

ΑΡΙΘΜΗΤΙΚΗ & ΛΟΓΙΚΗ ΜΟΝΑΔΑ - ALU

2

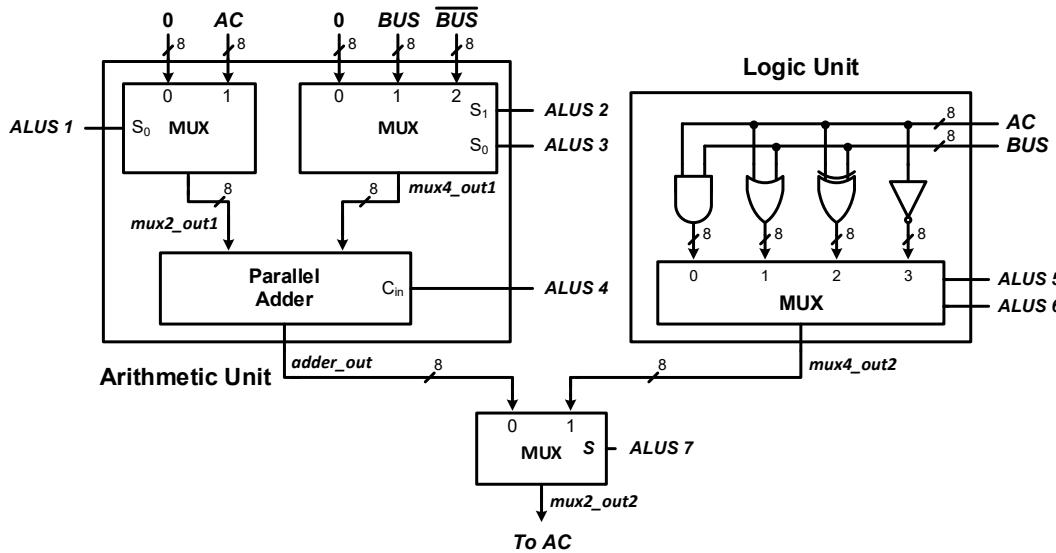
Σκοπός

Με αφορμή την σχεδίαση και την εξομοίωση με διάφορους τρόπους, απλών ψηφιακών κυκλωμάτων θα κατακτηθεί το αντικείμενο της άσκησης αυτής που είναι η σχεδίαση μιας αριθμητικής λογικής μονάδας (ALU) η οποία θα χρησιμοποιηθεί κατά την τελική σύνθεση της σχετικά απλής KME.

Συνοπτικά οι λειτουργίες που θα πρέπει να υλοποιεί η ALU είναι δέκα (10) συνολικά, οι οποίες μπορούν να χωριστούν σε τρεις κατηγορίες :

- Λογικές πράξεις μεταξύ δύο αριθμών των 8-bits : **AND, OR, XOR**
- Αριθμητικές πράξεις μεταξύ δύο αριθμών των 8-bits : **ADD, SUB**
- Διεργασίες που αφορούν το συσσωρευτή : **NOT, CLAC, INAC, MOVR, LDAC**

Το Σχήμα 1 απεικονίζει αναλυτικά την ALU.



Σχήμα 1: Λογικό διάγραμμα Αριθμητικής & Λογικής Μονάδας.

Για την υλοποίησή της ALU θα γραφεί κώδικας περιγραφής υλικού σε VHDL για κάθε υποκύκλωμα της ξεχωριστά και στη συνέχεια αυτά να χρησιμοποιηθούν σαν στοιχεία (components) για τη σύνθεση του τελικού κώδικα. Τα στοιχεία που απαιτούνται βάσει του παραπάνω σχήματος (**Σχήμα**

1), είναι τέσσερις (4) πολυπλέκτες και ένας παράλληλος αθροιστής (parallel adder) των 8-bits με αρχικό κρατούμενο. Για τις λογικές συναρτήσεις δεν απαιτείται η συγγραφή ξεχωριστού στοιχείου αφού αποτελούν τελεστές της γλώσσας VHDL.

Παράλληλος Αθροιστής

Η υλοποίηση σε VHDL του παράλληλου αθροιστή των 8-bits θα γίνει με το structural μοντέλο περιγραφής. Αυτό σημαίνει ότι θα πρέπει αρχικά να γραφεί κώδικας για το στοιχείο του πλήρη αθροιστή, το οποίο στη συνέχεια θα κληθεί οκτώ (8) φορές στο πρόγραμμα περιγραφής του παράλληλου αθροιστή των 8-bits.

