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.

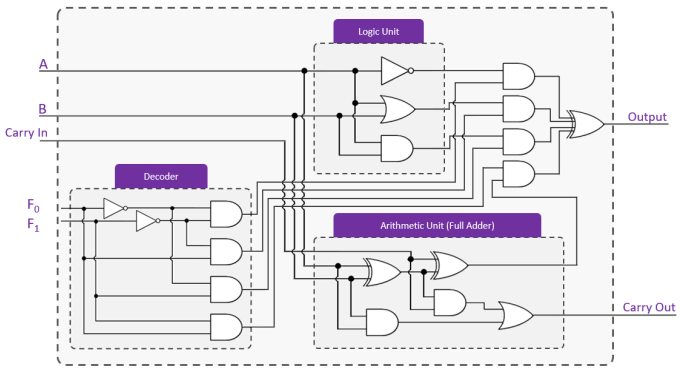

Figure 1. ALU ad 1 bit

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.

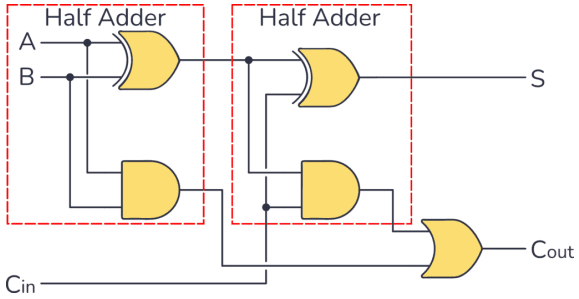

Figure 2. Full Adder

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:

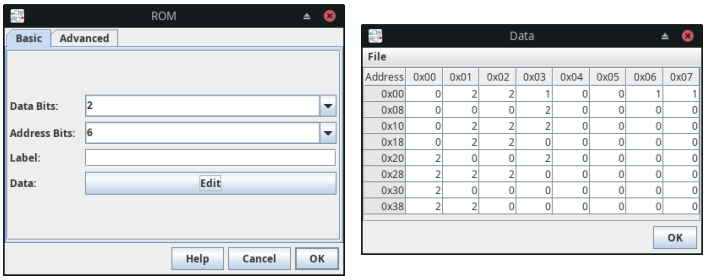

Figure 3. Configurazione della ROM su Digital

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.

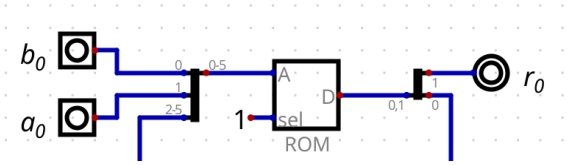

Figure 4. Modulo ALU ad 1 bit con ROM

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:

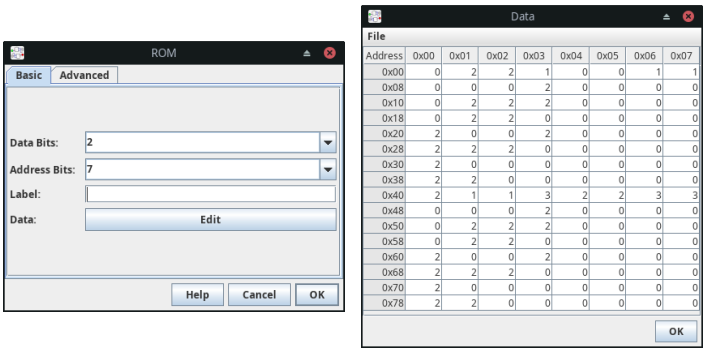

Figure 5. Configurazione della ROM su Digital

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

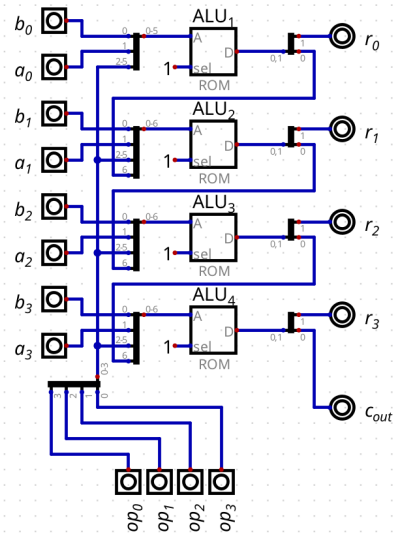

Figure 6. ALU a 4 bit completa implementata con ROM

In aggiunta riporto anche il codice HDL relativo ai linguaggi di descrizione hardware VHDL:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

