Δημήτρης Καρατζάς **icsd13072**
Νίκος Κατσιώπης **icsd13076**

ΑΝΑΦΟΡΑ 1ΗΣ ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ ΣΤΗ ΣΧΕΔΙΑΣΗ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΕΡΩΤΗΜΑ 1: ΥΛΟΠΟΙΗΣΗ FA ΜΕ STRUCTURAL ΚΑΙ BEHAVIORAL ΤΡΟΠΟ

Απάντηση:
I) Structural

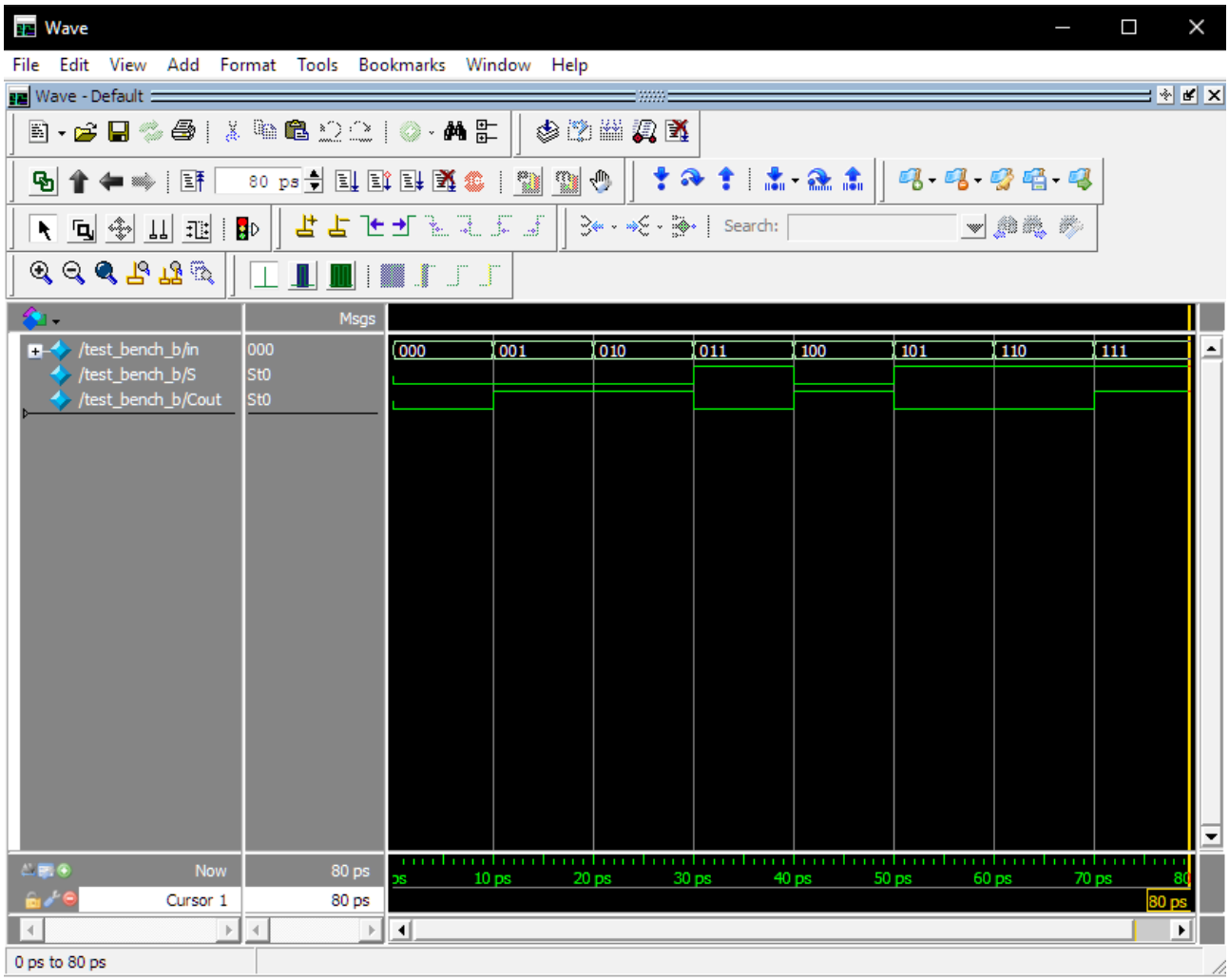
```
module FA_s(
    input A,
    input B,
    input Cin,
    output Cout,
    output S
);
    wire a_cin_and;
    wire a_b_and;
    wire b_cin_and;
    xor(S, A,B,Cin);
    and(a_b_and, A,B);
    and(b_cin_and, B, Cin);
    and(a_cin_and, A, Cin);
    or(Cout, a_cin_and, a_b_and, b_cin_and);
endmodule
```

