50	0
0x51	2
0x52	2
0x53	2
0x58	0
0x59	2
0x5A	2
0x5B	0
0x60	2
0x61	0
0x62	0
0x63	2
0x68	2
0x69	2
0x6A	2
0x6B	0
0x70	2
0x71	0
0x72	0
0x73	0
0x78	2
0x79	2
0x7A	0
0x7B	0

Table 25. Tabella di verità delle restanti operazioni per l'ALU a 4 bit in esadecimale

2.4.4 Implementazione della ROM

Per usare la ROM con il software Digital occorre configurare gli indirizzi di memoria con i relativi valori di output precedentemente calcolati, come segue:

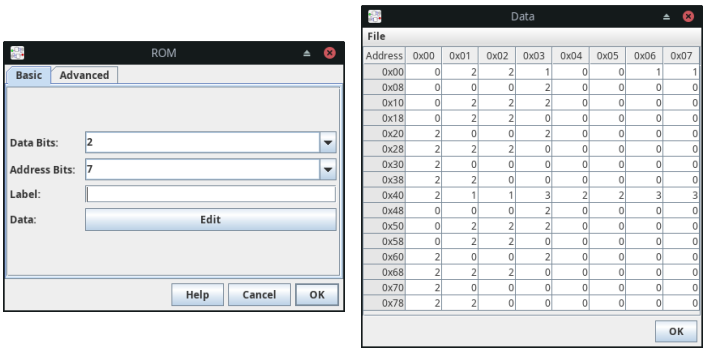

Figure 5. Configurazione della ROM su Digital

In conclusione l'ALU a 4 bit completa può essere realizzata come mostrato in Fig. 6.

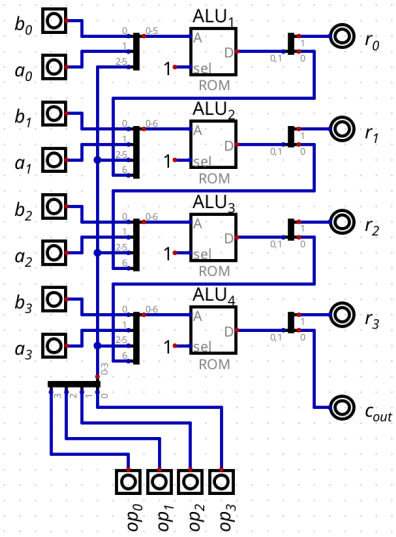

Figure 6. ALU a 4 bit completa implementata con ROM

2.5 Conclusioni

Progettare un'ALU a 4 bit è un compito che un ingegnere deve fare almeno una volta nella vita. Sarebbe interessante ampliare il progetto realizzando anche i registri, il modulo clock, il program counter e tutto ciò che serve per realizzare un vero e proprio computer a 4 bit da realizzare su breadboard o, ancora meglio, progettando il PCB tramite KiCAD.