entity DIG_ROM_ALU_1 is
  port (
    D: out std_logic_vector (1 downto 0);
    A: in std_logic_vector (5 downto 0);
    sel: in std_logic );
end DIG_ROM_ALU_1;

architecture Behavioral of DIG_ROM_ALU_1 is
  type mem is array ( 0 to 57) of std_logic_vector (1 downto 0);
  constant my_Rom : mem := (
    "00", "10", "10", "01", "00", "00", "01", "01", "00", "00", "00", "10",
    "00", "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00",
    "00", "10", "10", "00", "00", "00", "00", "00", "10", "00", "00", "10",
    "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00", "00",
    "10", "00", "00", "00", "00", "00", "00", "00", "10", "10");
begin
  process (A, sel)
  begin
    if sel='0' then
      D <= (others => 'Z');
    elsif A > "111001" then
      D <= (others => '0');
    else
      D <= my_rom(to_integer(unsigned(A)));
    end if;
  end process;
end Behavioral;


```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

entity DIG_ROM_ALU_2 is
  port (
    D: out std_logic_vector (1 downto 0);
    A: in std_logic_vector (6 downto 0);
    sel: in std_logic );
end DIG_ROM_ALU_2;

architecture Behavioral of DIG_ROM_ALU_2 is
  type mem is array ( 0 to 121) of std_logic_vector (1 downto 0);
  constant my_Rom : mem := (
    "00", "10", "10", "01", "00", "00", "01", "01", "00", "00", "00", "10",
    "00", "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00",
    "00", "10", "10", "00", "00", "00", "00", "00", "10", "00", "00", "10",
    "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00", "00",
    "10", "00", "00", "00", "00", "00", "00", "00", "10", "10", "00", "00",
    "00", "00", "00", "00", "10", "01", "01", "11", "10", "10", "11", "11",
    "00", "00", "00", "10", "00", "00", "00", "00", "00", "10", "10", "10",
    "00", "00", "00", "00", "00", "10", "10", "00", "00", "00", "00", "00",
    "10", "00", "00", "10", "00", "00", "00", "00", "10", "10", "10", "00",
    "00", "00", "00", "00", "10", "00", "00", "00", "00", "00", "00", "00",
    "10", "10");
begin
  process (A, sel)
  begin
    if sel='0' then
      D <= (others => 'Z');
    elsif A > "1111001" then
      D <= (others => '0');
    else
      D <= my_rom(to_integer(unsigned(A)));
    end if;
  end process;
end Behavioral;


LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

entity DIG_ROM_ALU_3 is
  port (
    D: out std_logic_vector (1 downto 0);
    A: in std_logic_vector (6 downto 0);
    sel: in std_logic );
end DIG_ROM_ALU_3;

architecture Behavioral of DIG_ROM_ALU_3 is
  type mem is array ( 0 to 121) of std_logic_vector (1 downto 0);
  constant my_Rom : mem := (
    "00", "10", "10", "01", "00", "00", "01", "01", "00", "00", "00", "10",
    "00", "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00",
    "00", "10", "10", "00", "00", "00", "00", "00", "10", "00", "00", "10",
```

```
    "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00", "00",
    "10", "00", "00", "00", "00", "00", "00", "00", "10", "10", "00", "00",
    "00", "00", "00", "00", "10", "01", "01", "11", "10", "10", "11", "11",
    "00", "00", "00", "10", "00", "00", "00", "00", "00", "10", "10", "10",
    "00", "00", "00", "00", "00", "10", "10", "00", "00", "00", "00", "00",
    "10", "00", "00", "10", "00", "00", "00", "00", "10", "10", "10", "00",
    "00", "00", "00", "00", "10", "00", "00", "00", "00", "00", "00", "00",
    "10", "10");
begin
  process (A, sel)
  begin
    if sel='0' then
      D <= (others => 'Z');
    elsif A > "1111001" then
      D <= (others => '0');