II) Behavioral

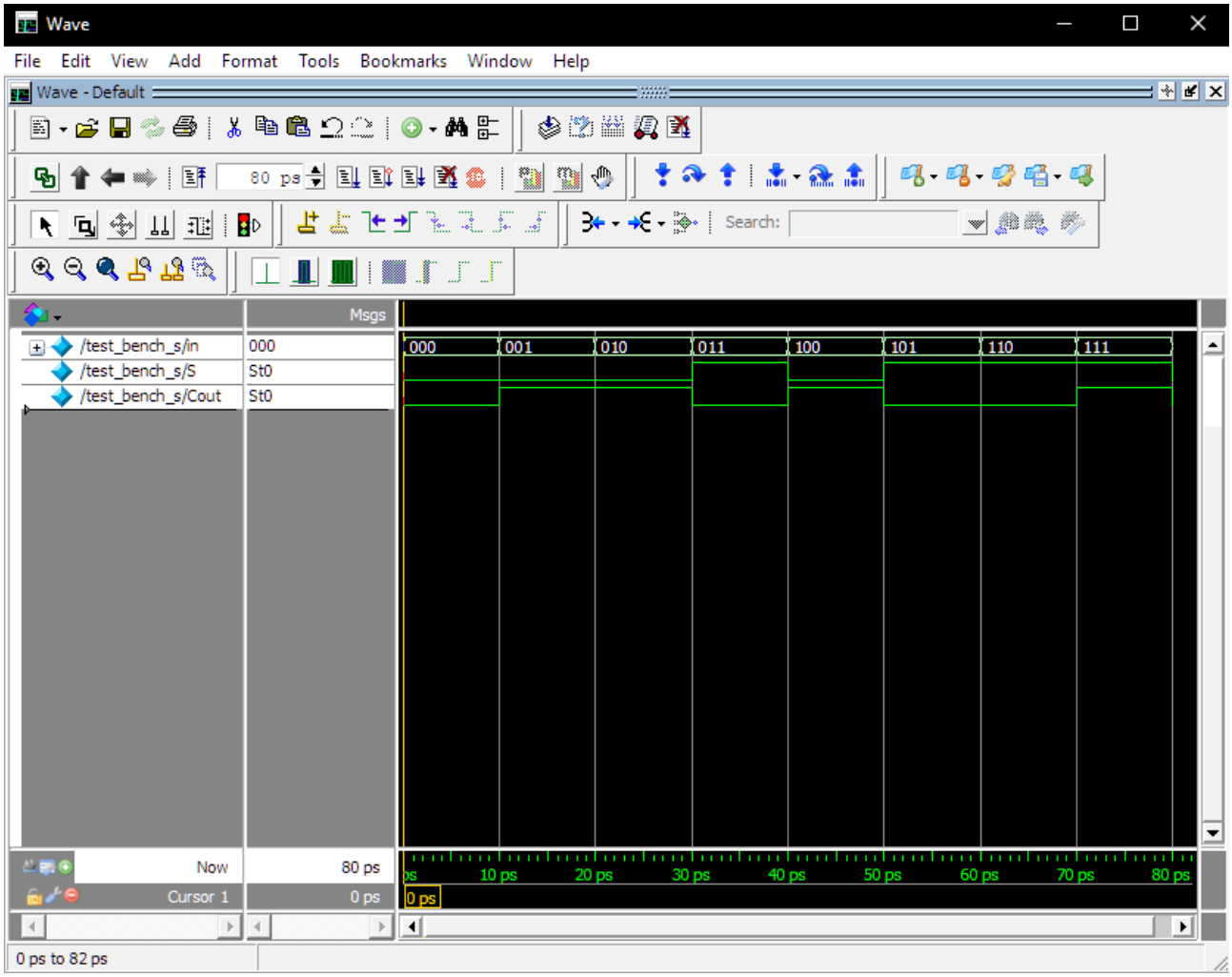
```
module FA_b(
    input A,
    input B,
    input Cin,
    output Cout,
    output S
);
    assign {Cout,S} = A + B + Cin;
endmodule
```