Γράψτε τον κώδικα για τον πλήρη αθροιστή με σήματα εισόδου a , b , c_{in} και σήματα εξόδου s και c_{out} εύρους 1 bit. Το κύκλωμα αυτό όπως είναι γνωστό ότι υλοποιεί την αριθμητική πράξη της πρόσθεσης ανάμεσα στις εισόδου (a) και (b) λαμβάνοντας υπόψιν και την τιμή του κρατούμενου εισόδου (c_{in}) και υπολογίζει το άθροισμα (s) και το κρατούμενο εξόδου(c_{out}). Να δώσετε όνομα στην entity adder1bit.

Γράψτε εδώ το πρόγραμμά σας:

Πρόγραμμα 1: Ο πλήρης αθροιστής.

```
library ieee;
use ieee.std_logic_1164.all;
entity adder1bit is
port(
    -- Inputs
    a : in std_logic;
    b : in std_logic;
    cin : in std_logic;

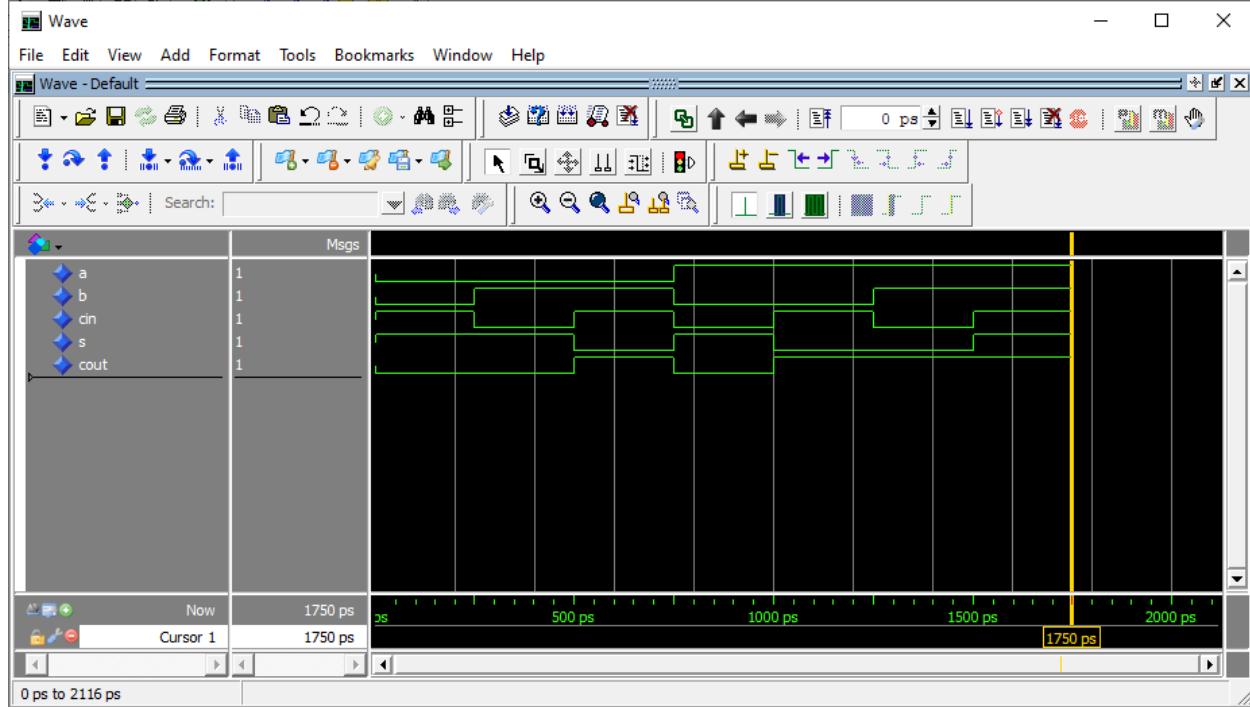
    -- Outputs
    s : out std_logic;
    cout : out std_logic
);
end entity;

architecture rtl of adder1bit is
begin
    -- calculating sum and carry out
```

```

s  <= a xor b xor cin;          -- sum bit
cout <= (a and b) or (a and cin) or (b and cin); -- carry out bit
end architecture;

```



Εικόνα 1: Αποτέλεσμα του adder1bit από το testbench adder1bitTb.vhd

Χρησιμοποιώντας το παραπάνω κύκλωμα του πλήρη αθροιστή γράψτε τον κώδικα περιγραφής για τον παράλληλο αθροιστή των 8 bit. Να δώσετε όνομα στην entity adder8bit.

ΣΗΜΕΙΩΣΗ Μπορείτε να κάνετε χρήση της εντολής generate για να καλέσετε το κύκλωμα του πλήρη αθροιστή 8 φορές.