    else
      D <= my_rom(to_integer(unsigned(A)));
    end if;
  end process;
end Behavioral;


LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;

entity DIG_ROM_ALU_4 is
  port (
    D: out std_logic_vector (1 downto 0);
    A: in std_logic_vector (6 downto 0);
    sel: in std_logic );
end DIG_ROM_ALU_4;

architecture Behavioral of DIG_ROM_ALU_4 is
  type mem is array ( 0 to 121) of std_logic_vector (1 downto 0);
  constant my_Rom : mem := (
    "00", "10", "10", "01", "00", "00", "01", "01", "00", "00", "00", "10",
    "00", "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00",
    "00", "10", "10", "00", "00", "00", "00", "00", "10", "00", "00", "10",
    "00", "00", "00", "00", "10", "10", "10", "00", "00", "00", "00", "00",
    "10", "00", "00", "00", "00", "00", "00", "00", "10", "10", "00", "00",
    "00", "00", "00", "00", "10", "01", "01", "11", "10", "10", "11", "11",
    "00", "00", "00", "10", "00", "00", "00", "00", "00", "10", "10", "10",
    "00", "00", "00", "00", "00", "10", "10", "00", "00", "00", "00", "00",
    "10", "00", "00", "10", "00", "00", "00", "00", "10", "10", "10", "00",
    "00", "00", "00", "00", "10", "00", "00", "00", "00", "00", "00", "00",
    "10", "10");
begin
  process (A, sel)
  begin
    if sel='0' then
      D <= (others => 'Z');
    elsif A > "1111001" then
      D <= (others => '0');
    else
      D <= my_rom(to_integer(unsigned(A)));
```

```
146     end if;
147   end process;
148 end Behavioral;
149 
150 
151 LIBRARY ieee;
152 USE ieee.std_logic_1164.all;
153 USE ieee.numeric_std.all;
154 
155 entity main is
156   port (
157     b_0: in std_logic;
158     a_0: in std_logic;
159     b_1: in std_logic;
160     a_1: in std_logic;
161     b_2: in std_logic;
162     a_2: in std_logic;
163     b_3: in std_logic;
164     a_3: in std_logic;
165     op_1: in std_logic;
166     op_2: in std_logic;
167     op_3: in std_logic;
168     op_0: in std_logic;
169     r_0: out std_logic;
170     r_1: out std_logic;
171     r_2: out std_logic;
172     r_3: out std_logic;
173     c_out: out std_logic);
174 end main;
175 
176 architecture Behavioral of main is
177   signal s0: std_logic_vector(5 downto 0);
178   signal s1: std_logic_vector(1 downto 0);
179   signal s2: std_logic_vector(3 downto 0);
180   signal s3: std_logic_vector(6 downto 0);
181   signal s4: std_logic_vector(1 downto 0);
182   signal s5: std_logic_vector(6 downto 0);
183   signal s6: std_logic_vector(6 downto 0);
184   signal s7: std_logic_vector(1 downto 0);
185   signal s8: std_logic_vector(1 downto 0);
186 begin
187   s2(0) <= op_3;
188   s2(1) <= op_2;
189   s2(2) <= op_1;
190   s2(3) <= op_0;
191   s0(0) <= b_0;
192   s0(1) <= a_0;
193   s0(5 downto 2) <= s2;
194   gate0: entity work.DIG_ROM_ALU_1 -- ALU_1
195     port map (
196       A => s0,
197       sel => '1',
198       D => s1);
199   s3(0) <= b_1;
200   s3(1) <= a_1;
201   s3(5 downto 2) <= s2;
```

```
  s3(6) <= s1(0);
  r_0 <= s1(1);
  gate1: entity work.DIG_ROM_ALU_2 -- ALU_2
    port map (
      A => s3,
      sel => '1',
      D => s4);
  s5(0) <= b_2;
  s5(1) <= a_2;
  s5(5 downto 2) <= s2;
  s5(6) <= s4(0);
  r_1 <= s4(1);
  gate2: entity work.DIG_ROM_ALU_3 -- ALU_3
    port map (
      A => s5,
      sel => '1',
      D => s8);
  s6(0) <= b_3;
  s6(1) <= a_3;
  s6(5 downto 2) <= s2;
  s6(6) <= s8(0);
  r_2 <= s8(1);
  gate3: entity work.DIG_ROM_ALU_4 -- ALU_4
    port map (
      A => s6,
      sel => '1',
      D => s7);
  c_out <= s7(0);
  r_3 <= s7(1);
end Behavioral;
```