ΕΡΩΤΗΜΑ 2: ΕΛΕΓΧΟΣ ΤΩΝ FA ΑΝ ΔΟΥΛΕΥΟΥΝ ΣΩΣΤΑ

Απάντηση:
Χρησιμοποιόντας το test bench που μας δόθηκε, ελέγχουμε και τις 8 εξόδους του κυκλώματος για κάθε είσοδο. Αυτό μας το κάνει το test bench που μας δίνεται, δηλαδή θέτει αρχικά (initial) την είσοδο (in) να είναι με 3'b000 (μέγεθος 3, σε binary αναπαράσταση → 3 bit, με τιμή 000) και στη συνέχεια κάθε 10 ps αυξάνει τη τιμή αυτή κατά 1
(δηλαδή στο 0ps → 000, 10ps → 001, ... 80ps → 111),
οπότε βάλαμε να τρέξει το simulation μέχρι τα 80ps διοτι σε εκείνο το σημείο ελέγχονται όλες οι εισόδοι. Οι παρακάτω δυο εικόνες είναι για το **simulation** που τρέξαμε, πρώτα για την behavioral υλοποίηση και έπειτα για την structural. Θεωρητικά (και πρακτικά) πρέπει να βγει το ίδιο αποτέλεσμα που όπως φαίνεται και από τις εικόνες είναι όντως ίδιο. Στη συνέχεια εξηγούμε τα αποτελέσματα.



Simulation για τη **behavioral υλοποίηση**



Simulation για τη **structural υλοποίηση**

Εξήγηση αποτελεσμάτων:
Από τις κυματομορφές φαίνεται σε ποιο χρονικό σημείο (και με ποια είσοδο) επηρεάζονται τα S και Cout, οι εξόδοι δηλαδή. Βλέπουμε πότε έχουν λογική τιμή 1 και πότε 0, οπότε βγάζουμε τον παρακάτω πίνακα αληθείας με βάση το Simulation

in[2]	in[1]	in[0]	S	Cout
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

Παρατηρούμε πως είναι **ίδιος** με τον πίνακα αληθείας ενός FA. Κάνει πρόσθεση δηλαδή τα 3bit στοιχεία. Η διαφορά είναι στην ονομασία των μεταβλητών μόνο, δηλαδή το S δρα ως "carry" (Cout) και το Cout δρα ως "sum" (S) αλλά το αποτέλεσμα είναι το ίδιο.