Γράψτε εδώ το πρόγραμμά σας:

Πρόγραμμα 2: Ο παράλληλος αθροιστής.

```

library ieee;
use ieee.std_logic_1164.all;

```

```
entity adder8bit is
```

```
port(
    -- Inputs
```

```

    a      : in std_logic_vector(7 downto 0);
    b      : in std_logic_vector(7 downto 0);
    cin   : in std_logic;

    -- Outputs
    s    : out std_logic_vector(7 downto 0);
    cout : out std_logic
);

end entity;

```

architecture rtl of adder8bit is

```

-- Internal signal
signal carry : std_logic_vector(7 downto 0);

begin

FA0: entity work.adder1bit
    port map(
        a      => a(0),
        b      => b(0),
        cin   => cin,
        s      => s(0),
        cout  => carry(0)
);

```

FA1: entity work.adder1bit

```

    port map(
        a      => a(1),
        b      => b(1),
        cin   => carry(0),

```

```
    s      => s(1),  
    cout => carry(1)  
);
```

FA2: entity work.adder1bit

```
port map(  
    a      => a(2),  
    b      => b(2),  
    cin  => carry(1),  
    s      => s(2),  
    cout => carry(2)  
);
```

FA3: entity work.adder1bit

```
port map(  
    a      => a(3),  
    b      => b(3),  
    cin  => carry(2),  
    s      => s(3),  
    cout => carry(3)  
);
```

FA4: entity work.adder1bit

```
port map(  
    a      => a(4),  
    b      => b(4),  
    cin  => carry(3),  
    s      => s(4),  
    cout => carry(4)  
);
```

FA5: entity work.adder1bit

```
port map(
    a      => a(5),
    b      => b(5),
    cin   => carry(4),
    s      => s(5),
    cout  => carry(5)
);
```

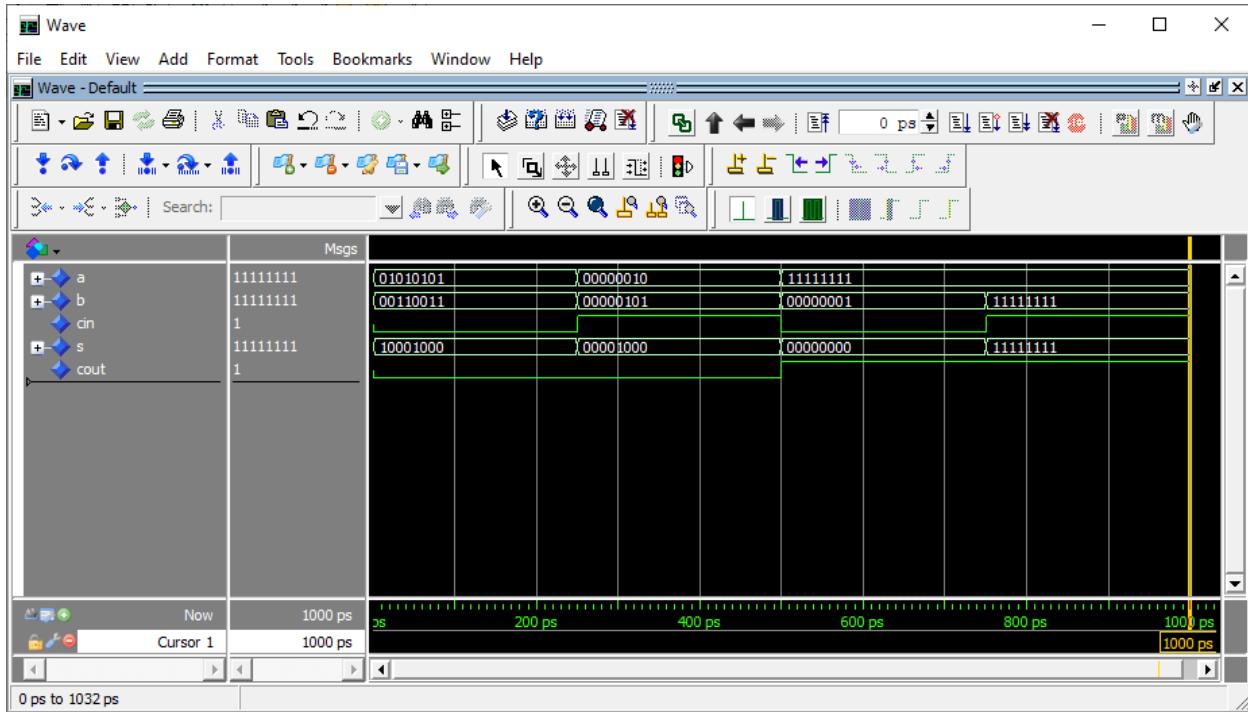
FA6: entity work.adder1bit

```
port map(
    a      => a(6),
    b      => b(6),
    cin   => carry(5),
    s      => s(6),
    cout  => carry(6)
);
```