e Verilog:

```
module DIG_ROM_64X2_ALU_1 (
    input [5:0] A,
    input sel,
    output reg [1:0] D
);
    reg [1:0] my_rom [0:57];

    always @ (*) begin
        if (~sel)
            D = 2'hz;
        else if (A > 6'h39)
            D = 2'h0;
        else
            D = my_rom[A];
    end

    initial begin
        my_rom[0] = 2'h0;
        my_rom[1] = 2'h2;
        my_rom[2] = 2'h2;
        my_rom[3] = 2'h1;
        my_rom[4] = 2'h0;
```

```
23          my_rom[5] = 2'h0;
24          my_rom[6] = 2'h1;
25          my_rom[7] = 2'h1;
26          my_rom[8] = 2'h0;
27          my_rom[9] = 2'h0;
28          my_rom[10] = 2'h0;
29          my_rom[11] = 2'h2;
30          my_rom[12] = 2'h0;
31          my_rom[13] = 2'h0;
32          my_rom[14] = 2'h0;
33          my_rom[15] = 2'h0;
34          my_rom[16] = 2'h0;
35          my_rom[17] = 2'h2;
36          my_rom[18] = 2'h2;
37          my_rom[19] = 2'h2;
38          my_rom[20] = 2'h0;
39          my_rom[21] = 2'h0;
40          my_rom[22] = 2'h0;
41          my_rom[23] = 2'h0;
42          my_rom[24] = 2'h0;
43          my_rom[25] = 2'h2;
44          my_rom[26] = 2'h2;
45          my_rom[27] = 2'h0;
46          my_rom[28] = 2'h0;
47          my_rom[29] = 2'h0;
48          my_rom[30] = 2'h0;
49          my_rom[31] = 2'h0;
50          my_rom[32] = 2'h2;
51          my_rom[33] = 2'h0;
52          my_rom[34] = 2'h0;
53          my_rom[35] = 2'h2;
54          my_rom[36] = 2'h0;
55          my_rom[37] = 2'h0;
56          my_rom[38] = 2'h0;
57          my_rom[39] = 2'h0;
58          my_rom[40] = 2'h2;
59          my_rom[41] = 2'h2;
60          my_rom[42] = 2'h2;
61          my_rom[43] = 2'h0;
62          my_rom[44] = 2'h0;
63          my_rom[45] = 2'h0;
64          my_rom[46] = 2'h0;
65          my_rom[47] = 2'h0;
66          my_rom[48] = 2'h2;
67          my_rom[49] = 2'h0;
68          my_rom[50] = 2'h0;
69          my_rom[51] = 2'h0;
70          my_rom[52] = 2'h0;
71          my_rom[53] = 2'h0;
72          my_rom[54] = 2'h0;
73          my_rom[55] = 2'h0;
74          my_rom[56] = 2'h2;
75          my_rom[57] = 2'h2;
76      end
77  endmodule
78
```

```
module DIG_ROM_128X2_ALU_2 (
    input [6:0] A,
    input sel,
    output reg [1:0] D
);
    reg [1:0] my_rom [0:121];

    always @ (*) begin
        if (~sel)
            D = 2'hz;
        else if (A > 7'h79)
            D = 2'h0;
        else
            D = my_rom[A];
    end

    initial begin
        my_rom[0] = 2'h0;
        my_rom[1] = 2'h2;
        my_rom[2] = 2'h2;
        my_rom[3] = 2'h1;
        my_rom[4] = 2'h0;
        my_rom[5] = 2'h0;
        my_rom[6] = 2'h1;
        my_rom[7] = 2'h1;
        my_rom[8] = 2'h0;
        my_rom[9] = 2'h0;
        my_rom[10] = 2'h0;
        my_rom[11] = 2'h2;
        my_rom[12] = 2'h0;
        my_rom[13] = 2'h0;
        my_rom[14] = 2'h0;
        my_rom[15] = 2'h0;
        my_rom[16] = 2'h0;
        my_rom[17] = 2'h2;
        my_rom[18] = 2'h2;
        my_rom[19] = 2'h2;
        my_rom[20] = 2'h0;
        my_rom[21] = 2'h0;
        my_rom[22] = 2'h0;
        my_rom[23] = 2'h0;
        my_rom[24] = 2'h0;
        my_rom[25] = 2'h2;
        my_rom[26] = 2'h2;
        my_rom[27] = 2'h0;
        my_rom[28] = 2'h0;
        my_rom[29] = 2'h0;