ΕΡΩΤΗΜΑ 3: ΥΛΟΠΟΙΗΣΗ 4BIT RIPPLE CARRY ADDER

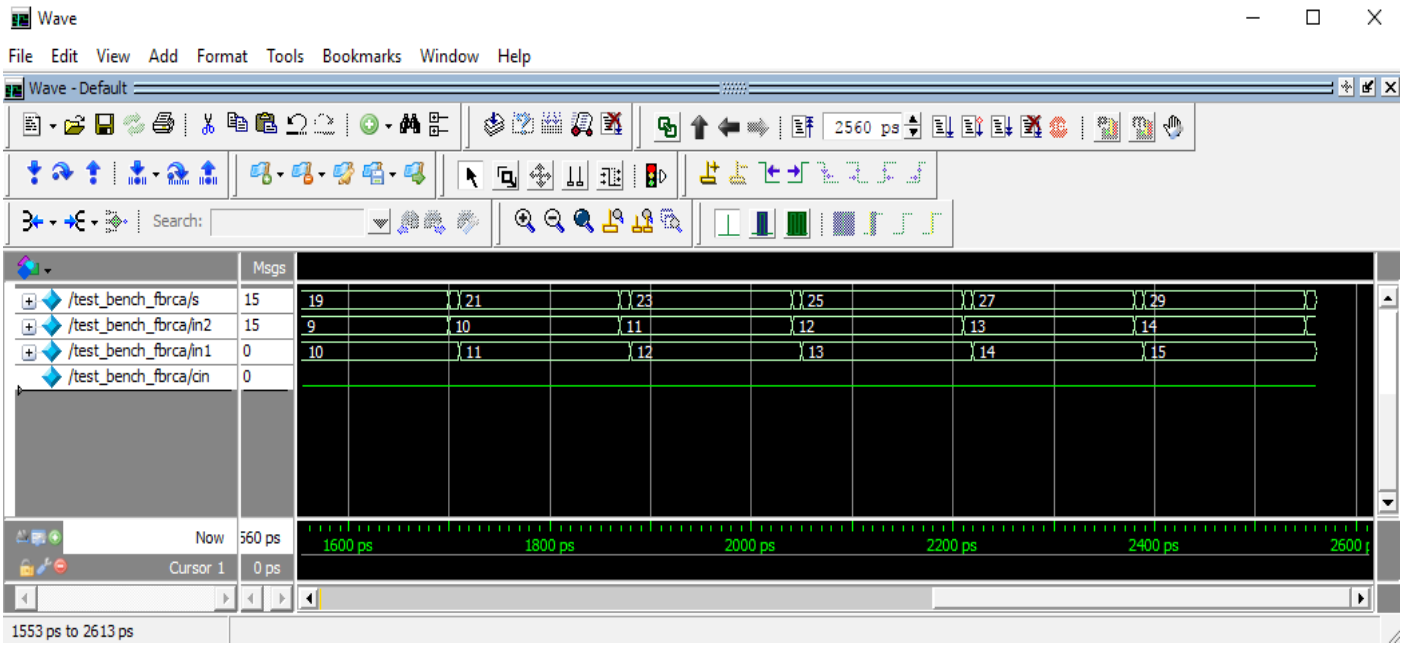
Απάντηση:
Οποιαδήποτε υλοποίηση FA χρησιμοποιήσουμε θα βγάλει το ίδιο αποτέλεσμα, οπότε ας επιλέξουμε τη behavioral.
Για να υλοποιήσουμε έναν 4bit ripple carry adder, θα instantiate 4 FA και θα ενώσουμε το Cout του προηγούμενου με το Cin του επόμενου.
Επίσης πρώτος FA θα έχει Cin=0 και το Cout του τελευταίου είναι και αυτό μέρος του αποτελέσματος.
Τέλος, το αποτέλεσμα της πρόσθεσης δυο n-bitων αριθμών έχει ως αποτέλεσμα ένα n+1 bit αποτέλεσμα. Άρα ο 4bit ripple carry adder επειδή προσθέτει δυο 4bitους, θα έχει ένα 5bit αποτέλεσμα.
Τέλος, για το simulation, θα βάλουμε ένας από τους δυο 4bitους να δίνει όλες τις τιμές του μέχρι να τελειώσουν για τη 1η τιμή του δευτέρου, μετά πάλι το ίδιο για την 2η τιμή του δευτέρου...κτλ, έτσι θα μπορέσουμε να ελέγξουμε όλους τους συνδυασμούς της εισόδου.
Βάζουμε καθυστέρηση 10ps για την 1η είσοδο, (και άρα καθυστέρηση 160ps για τη δεύτερη είσοδο, αφού σε 10*(2^4)ps θα έχουν χρησιμοποιηθεί όλοι οι συνδυασμοί για την 1η είσοδο).

```
module FBRCA(
    input  [3:0] A,
    input  [3:0] B,
    input   Cin,
    output [3:0] S,
    output Cout
);
wire [2:0] temp_cout;
FA_s fa0(A[0], B[0], Cin, temp_cout[0] ,S[0]);
FA_s fa1(A[1], B[1], temp_cout[0], temp_cout[1], S[1]);
FA_s fa2(A[2], B[2], temp_cout[1], temp_cout[2], S[2]);
FA_s fa3(A[3], B[3], temp_cout[2], Cout, S[3]);
endmodule
```

Κώδικας για τον 4-bit Ripple Carry Adder

```
module test_bench_fbrca;

reg[3:0] in1;
reg[3:0] in2;
wire[4:0] s;
reg cin;

FBRCA fbrca0(in1, in2, cin, s[3:0], s[4] );

initial begin
    in1 = 4'b0000;
    in2 = 4'b0000;
    cin = 1'b0;
end
always begin
   #10 in1 = in1 + 1'b1;
   #160 in2 = in2 + 1'b1;
end
endmodule
```

Κώδικας του test bench.



Simulation του FBRCA test bench.