FA7: entity work.adder1bit

```
port map(
    a      => a(7),
    b      => b(7),
    cin   => carry(6),
    s      => s(7),
    cout  => cout
);
```

end architecture;



Εικόνα 2: Αποτέλεσμα του adder8bit από το testbench **adder8bitTb.vhd**

Πολυπλέκτες

Οι πολυπλέκτες που θα χρησιμοποιηθούν είναι τέσσερις συνολικά, από τους οποίους οι δύο πρώτοι αφορούν την αριθμητική μονάδα (Arithmetic Unit) και ο τρίτος την Λογική μονάδα (Logic Unit). Πιο αναλυτικά θα χρησιμοποιηθούν: α) ένας πολυπλέκτης 2-σε-1 για την επιλογή είτε του περιεχομένου του AC, είτε του σήματος "00000000", β) ένας πολυπλέκτης 3-σε-1 για την επιλογή ενός από τα τρία σήματα BUS, BUS' και "00000000", γ) ένας πολυπλέκτης 4-σε-1 για την επιλογή λογικής πράξης (and, or, xor και not) και δ) ένας πολυπλέκτης 2-σε-1 για την επιλογή μιας από τις εξόδους, είτε της Αριθμητικής είτε της Λογικής μονάδας.

Θα πρέπει να σημειωθεί ότι ο πολυπλέκτης 3-σε-1 στην πραγματικότητα είναι 4-σε-1, αφού για την επιλογή ενός από τρία σήματα εισόδου απαιτούνται δύο σήματα ελέγχου. Επιπλέον, οι πολυπλέκτες 4-σε-1 και 3-σε-1 είναι πανομοιότυποι (4 είσοδοι των 8-bits & 2 σήματα ελέγχου), όπως και οι δύο πολυπλέκτες 2-σε-1 (2 είσοδοι των 8-bits & 1 σήμα ελέγχου). Αυτό σημαίνει ότι θα μπορούσε να γραφεί κώδικας μόνο για αυτά τα δύο είδη πολυπλεκτών, τα οποία θα χρησιμοποιούνταν δύο φορές το καθένα. Να δώσετε όνομα στις entities mux4 και mux2 αντίστοιχα.

Γράψτε τον κώδικα για ένα πολυπλέκτη 2 σε 1.

Γράψτε εδώ το πρόγραμμά σας:

Πρόγραμμα 3: Ο πολυπλέκτης 2 σε 1.

library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

```
--=====
-- This component is a 2 to 1 mux which takes two unsigned
-- 8-bit numbers (Sig1, Sig2) and depending on the selector
-- (Sel), it passes the number the Sel is set at.
=====
```

```
entity mux2 is
```

```
    port(
        -- Inputs
        Sig1 : in std_logic_vector(7 downto 0);
        Sig2 : in std_logic_vector(7 downto 0);

        Sel : in std_logic;

        -- Outputs
        Output : out std_logic_vector(7 downto 0)
    );
end entity;
```

```
architecture rtl of mux2 is
```

```
begin
```

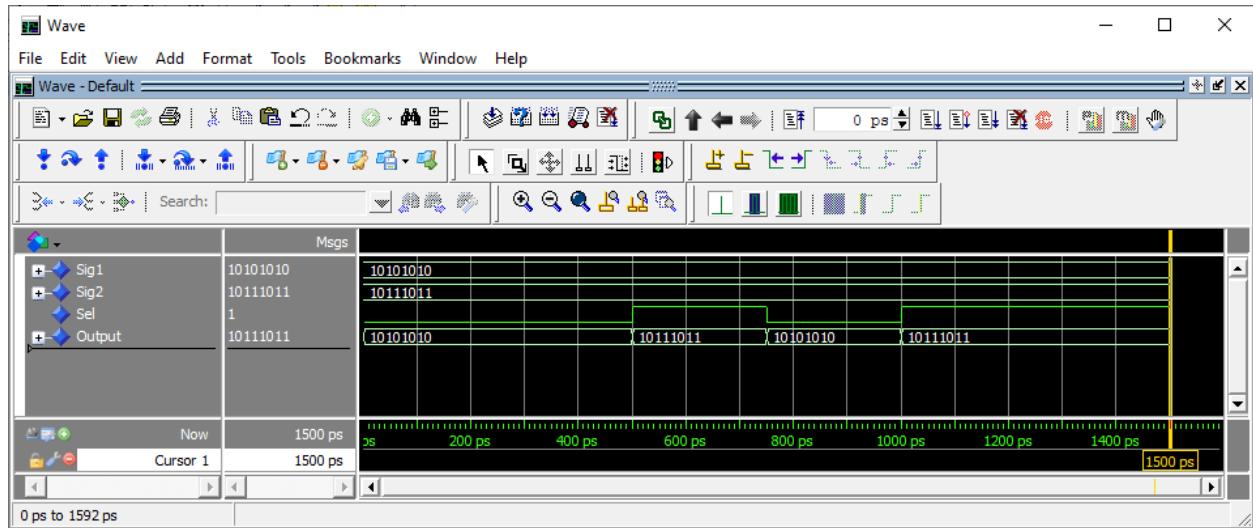
```
    process(Sig1, Sig2, Sel) is
        begin
            -- 2to1 Mux logic.
            case Sel is
                when '0' =>
                    Output <= Sig1;
                when '1' =>
```

```

        Output <= Sig2;
when others =>
        Output <= (others => 'X');
end case;
end process;

end architecture;