        my_rom[30] = 2'h0;
        my_rom[31] = 2'h0;
        my_rom[32] = 2'h2;
        my_rom[33] = 2'h0;
        my_rom[34] = 2'h0;
        my_rom[35] = 2'h2;
        my_rom[36] = 2'h0;
        my_rom[37] = 2'h0;
        my_rom[38] = 2'h0;
```

```
135         my_rom[39] = 2'h0;
136         my_rom[40] = 2'h2;
137         my_rom[41] = 2'h2;
138         my_rom[42] = 2'h2;
139         my_rom[43] = 2'h0;
140         my_rom[44] = 2'h0;
141         my_rom[45] = 2'h0;
142         my_rom[46] = 2'h0;
143         my_rom[47] = 2'h0;
144         my_rom[48] = 2'h2;
145         my_rom[49] = 2'h0;
146         my_rom[50] = 2'h0;
147         my_rom[51] = 2'h0;
148         my_rom[52] = 2'h0;
149         my_rom[53] = 2'h0;
150         my_rom[54] = 2'h0;
151         my_rom[55] = 2'h0;
152         my_rom[56] = 2'h2;
153         my_rom[57] = 2'h2;
154         my_rom[58] = 2'h0;
155         my_rom[59] = 2'h0;
156         my_rom[60] = 2'h0;
157         my_rom[61] = 2'h0;
158         my_rom[62] = 2'h0;
159         my_rom[63] = 2'h0;
160         my_rom[64] = 2'h2;
161         my_rom[65] = 2'h1;
162         my_rom[66] = 2'h1;
163         my_rom[67] = 2'h3;
164         my_rom[68] = 2'h2;
165         my_rom[69] = 2'h2;
166         my_rom[70] = 2'h3;
167         my_rom[71] = 2'h3;
168         my_rom[72] = 2'h0;
169         my_rom[73] = 2'h0;
170         my_rom[74] = 2'h0;
171         my_rom[75] = 2'h2;
172         my_rom[76] = 2'h0;
173         my_rom[77] = 2'h0;
174         my_rom[78] = 2'h0;
175         my_rom[79] = 2'h0;
176         my_rom[80] = 2'h0;
177         my_rom[81] = 2'h2;
178         my_rom[82] = 2'h2;
179         my_rom[83] = 2'h2;
180         my_rom[84] = 2'h0;
181         my_rom[85] = 2'h0;
182         my_rom[86] = 2'h0;
183         my_rom[87] = 2'h0;
184         my_rom[88] = 2'h0;
185         my_rom[89] = 2'h2;
186         my_rom[90] = 2'h2;
187         my_rom[91] = 2'h0;
188         my_rom[92] = 2'h0;
189         my_rom[93] = 2'h0;
190         my_rom[94] = 2'h0;
```

```
        my_rom[95] = 2'h0;
        my_rom[96] = 2'h2;
        my_rom[97] = 2'h0;
        my_rom[98] = 2'h0;
        my_rom[99] = 2'h2;
        my_rom[100] = 2'h0;
        my_rom[101] = 2'h0;
        my_rom[102] = 2'h0;
        my_rom[103] = 2'h0;
        my_rom[104] = 2'h2;
        my_rom[105] = 2'h2;
        my_rom[106] = 2'h2;
        my_rom[107] = 2'h0;
        my_rom[108] = 2'h0;
        my_rom[109] = 2'h0;
        my_rom[110] = 2'h0;
        my_rom[111] = 2'h0;
        my_rom[112] = 2'h2;
        my_rom[113] = 2'h0;
        my_rom[114] = 2'h0;
        my_rom[115] = 2'h0;
        my_rom[116] = 2'h0;
        my_rom[117] = 2'h0;
        my_rom[118] = 2'h0;
        my_rom[119] = 2'h0;
        my_rom[120] = 2'h2;
        my_rom[121] = 2'h2;
    end
endmodule

module DIG_ROM_128X2_ALU_3 (
    input [6:0] A,
    input sel,
    output reg [1:0] D
);
    reg [1:0] my_rom [0:121];

    always @ (*) begin
        if (~sel)
            D = 2'hz;
        else if (A > 7'h79)
            D = 2'h0;
        else
            D = my_rom[A];
    end

    initial begin
        my_rom[0] = 2'h0;
        my_rom[1] = 2'h2;
        my_rom[2] = 2'h2;
        my_rom[3] = 2'h1;
        my_rom[4] = 2'h0;