Έχουμε βάλει η αναπαράσταση των αριθμών για τις εισόδους και εξόδους να είναι unsigned ώστε να βγαίνουν σωστά αποτελέσματα. Επίσης για ευκολότερο έλεγχο αναπαριστούμε τους αριθμούς στο δεκαδικό σύστημα, οπότε παρατηρώντας εισόδους και την έξοδο το αποτέλεσμα είναι το αναμενόμενο (πχ στην εικόνα 1+1=1, 1+2=3, 2+3=5 κτλ)

ΕΡΩΤΗΜΑ 4: ΥΛΟΠΟΙΗΣΗ 8BIT RIPPLE CARRY ADDER

Απάντηση:
Στην ουσία είναι η ίδια λογική με την υλοποίηση του 4bit ripple carry adder. Εδώ θα χρησιμοποιήσουμε δυο 4bit ripple carry adders, ο 1ος θα δέχεται τα 4 πιο σημαντικά ψηφία των δυο 8bit αριθμών(είσοδος) και ο δεύτερος τα 4 λιγότερο σημαντικά ψηφία.

```
module EBRCA (
    input  [7:0] A,
    input  [7:0] B,
    input   cin,
    output [7:0] s,
    output Cout
);
wire temp_wire;
FBRCA fbrca0(A[3:0], B[3:0], cin, s[3:0], temp_wire );
FBRCA fbrca1(A[7:4], B[7:4], temp_wire, s[7:4], Cout );
endmodule
```

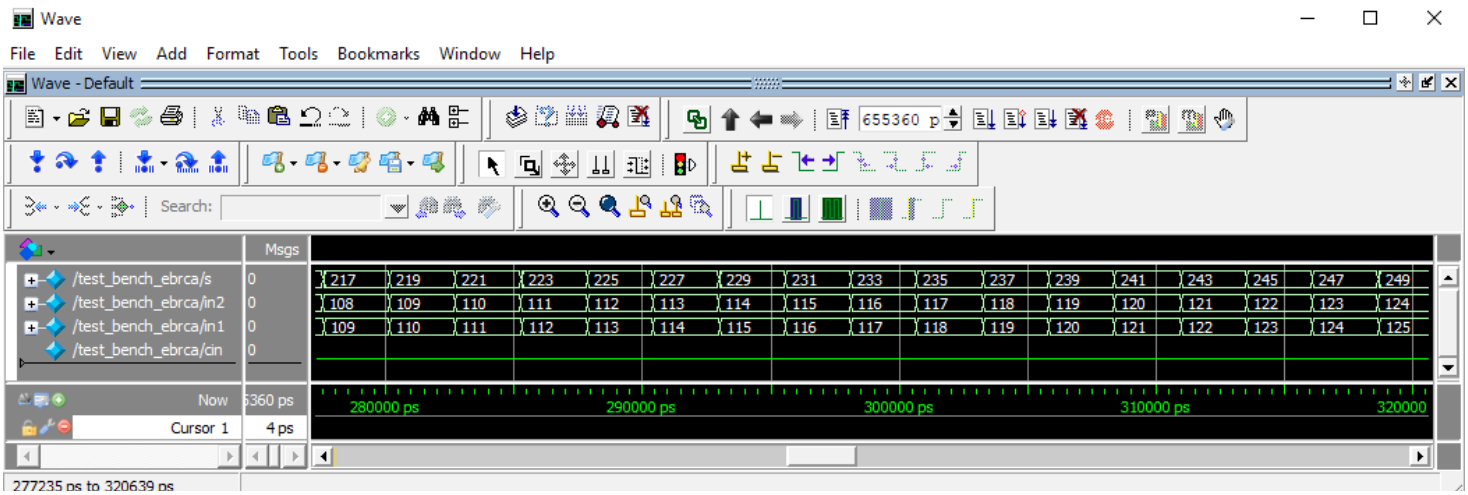
Κώδικας για τον Eight bit Ripple Carry Adder.

```
module test_bench_ebrca;

reg[7:0] in1;
reg[7:0] in2;
wire[8:0] s;
reg cin;

EBRCA ebrca0(in1, in2, cin, s[7:0], s[8] );
initial begin
    in1 = 8'b00000000;
    in2 = 8'b00000000;
    cin = 1'b0;
end
always begin
   #10 in1 = in1 + 1'b1;
   #2560 in2 = in2 + 1'b1;
end
endmodule
```

Κώδικας του test bench

Στο test bench, επειδή ο πρώτος έχουμε βάλει καθυτσέρηση = 10ps, και επειδή έχει 2^8 = 256 πιθανές εισόδους, ο 2ος πρέπει να αλλάξει τιμή κάθε 2560ps.

.



Στιγμιότυπο simulation για τον 8bir Ripple Carry Adder