```



Εικόνα 3: Αποτέλεσμα του mux2 από το testbench **mux2Tb.vhd**

Γράψτε τον κώδικα για ένα πολυπλέκτη 4 σε 1.

Γράψτε εδώ το πρόγραμμά σας:

Πρόγραμμα 4: Ο πολυπλέκτης 2 σε 1.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

```

```

=====
-- This component is a 4 to 1 mux which takes two unsigned
-- 8-bit numbers (Sig1, Sig2) and depending on the selector
-- (Sel), it passes the number the Sel is set at.
=====
```

```

entity mux4 is
    port(
        -- Inputs
        Sig1    : in std_logic_vector(7 downto 0);
        Sig2    : in std_logic_vector(7 downto 0);
        Sig3    : in std_logic_vector(7 downto 0);
        Sig4    : in std_logic_vector(7 downto 0);

        Sel     : in std_logic_vector(1 downto 0);

        -- Outputs
        Output : out std_logic_vector(7 downto 0)
    );
end entity;

```

```

architecture rtl of mux4 is
begin

    process(Sig1, Sig2, Sig3, Sig4, Sel) is
    begin

        -- 4to1 Mux logic.

        case Sel is
            when "00" =>
                Output <= Sig1;
            when "01" =>
                Output <= Sig2;
            when "10" =>
                Output <= Sig3;
            when "11" =>