        my_rom[5] = 2'h0;
        my_rom[6] = 2'h1;
        my_rom[7] = 2'h1;
        my_rom[8] = 2'h0;
```

```
247         my_rom[9] = 2'h0;
248         my_rom[10] = 2'h0;
249         my_rom[11] = 2'h2;
250         my_rom[12] = 2'h0;
251         my_rom[13] = 2'h0;
252         my_rom[14] = 2'h0;
253         my_rom[15] = 2'h0;
254         my_rom[16] = 2'h0;
255         my_rom[17] = 2'h2;
256         my_rom[18] = 2'h2;
257         my_rom[19] = 2'h2;
258         my_rom[20] = 2'h0;
259         my_rom[21] = 2'h0;
260         my_rom[22] = 2'h0;
261         my_rom[23] = 2'h0;
262         my_rom[24] = 2'h0;
263         my_rom[25] = 2'h2;
264         my_rom[26] = 2'h2;
265         my_rom[27] = 2'h0;
266         my_rom[28] = 2'h0;
267         my_rom[29] = 2'h0;
268         my_rom[30] = 2'h0;
269         my_rom[31] = 2'h0;
270         my_rom[32] = 2'h2;
271         my_rom[33] = 2'h0;
272         my_rom[34] = 2'h0;
273         my_rom[35] = 2'h2;
274         my_rom[36] = 2'h0;
275         my_rom[37] = 2'h0;
276         my_rom[38] = 2'h0;
277         my_rom[39] = 2'h0;
278         my_rom[40] = 2'h2;
279         my_rom[41] = 2'h2;
280         my_rom[42] = 2'h2;
281         my_rom[43] = 2'h0;
282         my_rom[44] = 2'h0;
283         my_rom[45] = 2'h0;
284         my_rom[46] = 2'h0;
285         my_rom[47] = 2'h0;
286         my_rom[48] = 2'h2;
287         my_rom[49] = 2'h0;
288         my_rom[50] = 2'h0;
289         my_rom[51] = 2'h0;
290         my_rom[52] = 2'h0;
291         my_rom[53] = 2'h0;
292         my_rom[54] = 2'h0;
293         my_rom[55] = 2'h0;
294         my_rom[56] = 2'h2;
295         my_rom[57] = 2'h2;
296         my_rom[58] = 2'h0;
297         my_rom[59] = 2'h0;
298         my_rom[60] = 2'h0;
299         my_rom[61] = 2'h0;
300         my_rom[62] = 2'h0;
301         my_rom[63] = 2'h0;
302         my_rom[64] = 2'h2;
```

```
303         my_rom[65] = 2'h1;
304         my_rom[66] = 2'h1;
305         my_rom[67] = 2'h3;
306         my_rom[68] = 2'h2;
307         my_rom[69] = 2'h2;
308         my_rom[70] = 2'h3;
309         my_rom[71] = 2'h3;
310         my_rom[72] = 2'h0;
311         my_rom[73] = 2'h0;
312         my_rom[74] = 2'h0;
313         my_rom[75] = 2'h2;
314         my_rom[76] = 2'h0;
315         my_rom[77] = 2'h0;
316         my_rom[78] = 2'h0;
317         my_rom[79] = 2'h0;
318         my_rom[80] = 2'h0;
319         my_rom[81] = 2'h2;
320         my_rom[82] = 2'h2;
321         my_rom[83] = 2'h2;
322         my_rom[84] = 2'h0;
323         my_rom[85] = 2'h0;
324         my_rom[86] = 2'h0;
325         my_rom[87] = 2'h0;
326         my_rom[88] = 2'h0;
327         my_rom[89] = 2'h2;
328         my_rom[90] = 2'h2;
329         my_rom[91] = 2'h0;
330         my_rom[92] = 2'h0;
331         my_rom[93] = 2'h0;
332         my_rom[94] = 2'h0;
333         my_rom[95] = 2'h0;
334         my_rom[96] = 2'h2;
335         my_rom[97] = 2'h0;
336         my_rom[98] = 2'h0;
337         my_rom[99] = 2'h2;
338         my_rom[100] = 2'h0;
339         my_rom[101] = 2'h0;
340         my_rom[102] = 2'h0;
341         my_rom[103] = 2'h0;
342         my_rom[104] = 2'h2;
343         my_rom[105] = 2'h2;
344         my_rom[106] = 2'h2;
345         my_rom[107] = 2'h0;
346         my_rom[108] = 2'h0;