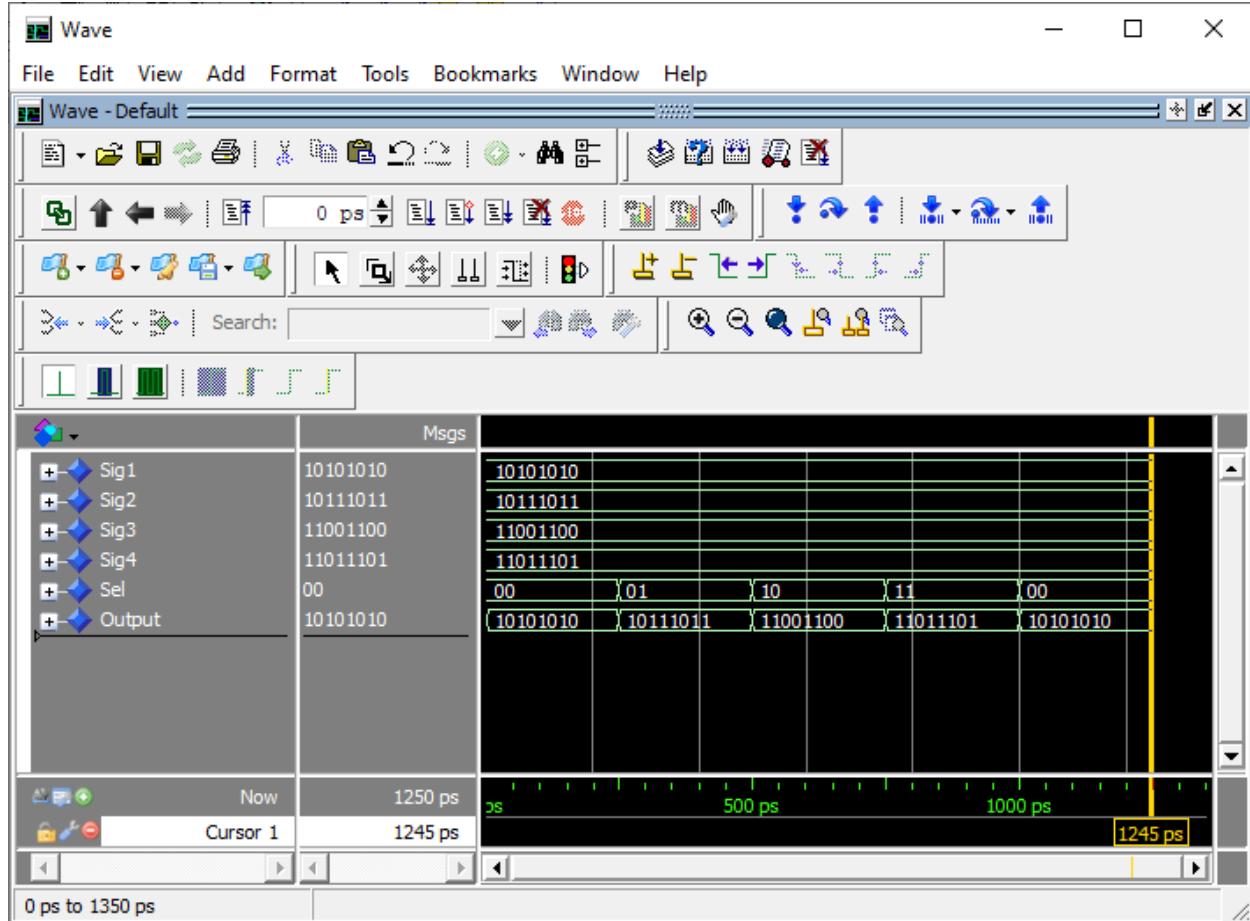
```

```

        Output <= Sig4;
when others =>
        Output <= (others => 'X');
end case;
end process;

end architecture;

```



Εικόνα 4: Αποτέλεσμα του mux4 από το testbench **mux4Tb.vhd**

Αριθμητική & Λογική Μονάδα

Έχοντας ολοκληρώσει τη συγγραφή του κώδικα για τα επιμέρους στοιχεία που συνθέτουν την ALU και αφού όλα συγκεντρωθούν σε μία βιβλιοθήκη, μπορεί πλέον να γραφεί το συνολικό πρόγραμμα περιγραφής της. Τα σήματα ελέγχου της (ALUS 1, ALUS 2, ... ALUS 7), θεωρούνται σαν μία ψηφιολέξη των 7-bits, όπου κάθε ένα bit αντιστοιχεί σε ένα σήμα από αυτά.

Γράψτε τον κώδικα για τη βιβλιοθήκη (package), με το όνομα **alulib**, η οποία θα περιέχει τα επιμέρους στοιχεία που συνθέτουν την ALU.

Γράψτε εδώ το πρόγραμμά σας:

Πρόγραμμα 5: βιβλιοθήκη στοιχείων για την ALU.

Με βάση το σκελετό που ακολουθεί (πρόγραμμα 6) γράψτε τον κώδικα περιγραφής για ALU

Γράψτε εδώ το πρόγραμμά σας:

Πρόγραμμα 6: Αριθμητική & Λογική Μονάδα.

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;
```

entity alu is

```
generic (n : integer := 8); -- This variable can change to reduce/increase the bits.  
port (  
    ac  : in std_logic_vector(n-1 downto 0);  
    db  : in std_logic_vector(n-1 downto 0);  
    alus : in std_logic_vector(7 downto 1);  
    dout : out std_logic_vector(n-1 downto 0)  
);  
end alu;
```

architecture rtl of alu is

```
-- Internal signals  
signal mux2_out1 : std_logic_vector(n-1 downto 0);  
signal mux2_out2 : std_logic_vector(n-1 downto 0);  
signal mux4_out1 : std_logic_vector(n-1 downto 0);  
signal mux4_out2 : std_logic_vector(n-1 downto 0);  
signal adder_out : std_logic_vector(n-1 downto 0);
```

```
-- Logic signals  
signal and_out : std_logic_vector(n-1 downto 0);  
signal or_out : std_logic_vector(n-1 downto 0);  
signal xor_out : std_logic_vector(n-1 downto 0);  
signal not_out : std_logic_vector(n-1 downto 0);
```

```
-- Inverted signals  
signal db_inv : std_logic_vector(n-1 downto 0);
```

```
begin
```

```
-- Here we begin to write the main code.
```

```
-- Logic Unit outputs  
and_out <= ac and db;  
or_out <= ac or db;  
xor_out <= ac xor db;  
not_out <= not(ac);
```

```
-- Inverted db signal  
db_inv <= not(db);
```

```
-- Arithmetic Unit  
i_Mux2_AU: entity work.mux2(rtl)  
port map(  
    Sig1      => (others => '0'),  
    Sig2      => ac,  
    Sel       => alus(1),  
    Output    => mux2_out1
```

```
);
```

```
i_Mux4_AU: entity work.mux4(rtl)
port map(
    Sig1      => (others => '0'),
    Sig2      => (others => '0'),
    Sig3      => db,
    Sig4      => db_inv,
    Sel       => alus(3 downto 2),
    Output    => mux4_out1
);
```

```
i_adder8bit_AU: entity work.adder8bit(rtl)
port map(
    a          => mux2_out1,
    b          => mux4_out1,
    cin       => alus(4),
    s          => adder_out,
    cout      => open
);
```

```
-- Logic Unit
```

```
i_Mux4_LU: entity work.mux4(rtl)
port map(
    Sig1      => and_out,
    Sig2      => xor_out,
    Sig3      => or_out,
    Sig4      => not_out,
    Sel       => alus(6 downto 5),
```

```

        Output => mux4_out2
    );
-- Ouput of the circuit
i_Mux2_OUT: entity work.mux2(rtl)
port map(
    Sig1    => adder_out,
    Sig2    => mux4_out2,
    Sel     => alus(7),
    Output  => mux2_out2
);
dout <= mux2_out2;

end rtl;