347         my_rom[109] = 2'h0;
348         my_rom[110] = 2'h0;
349         my_rom[111] = 2'h0;
350         my_rom[112] = 2'h2;
351         my_rom[113] = 2'h0;
352         my_rom[114] = 2'h0;
353         my_rom[115] = 2'h0;
354         my_rom[116] = 2'h0;
355         my_rom[117] = 2'h0;
356         my_rom[118] = 2'h0;
357         my_rom[119] = 2'h0;
358         my_rom[120] = 2'h2;
```

```
359         my_rom[121] = 2'h2;
360     end
361 endmodule
362
363 module DIG_ROM_128X2_ALU_4 (
364     input [6:0] A,
365     input sel,
366     output reg [1:0] D
367 );
368     reg [1:0] my_rom [0:121];
369
370     always @ (*) begin
371         if (~sel)
372             D = 2'hz;
373         else if (A > 7'h79)
374             D = 2'h0;
375         else
376             D = my_rom[A];
377     end
378
379     initial begin
380         my_rom[0] = 2'h0;
381         my_rom[1] = 2'h2;
382         my_rom[2] = 2'h2;
383         my_rom[3] = 2'h1;
384         my_rom[4] = 2'h0;
385         my_rom[5] = 2'h0;
386         my_rom[6] = 2'h1;
387         my_rom[7] = 2'h1;
388         my_rom[8] = 2'h0;
389         my_rom[9] = 2'h0;
390         my_rom[10] = 2'h0;
391         my_rom[11] = 2'h2;
392         my_rom[12] = 2'h0;
393         my_rom[13] = 2'h0;
394         my_rom[14] = 2'h0;
395         my_rom[15] = 2'h0;
396         my_rom[16] = 2'h0;
397         my_rom[17] = 2'h2;
398         my_rom[18] = 2'h2;
399         my_rom[19] = 2'h2;
400         my_rom[20] = 2'h0;
401         my_rom[21] = 2'h0;
402         my_rom[22] = 2'h0;
403         my_rom[23] = 2'h0;
404         my_rom[24] = 2'h0;
405         my_rom[25] = 2'h2;
406         my_rom[26] = 2'h2;
407         my_rom[27] = 2'h0;
408         my_rom[28] = 2'h0;
409         my_rom[29] = 2'h0;
410         my_rom[30] = 2'h0;
411         my_rom[31] = 2'h0;
412         my_rom[32] = 2'h2;
413         my_rom[33] = 2'h0;
414         my_rom[34] = 2'h0;
```

```
415         my_rom[35] = 2'h2;
416         my_rom[36] = 2'h0;
417         my_rom[37] = 2'h0;
418         my_rom[38] = 2'h0;
419         my_rom[39] = 2'h0;
420         my_rom[40] = 2'h2;
421         my_rom[41] = 2'h2;
422         my_rom[42] = 2'h2;
423         my_rom[43] = 2'h0;
424         my_rom[44] = 2'h0;
425         my_rom[45] = 2'h0;
426         my_rom[46] = 2'h0;
427         my_rom[47] = 2'h0;
428         my_rom[48] = 2'h2;
429         my_rom[49] = 2'h0;
430         my_rom[50] = 2'h0;
431         my_rom[51] = 2'h0;
432         my_rom[52] = 2'h0;
433         my_rom[53] = 2'h0;
434         my_rom[54] = 2'h0;
435         my_rom[55] = 2'h0;
436         my_rom[56] = 2'h2;
437         my_rom[57] = 2'h2;
438         my_rom[58] = 2'h0;
439         my_rom[59] = 2'h0;
440         my_rom[60] = 2'h0;
441         my_rom[61] = 2'h0;
442         my_rom[62] = 2'h0;
443         my_rom[63] = 2'h0;
444         my_rom[64] = 2'h2;
445         my_rom[65] = 2'h1;
446         my_rom[66] = 2'h1;
447         my_rom[67] = 2'h3;
448         my_rom[68] = 2'h2;
449         my_rom[69] = 2'h2;
450         my_rom[70] = 2'h3;
451         my_rom[71] = 2'h3;
452         my_rom[72] = 2'h0;
453         my_rom[73] = 2'h0;
454         my_rom[74] = 2'h0;
455         my_rom[75] = 2'h2;
456         my_rom[76] = 2'h0;