```

Εξομοίωση Αριθμητικής & Λογικής Μονάδας

Το επόμενο στάδιο περιλαμβάνει την εξομοίωση της ALU με τον Waveform Editor με σκοπό τον έλεγχο της λειτουργίας της. Σαν σήματα εισόδου εκτός των σημάτων ελέγχου, θεωρούνται δύο τυχαίοι αριθμοί, οι "01001001" και "10011001" που αντιστοιχούν στις δύο εισόδους της ALU (ac & db). Στον πίνακα που ακολουθεί, παρουσιάζονται τα θεωρητικά αναμενόμενα αποτελέσματα για κάθε μια από τις διεργασίες που σχετίζονται με την ALU και τον AC.

State	ALUS	result	state	ALUS	Result
AND	1000000	00001001	INAC	0001001	01001010
OR	1100000	11011001	ADD	0000101	11100010
NOT	1110000	10110110	SUB	0001011	10110000
XOR	1010000	11010000	MOV	0000100	10011001
CLAC	0000000	00000000	LDAC 5	0000100	00000000

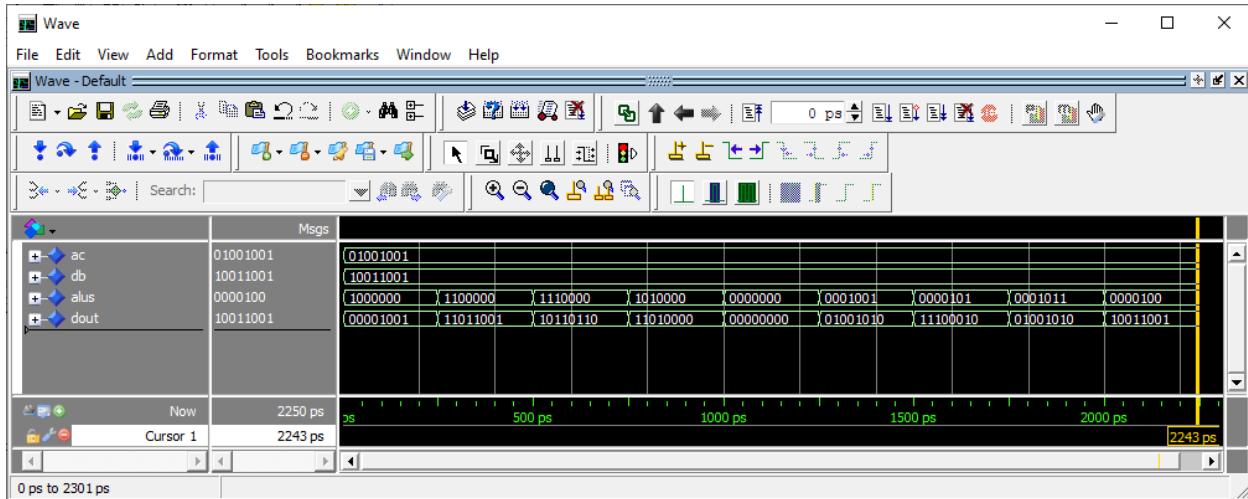
Πίνακας 1: Θεωρητικά αποτελέσματα για τις διεργασίες που σχετίζονται με την ALU και τον AC.

Με οδηγό την άσκηση 1 δημιουργήστε ένα καινούργιο project και εξομοίωστε τη λειτουργία της ALU με τη βοήθεια του Waveform Editor και επιβεβαιώστε τα θεωρητικά αποτελέσματα.

ΠΡΟΣΟΧΗ: Εισάγετε τα nodes alus ξεχωριστά δηλαδή alus[1] έως alus[7] για να μην έχετε προβλήματα (error) κατά την εξομοίωση.