457         my_rom[77] = 2'h0;
458         my_rom[78] = 2'h0;
459         my_rom[79] = 2'h0;
460         my_rom[80] = 2'h0;
461         my_rom[81] = 2'h2;
462         my_rom[82] = 2'h2;
463         my_rom[83] = 2'h2;
464         my_rom[84] = 2'h0;
465         my_rom[85] = 2'h0;
466         my_rom[86] = 2'h0;
467         my_rom[87] = 2'h0;
468         my_rom[88] = 2'h0;
469         my_rom[89] = 2'h2;
470         my_rom[90] = 2'h2;
```

```
471         my_rom[91] = 2'h0;
472         my_rom[92] = 2'h0;
473         my_rom[93] = 2'h0;
474         my_rom[94] = 2'h0;
475         my_rom[95] = 2'h0;
476         my_rom[96] = 2'h2;
477         my_rom[97] = 2'h0;
478         my_rom[98] = 2'h0;
479         my_rom[99] = 2'h2;
480         my_rom[100] = 2'h0;
481         my_rom[101] = 2'h0;
482         my_rom[102] = 2'h0;
483         my_rom[103] = 2'h0;
484         my_rom[104] = 2'h2;
485         my_rom[105] = 2'h2;
486         my_rom[106] = 2'h2;
487         my_rom[107] = 2'h0;
488         my_rom[108] = 2'h0;
489         my_rom[109] = 2'h0;
490         my_rom[110] = 2'h0;
491         my_rom[111] = 2'h0;
492         my_rom[112] = 2'h2;
493         my_rom[113] = 2'h0;
494         my_rom[114] = 2'h0;
495         my_rom[115] = 2'h0;
496         my_rom[116] = 2'h0;
497         my_rom[117] = 2'h0;
498         my_rom[118] = 2'h0;
499         my_rom[119] = 2'h0;
500         my_rom[120] = 2'h2;
501         my_rom[121] = 2'h2;
502     end
503 endmodule
504
505
506 module rom (
507   input b_0,
508   input a_0,
509   input b_1,
510   input a_1,
511   input b_2,
512   input a_2,
513   input b_3,
514   input a_3,
515   input op_1,
516   input op_2,
517   input op_3,
518   input op_0,
519   output r_0,
520   output r_1,
521   output r_2,
522   output r_3,
523   output c_out
524 );
525   wire [5:0] s0;
526   wire [1:0] s1;
```

```
527   wire [3:0] s2;
528   wire [6:0] s3;
529   wire [1:0] s4;
530   wire [6:0] s5;
531   wire [6:0] s6;
532   wire [1:0] s7;
533   wire [1:0] s8;
534   assign s2[0] = op_3;
535   assign s2[1] = op_2;
536   assign s2[2] = op_1;
537   assign s2[3] = op_0;
538   assign s0[0] = b_0;
539   assign s0[1] = a_0;
540   assign s0[5:2] = s2;
541   // ALU_1
542   DIG_ROM_64X2_ALU_1 DIG_ROM_64X2_ALU_1_i0 (
543     .A( s0 ),
544     .sel( 1'b1 ),
545     .D( s1 )
546   );
547   assign s3[0] = b_1;
548   assign s3[1] = a_1;
549   assign s3[5:2] = s2;
550   assign s3[6] = s1[0];
551   assign r_0 = s1[1];
552   // ALU_2
553   DIG_ROM_128X2_ALU_2 DIG_ROM_128X2_ALU_2_i1 (
554     .A( s3 ),
555     .sel( 1'b1 ),
556     .D( s4 )
557   );
558   assign s5[0] = b_2;
559   assign s5[1] = a_2;
560   assign s5[5:2] = s2;
561   assign s5[6] = s4[0];
562   assign r_1 = s4[1];
563   // ALU_3
564   DIG_ROM_128X2_ALU_3 DIG_ROM_128X2_ALU_3_i2 (
565     .A( s5 ),
566     .sel( 1'b1 ),
567     .D( s8 )
568   );
569   assign s6[0] = b_3;
570   assign s6[1] = a_3;
571   assign s6[5:2] = s2;
572   assign s6[6] = s8[0];
573   assign r_2 = s8[1];
574   // ALU_4
575   DIG_ROM_128X2_ALU_4 DIG_ROM_128X2_ALU_4_i3 (
576     .A( s6 ),
577     .sel( 1'b1 ),
578     .D( s7 )
579   );
580   assign c_out = s7[0];
581   assign r_3 = s7[1];
582 endmodule
```

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.