Τοποθετήστε εδώ τις κυματομορφές σας:

Εικόνα 5: Κυματομορφές εξομοίωσης της ALU



Εικόνα 6: Αποτέλεσμα του alu από το testbench **aluTb.vhd**

Τα ίδια αποτελέσματα θα βγουν τρέχοντας τον κώδικα στο Quartus Prime Lite, απλά για να γλυτώσω φωτογραφίες έκανα την προσομοίωση στο ModelSim με ένα testbench που δείχνει τα διαφορετικά alu και βγάζει το ίδιο αποτέλεσμα όπως με τον πίνακα παραπάνω.

Έλεγχος στην αναπτυξιακή πλατφόρμα DE10Lite

Δημιουργήστε δύο αρχεία κώδικα vhdl , με όνομα alu_dut_lib.vhd και alu_dut.vhd, τα οποία περιέχουν τα προγράμματα 7 και 8 αντίστοιχα. Προσθέστε τα στο project σας και ορίστε ως top level entity το αρχείο alu_dut.vhd.

```
library ieee ;
use ieee.std_logic_1164.all ;

package alu_dut_lib is

component alu is
generic (n : integer := 8);
port ( ac      : in std_logic_vector(n-1 downto 0) ;
       db      : in std_logic_vector(n-1 downto 0) ;
       alus    : in std_logic_vector(7 downto 1) ;
       dout    : out std_logic_vector(n-1 downto 0) );
end component ;

end package alu_dut_lib;
```

Πρόγραμμα 7: βιβλιοθήκη που περιέχει την ALU.

```

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_unsigned.all ;
use work.alu_dut_lib.all;

entity alu_dut is
generic (n : integer := 8);
port ( alus      : in std_logic_vector(7 downto 1) ;
       dout      : out std_logic_vector(n-1 downto 0) );
end alu_dut ;

architecture arc of alu_dut is

signal ac, db : std_logic_vector(n-1 downto 0) ;
begin

ac  <= "01001001";
db  <= "10011001";

alu_i: alu port map(ac,db,alus,dout);

end arc;

```

Πρόγραμμα 8: κώδικας ελέγχου για την ALU..

Ρυθμίστε τα pin της ALU μέσω του μενού **Assignments|Pin Planer** ώστε να οδηγήσετε τα σήματα εισόδου alus[1] - alus[7] στα sw[0] - sw[6] και εξόδου dout[0] - dout[7] στα led[0] - led[7] της πλακέτας DE10Lite.

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate
in alus[7]	Input	PIN_A13	7	B7_N0	PIN_T3	3.3-V LVTTL		8mA (default)	
in alus[6]	Input	PIN_B12	7	B7_N0	PIN_U4	3.3-V LVTTL		8mA (default)	
in alus[5]	Input	PIN_A12	7	B7_N0	PIN_U1	3.3-V LVTTL		8mA (default)	
in alus[4]	Input	PIN_C12	7	B7_N0	PIN_N1	3.3-V LVTTL		8mA (default)	
in alus[3]	Input	PIN_D12	7	B7_N0	PIN_U5	3.3-V LVTTL		8mA (default)	
in alus[2]	Input	PIN_C11	7	B7_N0	PIN_V3	3.3-V LVTTL		8mA (default)	
in alus[1]	Input	PIN_C10	7	B7_N0	PIN_U3	3.3-V LVTTL		8mA (default)	
out dout[7]	Output	PIN_D14	7	B7_N0	PIN_P8	3.3-V LVTTL	2	8mA (default)	
out dout[6]	Output	PIN_E14	7	B7_N0	PIN_P1	3.3-V LVTTL	2	8mA (default)	
out dout[5]	Output	PIN_C13	7	B7_N0	PIN_N9	3.3-V LVTTL		8mA (default)	
out dout[4]	Output	PIN_D13	7	B7_N0	PIN_W1	3.3-V LVTTL		8mA (default)	
out dout[3]	Output	PIN_B10	7	B7_N0	PIN_V1	3.3-V LVTTL	2	8mA (default)	
out dout[2]	Output	PIN_A10	7	B7_N0	PIN_N8	3.3-V LVTTL	2	8mA (default)	
out dout[1]	Output	PIN_A9	7	B7_N0	PIN_R7	3.3-V LVTTL	2	8mA (default)	
out dout[0]	Output	PIN_A8	7	B7_N0	PIN_U2	3.3-V LVTTL	2	8mA (default)	
<<new node>>									

Εικόνα 7: ρυθμίσεις pin για την ALU

Ελέγξτε τη λειτουργία της ALU με τη βοήθεια των διακοπτών και επιβεβαιώστε τα θεωρητικά αποτελέσματα (πίνακας 1).